



NORDUGRID-TECH-25

20/7/2009

ADVANCED USER INTERFACES

ArcGui and the Lunarc Application Portal

Jonas Lindemann*

*jonas.lindemann@lunarc.lu.se

Contents

1	Preface	5
2	ArcGui standalone user interface	7
2.1	Job submission	7
2.2	Job control	10
3	Application Portal Integration	11
3.1	Lunarc Application Portal	11
3.2	User authentication	11
3.3	libarcclient integration	12
3.4	Job definitions	12
3.5	Job management and submission	14
4	Middleware integration module	17
4.1	Creating an ArcClient instance	17
4.2	Creating a managed job description	18
4.3	Job submission	19
4.4	Job status query	20
4.5	Job control	20

Chapter 1

Preface

The **libarcclient** [1] library provides a lot of methods for implementing client functionality in graphical user interfaces as well as Web based interfaces. The library provides bindings for C++, Java and Python. This documents focuses on the Python binding and its use in implementing advanced graphical user interfaces.

Python is a dynamic scripting language which is easy to learn while providing advanced features for implementing larger applications. This documents describes the implementation of a standalone graphical user interface for ARC based on **wxPython** , **ArcGui** , and the integration of **libarcclient** into the Lunar Application Portal.

This document describes how the **libarcclient** library is used

Chapter 2

ArcGui standalone user interface

The ArcGui application is a standalone graphical user interface for simple job submission and job control. The user interface uses the **wxPython** graphical user interface library, which is a Python binding for the **wxWidgets** library. The benefits of using **wxPython** is that the finished application can be run on all available platforms, such as Linux, Mac OS X and Microsoft Windows. In addition to being platform independent, **wxPython** also adapts the appearance of the user interface to the target platform, so that when the application is run on Windows it will look like a native Windows application.

The design of the ArcGui user interface is based on a tabbed window design. Currently there are tabs for a **Generic Job**, **Active Jobs**. The design of the application is flexible, so that it can be easily extended with additional functionality.

The ArcGui application attempts to implement a fully non-blocking user interface, no operation will lock the user interface. This is especially important when dealing with Grid operations, that can take a while to process.

2.1 Job submission

The current implementation of ArcGui only supports a generic job defined entirely by a job description, however the architecture of the application will allow easy extension of functionality. In the generic job tab a job description can be entered either as a XRSL or JSDL file and submitted directly. All parameters are controlled from the job description. Figure 2.1 shows the generic job tab in the user interface.

In the simple job tab a simple shell script based job can be submitted. Input and output files can be selected by using the buttons on the right side. Figure 2.2 show the simple script job tab.

To enable a non-blocking user interface the actual job submission process is encapsulated in a special worker thread *SubmitJobThread*. This thread is spawned when the **Submit** button is pressed. To communicate status updates in the user interface the thread sends events to the main thread. Each thread defines an event that is sent when it has finished; in addition to this, each thread also sends the *UpdateProgressEvent* during the execution to send progress information messages to the main thread.

The *onSubmit()* method creates a *ManagedJobDescription* based on the input in the user interface controls. It then passes the job description class to an instance of a *SubmitJobThread* instance. The complete event method is shown in the following code.

```
def onSubmit(self, event): # wxGlade: ArcWindow.<event_handler>
    """
    Event handler: Submit a job.
    """

    # Create job description

    job = ManagedJobDescription();
```

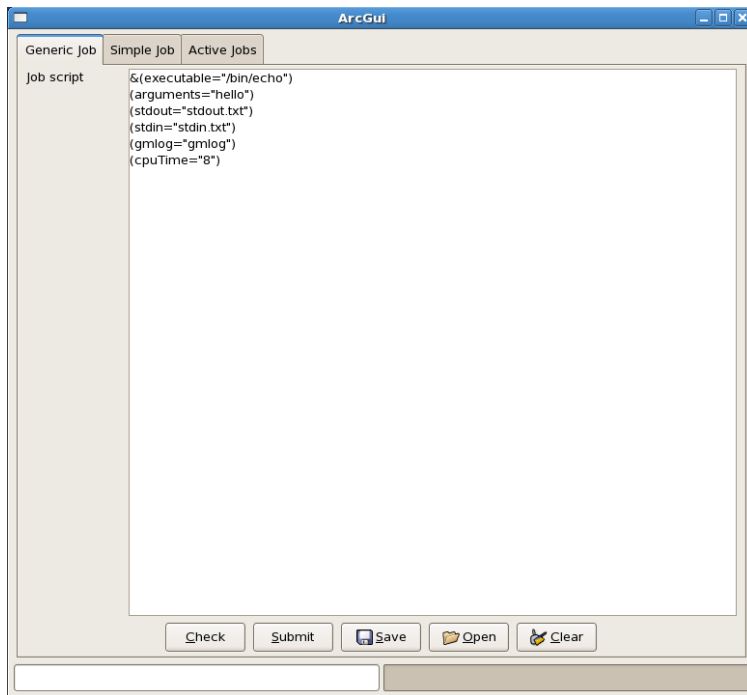


Figure 2.1: Generic job tab

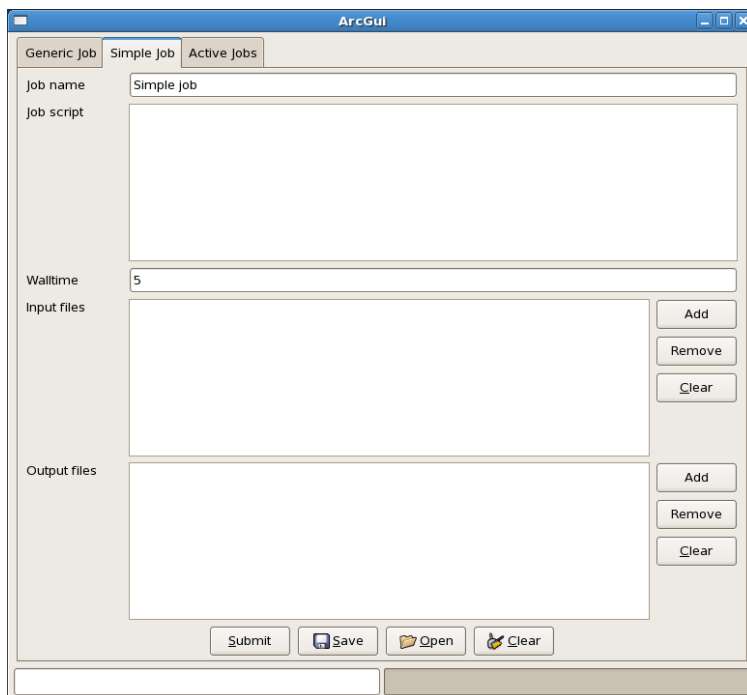


Figure 2.2: Simple script job tab


```

job.JobName = str(self.jobNameText.GetValue())
job.TotalWallTime = arc.Period(str(self.jobWallTimeText.GetValue()),arc.PeriodMinutes)
job.Executable = "/bin/sh"
job.addArgument("run.sh")
job.addInputFile("run.sh")
job.Output = "stdout.txt"
job.Error = "stderr.txt"
job.TotalWallTime = arc.Period("5",arc.PeriodMinutes)
job.Print(True)

# Create run-script

jobScriptFile = open("run.sh", "w")
jobScriptFile.write(self.jobScriptText.GetValue())
jobScriptFile.close()

# Start job submission thread

worker = SubmitJobThread(self, self.__arcClient, job)
worker.start()
self.__progressTimer.Start(100)

```

The job submission thread takes an instance of the ArcClient instance and a job description instance. To enable sending of status events the *updateProgress* property of the ArcClient class is assigned a class method of the thread, *doProgress*, which sends an event to the main thread to update status information. Then *run()* method then does brokering and submission and finally sends an event to indicate that processing has finished.

```

class SubmitJobThread(threading.Thread):
    """
    Worker thread for submitting a job.
    """
    def __init__(self, parent, arcClient, job):
        threading.Thread.__init__(self)
        self.__parent = parent
        self.__arcClient = arcClient
        self.__job = job
        self.__arcClient.updateProgress = self.doProgress

    def run(self):
        self.__arcClient.debugLevel = arc.WARNING
        self.__arcClient.findTargets()
        self.__arcClient.filterTargets(self.__job)
        success = self.__arcClient.submit(self.__job)
        evt = SubmitJobDoneEvent(EVT_KILL_JOBS_DONE_TYPE, -1, success)
        wx.PostEvent(self.__parent, evt)

    def doProgress(self, message):
        evt = UpdateProgressEvent(EVT_PROGRESS_UPDATE_TYPE, -1, message)
        wx.PostEvent(self.__parent, evt)

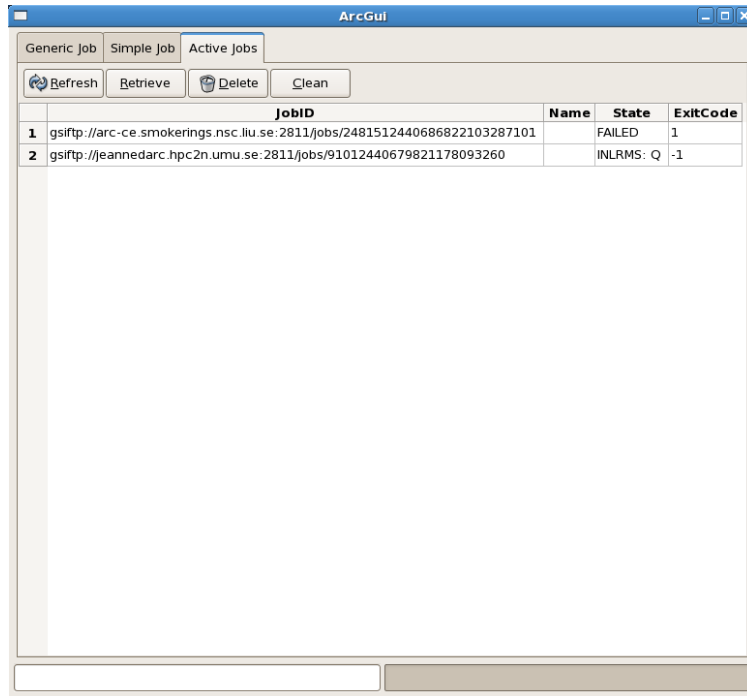
```

2.2 Job control

Active jobs are presented in the **Active Jobs** tab in an interactive table. Available actions are shown as buttons in the top of the tab. To invoke an action, jobs are selected by clicking in the left column. Multiple jobs can be selected by using modifier keys.

When an action is invoked, the selected jobs in the table are collected in a **JobController** and the action is invoked on the job controller instance.

Jobs in the table can be grouped by selecting the columns in topmost row. Figure 2.3 shows then job control table.



The screenshot shows the ArcGui interface with the 'Active Jobs' tab selected. The interface includes a toolbar with 'Refresh', 'Retrieve', 'Delete', and 'Clean' buttons. Below the toolbar is a table with the following data:

	JobID	Name	State	ExitCode
1	gsiftp://arc-ce.smokerings.nsc.liu.se:2811/jobs/2481512440686822103287101		FAILED	1
2	gsiftp://jeannedarc.hpc2n.umu.se:2811/jobs/91012440679821178093260		INLRMS: Q	-1

Figure 2.3: Job control tab

Chapter 3

Application Portal Integration

The Lunarc Application Portal is an ongoing development to provide a framework for developing portals for Grid resources. Existing versions of the portal also used ARC for job submission, job monitoring and job control. However, these versions mostly used command line tools and some classes from the **arclib** library. The aim of this project is to implement all the job handling routines using the newly developed **libarclient** instead of using the error prone parsing of command line output.

3.1 Lunarc Application Portal

The Lunarc Application Portal is a lightweight application portal for accessing Grid resources. The portal is implemented using WebWare for Python which is a Python based application server. The WebWare for Python handles takes care of servlets, session handling and concurrency.

The Lunarc Application portal adds a framework for implementing application specific portals using a plugin based approach. Developers of application plugins provide two classes a Job class (**CustomJob**) and a user interface class (**CustomJobPage**). Job submission, control and monitoring is handled by the portal framework.

The portal implementation uses a file based structure to implement user configuration and storage of user files instead of a database backend. This leads to a portal that is relatively easy to setup and maintain. The file structure is described in table:

3.2 User authentication

In the current version of the portal a user logs to the portal over SSL with his browser certificate. A login prompt is shown and the user logs in with a user name and a password. In the main user interface the user can upload a proxy certificate from the Session/Upload proxy... menu. To validate the portal the **arc.Credential** class is used from the **libarclient** library.

```
cred = arc.Credential(proxyLocation, "", "", "")
period = cred.GetEndTime()-arc.Time()
```

Directory	Description	example
User directories	Storage of user configuration and job definitions	/var/spool/lap
Log files	Log files for portal and WebWare for Python	/var/log/lap
Application instance	WebWare application instance	/opt/lap

Table 3.1: File structure of the Lunarc Application portal

```

if period <= arc.Period():
    [...]
    self.writeln("Proxy has expired.")
    [...]
else:
    [...]
    self.writeln("Proxy is remaining = " + (period.tolongstring()), "Information")
    [...]

```

Authentication information is stored in an **arc.UserConfig** instance in the **Grid.Clients.ArcClient** instance.

3.3 libarcclient integration

To make it easier and more maintainable to integrate the Lunarc Application portal with **libarcclient** a special client class, **Grid.Clients.ArcClient**, was implemented. This class handles all interaction with the **libarcclient** library as well as maintains job lists and user configuration information.

An **ArcClient** instance is created upon user session creation and destroyed when the session is destroyed. WebWare has a special session store which stores session specific information during the session lifetime. The session store is implemented as a python dictionary and can be accessed by the methods **setValue**, **getValue**, **delValue** and **hasValue**. The session store is persistent during the session lifetime, that is if the portal service is shutdown, the session store is serialised to disk before shutdown and loaded again when the service is started again. The **ArcClient** instance is stored in the session store, but not serialised when the service is shutdown. The following code shows how an **ArcClient** instance is created if needed when a servlet is awakened.

```

def awake(self, trans):
    ApplicationSecurePage.awake(self, trans)

    # Instantiate a arc client instance if needed.

    if not self.session().hasValue("arc_client"):
        user = Lap.Session.User(self.session().value('authenticated_user'))
        userDir = user.getDir();
        proxyFilename = os.path.join(userDir, "proxy.pem")
        jobListFilename = os.path.join(userDir, "jobs.xml")
        userConfigurationFilename = os.path.join(userDir, "client.xml")
        self.session().setValue("arc_client",
            ArcClient(proxyFilename,
                jobListFilename, userConfigurationFilename)

```

The **ArcClient** is described in detail in chapter 4.

3.4 Job definitions

The Lunarc Application Portal is based on a job definition concept. A job definition can best be described as a template for how a job for a specific application is submitted to a Grid resource. It contains a template for a run script and maintains the list of files needed to run the job. It is also responsible for creating the job description for the Grid.

The main functionality of a job template is implemented in the **Lap.Job.LapBaseTask** class. This class is used by the portal to maintain input and output files for the job, parameter sweep functionality and job description creation.

A job definition can have a sweep size set, which defines the number of jobs that will be submitted with different parameters. A special task directory is setup for each parameter in the set and results are retrieved and stored the same directory. To use the parameter sweep functionality the user supplies input files with special keywords, which are then replaced by the portal with special values which can be used to calculate the exact sweep parameters. The following code shows how an input file for NumPy is modified with sweep parameters:

```
#!/bin/env python

from numpy import *

sweepSize = %(sweepSize)d
jobName = "%(name)s"
id = %(id)d

matrix = array([id, id])

[...]
```

When an application plugin is developed a special class **CustomTask** is derived from **Lap.Job.LapBaseTask**, which is the base from which job definitions are instantiated. There are two important methods that must be implemented by a derived job definition class, **doCreateRunScript** and **doCreateJobDescription**. The first is responsible for creating the script which is executed on the Grid resources and the second class is used to create a **libarcclient arc.JobDescription** instance which is then used by the portal framework when the job is submitted. The following code shows an example of a job description class, **CustomTask**

```
class CustomTask(Lap.Job.LapBaseTask):
    def __init__(self):
        Lap.Job.LapBaseTask.__init__(self)

        self.description = "NumPy"
        self.taskEditPage = "CustomJobPage"

        # Task specific attributes

        self.__mainFile = ""
        self.packages = []
        self.extraFiles = []

    def doCreateRunScript(self, taskName, taskId):
        """
        Abstract routine responsible for returning a
        run-script for the job.
        """
        return runScriptTemplate % (self.__mainFile)

    def doCreateJobDescription(self, taskName, taskId, taskDir):
        """
        Abstract routines responsible for returning a jobdescription for
        the job.
        """

        # Create a managed job description

        job = ManagedJobDescription();
```

```

job.JobName = str(taskName)
job.TotalWallTime = arc.Period(str(self.cpuTime),arc.PeriodMinutes)
job.Executable = "/bin/sh"
job.addArgument("run.sh")

# Make sure we store the full paths of input files

for inputFile in self.inputFiles.keys():
    url = self.inputFiles[inputFile]
    if url == "":
        fullPath = os.path.join(taskDir, inputFile)
        job.addInputFile(fullPath)
    else:
        job.addInputFile(inputFile, url)

job.Output = "stdout.txt"
job.Error = "stderr.txt"

return job

```

3.5 Job management and submission

An important part of the Lunarc Application Portal is the job management page (**ManageJobPage**). From this page the user manages the created job definitions. Job definitions can be deleted, edited and submitted. The user interface is centered around a dynamic table control, which make it easy to sort job definition as well as select multiple definitions.

The job management page is also responsible for initiating job submission. In this initial version job submission is done in a non-threaded way using and **ArcClient** instance and blocks page rendering until the job has been submitted. Information on the submitted job is stored in the, **job.xml**, file which is stored in the user directory. The **libarcclient** library maintains this file when it submits jobs. Part of the submit method of the job management page is shown in the following example:

```

def submitJob(self):
    """Submit selected job(s) to the grid (action)."""

    [...]

    for jobName in jobNameList:

        jobDir = os.path.join(userDir, "job_%s" % jobName)

        # Read the job task

        taskFile = file(os.path.join(jobDir,"job.task"), "r")
        task = pickle.load(taskFile)
        taskFile.close()

        jobList = task.getJobList()

        # Do brokering

        self.__arcClient.debugLevel = arc.DEBUG
        self.__arcClient.findTargets()

```

```
for job in jobList:
    job.Print(True)
    self.__arcClient.filterTargets(job)
    submitted = self.__arcClient.submit(job)
    if submitted:
        print "Job submitted succesfully."
    else:
        print "Job submission failed."

[...]
```

In the final version of the portal the submission procedure will be threaded and the **libarcclient** parts will be executed by a special submission thread. This will make submission of large parameter jobs much more efficient as well as preventing a blocking Web page. Job submission status will be reported and stored in the task parameter directories. To prevent threading issues status for the submission threads will be reported in separate files.

Chapter 4

Middleware integration module

To hide the complexity and make the integration of the **libarcclient** library easier a special Python integration module, **Grid.Clients** , was implemented. The module contains the **ArcClient** class which implements the necessary commands for integrating with ARC as well as more advanced features such as bulk job handling.

4.1 Creating an ArcClient instance

The **ArcClient** class communicates most of its configuration by a property based interface. That is directly assigning class variables as variables. This can seem like a bad programming practice, but instead of providing get and set methods for each property, Python provides a way of defining properties which behave as variables externally, but internally call get and set methods. The following example shows how this can be implemented in a Python class:

```
class ArcClient(object):
def __init__(self):
...
self.__debugLevel = arc.DEBUG

def setDebugLevel(self, level):
self.__debugLevel = level
    arc.Logger_getRootLogger().setThreshold(self.__debugLevel)

    def getDebugLevel(self):
        return self.__debugLevel

...

debugLevel = property(getDebugLevel, setDebugLevel)
```

In the previous example the debugLevel property is implemented through the **getDebugLevel** and **setDebugLevel** . The user of the instance can then assign the **debugLevel** property as a normal member variable.

```
arcClient.debugLevel = arc.WARNING
```

The property interface provide classes with a more direct and intuitive way of interacting with objects.

The **ArcClient** class is instantiated by calling its constructor. The constructor does not have any parameters. Most configuration options are defined by properties. In the following example an **ArcClient** instance is created and locations for proxy, job list file, download directory and user configuration is set by assigning class properties.

```
arcClient = ArcClient()

uid = os.getuid()
gid = os.getgid()

arcClient.proxyFilename = "/tmp/x509up_u%d" % uid
arcClient.jobListFilename = os.path.abspath("./jobs.xml")
arcClient.userConfigFilename = os.path.abspath("./client.xml")
arcClient.downloadDir = os.path.abspath(".")
```

4.2 Creating a managed job description

The new **libarcclient** library contains a special class, **JobDescription**, which implements a Grid job description. The class is implemented as generic job description and can be used to generate job descriptions for both ARC 0.6.x resources as well as future ARC 1.x based resources. The Python binding for this class is quite expressive and requires a lot of extra steps to create a simple description. To simplify the job creation process a special **ManagedJobDescription** class has been implemented. This class is derived from **arc.JobDescription** and extends this class with methods for adding input and output files as well as adding runtime environment definitions. To illustrate the expressiveness of the **libarcclient JobDescription** class the implementation of the **addInputFile()** method is shown in the following code excerpt:

```
def addInputFile(self, name, url="", keepData = True, isExecutable = False,
    downloadToCache = False, threads = -1):
    """
    Add an input file, name, to the job description.
    """
    inputFile = arc.FileType()
    inputFile.Name = os.path.basename(name)
    inputFile.KeepData = False
    inputFile.IsExecutable = False
    inputFile.DownloadToCache = False
    inputFileSource = arc.SourceType()
    if url=="":
        fullPath = os.path.abspath(name)
        urlRepr = "file://" + fullPath
        inputFileSource.URI = arc.URL(urlRepr)
    else:
        inputFileSource.URI = arc.URL(url)
    inputFileSource.Threads = threads
    inputFile.Source.append(inputFileSource)
    self.File.append(inputFile)
```

By using the **ManagedJobDescription** class a job description can be described by the following lines of code:

```
job = Grid.Clients.ManagedJobDescription()
job.Executable = "/bin/sh"
```

```

job.addArgument("run.sh")
job.addInputFile("run.sh")
job.Output = "stdout.txt"
job.Error = "stderr.txt"
job.addOutputFile("result.txt")
job.TotalWallTime = arc.Period("5",arc.PeriodMinutes)

```

4.3 Job submission

To handle job submission in a efficient way the **ArcClient** separates target selection, brokering and submission into separate methods, **findTargets** , **filterTargets** and **submit** . This makes it possible to reuse the selected targets and eliminate the need to query the information system for each job submission. To find suitable targets the **findTargets()** method uses the **libarcclient TargetGenerator** class. The found targets are then stored in the **self.targets** member variable, so that it can be reused when targets are filtered. The following code shows the **ArcClient**]pythonfindTargets() implementation.

```

def findTargets(self):
    """
    Find possible targets by querying information system.
    """
    self.targets = None
    self.__targetGenerator.GetTargets(0, 1);
    self.targets = self.__targetGenerator.ModifyFoundTargets()

```

The **findTargets()** method only queries the information system for all possible targets. To select a target for submission, job brokering or filtering must be done. This is implemented in **filterTargets()** method in the **ArcClient** class. This method selects a broker and uses this broker to select suitable submission targets. The implementation of this class is shown in the following code:

```

def filterTargets(self, job):
    """
    Return a filtered list of suitable targets based on the
    RandomBroker component.
    """
    chosenBroker = self.loadBroker()
    chosenBroker.PreFilterTargets(self.targets, job)

    target = chosenBroker.GetBestTarget()
    while not target==None:
        target = chosenBroker.GetBestTarget()
        if target!=None:
            self.filteredTargets.append(target)

    return self.filteredTargets

```

A typical job submission with **ArcClient** is shown in the following code:

```

# Brokering

```

```

arcClient.findTargets()
arcClient.filterTargets()

# Submission (reusing target information)

for job in jobs:
    if arcClient.submit(job):
        print "Job submission succeeded."
    else:
        print "Job submission failed."

```

4.4 Job status query

The **JobController** class has methods for querying job status for the jobs it manages. However, to make it more manageable in Python the **ArcClient** class converts the mapped C++ **JobInformation** instances to a dictionary keyed on the job id. The following code shows this conversion in the **updateStatus()** method.

```

def updateStatus(self):

    [...]

    for controller in jobControllers:
        controller.GetJobInformation()
        jobStore = controller.GetJobs()
        for job in jobStore:
            jobId = job.JobID.str()
            if self.jobDict.has_key(jobId):
                try:
                    self.jobDict[jobId]["State"] = job.State
                    self.jobDict[jobId]["Name"] = job.Name
                    self.jobDict[jobId]["Type"] = job.Type
                    self.jobDict[jobId]["JobDescription"] = job.JobDescription
                    [...]

```

When the **updateStatus()** method has been called job information can be retrieved by querying the Python dictionary.

```

for jobId in arcClient.jobDict.keys():
    if arcClient.jobDict[jobId].has_key("Name")
        print arcClient.jobDict["Name"], arcClient["State"]

```

A special method, **sortKeysBy()**, has been added to return a list of keys sorted by a specific field in the job information structure.

4.5 Job control

The **ArcClient** module implements the methods **get()**, **kill()** and **clean()** for controlling active Grid jobs. All methods are implemented using the **JobController** class for invoking operations on lists of jobs. When any of these operations are invoked the job list is reloaded. Job status information must also be reloaded after these operations.

Downloading of jobs will be done to the directory specified by the **downloadDir** property of the **ArcClient** class.

The following code shows an example of how these methods can be used to control active Grid jobs.

```
arcClient.submit(job)
arcClient.updateStatus()

[...]

arcClient.get(arcClient.jobDict.keys())

[...]

arcClient.kill(["gsiftp://...", "gsiftp://..."])
arcClient.clean(["gsiftp://..."])
```


Bibliography

- [1] libarcclient – A Client Library for ARC, M. Ellert, B. Mohn, I. Márton, G. Róczy, NORDUGRID-TECH-20, http://www.nordugrid.org/documents/client_technical.pdf