EGEE

Grid Middleware

HANDOUTS FOR STUDENTS

Date: **November 10, 2008**

Author(s): **Fokke Dijkstra, Jeroen Engelberts, Sjors Grijpink, David Groep, Arnold Meijster, Jeff Templon, Machiel Jansen**

Abstract: These handouts are provided for people to learn how to use the gLite middleware components to submit jobs to the Grid, manage data files and get information about their jobs and the testbed. It is intended for people who have a basic knowledge of the Linux/UNIX operating system and know basic text editor and shell commands.

1. INTRODUCTION

This document leads you through a number of increasingly sophisticated exercises covering aspects of job submission and data management. It is assumed that you are familiar with the basic Linux/UNIX user environment. This document is designed to be accompanied by a series of presentations providing a general overview of Grids and the gLite tools.

This document is based on previous tutorial documents to which the following people have contributed: Kors Bos, Simone Campana, Flavia Donno, Leanne Guy, Patricia Méndez Lorenzo, Antonio Delgado Peris, Mario Reale, Ricardo Rocha, Elisabetta Ronchieri, Roberto Santinelli, Andrea Sciabà, Massimo Sgaravatto, Heinz Stockinger, Kurt Stockinger, Antony Wilson.

1.1. WORKING ENVIRONMENT

In order to make use of the Grid you will need access to a so called User Interface machine (UI). For example, you could request a login account on a UI at one of your local Grid facilities. You can then login to the UI with your username and password via ssh, like in the screendump below:

```
$ ssh demo07@ui.grid.sara.nl
demo07@ui.grid.sara.nl's password:
*******************************************************************************
Welcome to the SARA NL-Grid Matrix User interface
- For information on use see http://www.sara.nl/userinfo/grid/description
- If you have problems or questions please contact grid.support@sara.nl
*******************************************************************************

The Matrix cluster runs gLite-3 software.
The OS is Scientific Linux 3.0.7 (A Redhat Enterprixe Linux 3 clone)

*********************************Last modified: Thu May 17 15:26:35 CEST 2006**
/usr/X11R6/bin/xauth:  creating new authority file /home/demo07/.Xauthority

ui.grid.sara.nl:~
demo07$
```

Once logged in you have access to the relevant Grid toolkits and you may start working on the Grid.

Alternatively, you may want to install your own UI. You could attempt to build the UI setup yourself using your favorite Linux distribution, but as of today this is not a trivial procedure. A more convenient way to install your own User Interface would be to use a dedicated distribution that has the pre-configured UI software on-board, such as the Virtual Lab for e-Science Proof-of-Concept distribution[1], or Vle-PoC for short. For the purpose of this tutorial we have created an image that contains the Vl-e PoC distribution. The image is installed on the machine in front of you and can be booted as a virtual machine.

1.2. GETTING ACCESS TO THE GRID

Before you can make use of the many Grid resources out there (computing, storage) you need a *Grid certificate*. A certificate will allow you to uniquely identify yourself anywhere in the world, much like a passport.

Obtaining a new certificate requires filling out paperwork, jumping through virtual hoops and waiting for approval; this could take anywhere between a couple of hours and a couple of days. So for the sake of this tutorial we've already prepared some certificates that you can use right away, but they give limited access to resources. In fact, they can only be used for the duration of the tutorial.

[1]Please look at http://poc.vl-e.nl/ for detailed information

If you already have a real Grid certificate, you can of course use that one. If not, you should request one as soon as possible, following the procedure given in Appendix A. For the tutorial, use one of the pre-made certificates.

Make sure your certificate files are in the `.globus` subdirectory on your home directory on the UI. There are two files:

usercert.pem the public certificate and

userkey.pem the private key, which should be kept read-only by yourself.

The file modes should be as follows:

```
-rw-r--r--   usercert.pem
-r--------   userkey.pem
```

If necessary, repair permissions with the commands

```
$ chmod 644 ~/.globus/usercert.pem
$ chmod 400 ~/.globus/userkey.pem
```

1.2.1. Creating a proxy

Working on the Grid means letting systems do work on your behalf. This brings a complication, because the work done ‘on your behalf’ must have your trusted identity attached to it. Yet it is unsafe to pass along your entire certificate. As a solution you should create a delegation of your certificate called a *proxy*. This is just like your certificate, only much shorter-lived: typically only twelve hours.

In order to obtain a proxy and use it, the following commands can be used:

`voms-proxy-init`	you need to type the passphrase
`voms-proxy-info`	gives information about the current proxy
`voms-proxy-destroy`	destroys the proxy for this session
`voms-proxy-xxx -help`	shows the usage of the command voms-proxy-xxx

Examples:

```
$ voms-proxy-init --voms tutor
Your identity: /O=dutchgrid/O=users/O=sara/CN=Fokke Dijkstra
Enter GRID pass phrase:
Creating temporary proxy ........................................ Done
Contacting  voms.grid.sara.nl:30014 [/O=dutchgrid/O=hosts/OU=sara.nl/CN=voms.grid.sara.nl] "tutor" Done
Creating proxy .............................................. Done
Your proxy is valid until Sat Jun 24 00:47:02 2006
$ voms-proxy-info
subject   : /O=dutchgrid/O=users/O=sara/CN=Fokke Dijkstra/CN=proxy
issuer    : /O=dutchgrid/O=users/O=sara/CN=Fokke Dijkstra
identity  : /O=dutchgrid/O=users/O=sara/CN=Fokke Dijkstra
type      : proxy
strength  : 512 bits
path      : /tmp/x509up_u503
timeleft  : 11:59:21
VO        : tutor
subject   : /O=dutchgrid/O=users/O=sara/CN=Fokke Dijkstra
issuer    : /O=dutchgrid/O=hosts/OU=sara.nl/CN=voms.grid.sara.nl
attribute : /tutor/Role=NULL/Capability=NULL
timeleft  : 11:59:21
$voms-proxy-destroy
Would remove /tmp/x509up_u503
```

1.3. GETTING THE EXERCISES

Some material for the exercises has been prepared in advance and you can copy it (e.g. with `wget`) to your home directory on the UI machine. You can find the exercises at the following URL:

https://hpcv.projects.sara.nl/wiki/images/7/7d/Tut_ex_2008-11.tgz

The files can be downloaded and extracted with:

```
$ wget https://hpcv.projects.sara.nl/wiki/images/7/7d/Tut_ex_2008-11.tgz
$ tar xzf Tut_ex_2008-11.tgz
$ cd exercises
```

The target directory will be **exercises**.

2. Job submission

2.1. Introduction

Suppose you have a problem which takes weeks, maybe months of computing time. If there is a way to divide the problem into smaller chunks, then all these could run on a different machine simultaneously. A Grid allows you to do just that. Each chunk of your bigger problem is called a job and will be described in a so called job description language.

When you submit a job to the Grid it will be sent to the Workload Management System (WMS). This system will then schedule your job and send it to a Compute Element (CE) somewhere on the Grid. This CE manages a number of Worker Nodes (WN's) on which jobs will actually run. The job will run on a WN on your behalf, therefore you should delegate your credentials to the WMS. You only have to do this once per session. As your proxy certificate expires and you create a new one, you should also delegate it again to the WMS. Typically you do this once a day.

We assume that you have used the **voms-proxy-init** command and have a valid proxy credential. If not, please refer back to the previous chapter.

When you have created a valid proxy certificate, you can delegate it to the WMS. This is done by using the following command.

```
glite-wms-job-delegate-proxy -d $USER
```

The argument to the **-d** option should be a string, in this case it's `$USER`. This string is needed in later commands to identify your session and is called the delegation ID. You can use any string you like after the **-d** option. We use `$USER` in our examples.

Let's create a delegation ID using the WMS by creating a delegation identifier using your username. To get the username we take the `$USER` environment variable. Remember that you can use any string you like as a delegation identifier.

```
$ echo $USER
tutor

$ glite-wms-job-delegate-proxy -d $USER

Connecting to the service https://wms.grid.sara.nl:7443/glite_wms_wmproxy_server


================== glite-wms-job-delegate-proxy Success ==================

Your proxy has been successfully delegated to the WMProxy:
https://wms.grid.sara.nl:7443/glite_wms_wmproxy_server

with the delegation identifier: student101
```

Now the credentials of the user **student101** are delegated to the WMS and jobs can be submitted on behalf of this user.

Instead of creating a delegation ID with **-d**, the **-a** option can be used. This causes a delegated proxy to be established automatically. In this case you do not need to remember a delegation identifier. However, repeated use of this option is not recommended, since it delegates a new proxy each time the commands are issued. Delegation is a time-consuming operation, so it's better to use **glite-wms-job-delegate-proxy** and reuse the delegation ID when submitting your jobs. The **-a** option is useful in situations where proxy certificates are prolonged beyond their expiration time. We will not cover such cases in this tutorial.

2.2. A Simple JDL Job

To submit a job to the Workload Management System (WMS), a text file is needed in which the job is described. For this purpose a special language, Job Description Language (JDL) is used. The JDL describes the job and its requirements in simple attribute value pairs.

Here is an example of the contents of a JDL file:

```
Type = "Job";
JobType = "Normal";
Executable = "/bin/hostname";
Arguments = "-f";
StdOutput = "hostname.out";
StdError = "hostname.err";
OutputSandbox = {"hostname.err","hostname.out"};
ShallowRetryCount = 3;
```

When this file is submitted to the WMS, a job will be created and sent to a Worker Node in the Grid by the WMS. There it will execute the command `/bin/hostname -f`. It will write its standard output in the file *hostname.out* and its standard error in the file *hostname.err*, as specified in the JDL file. This will all take place on the remote Worker Node. In order to get results back an **OutputSandbox** is used.

The **Executable** attribute specifies the command to be run on the Worker Node. The **OutputSandbox** attribute indicates the files to be copied back after job execution; normally these are files where output and error streams are redirected to; their names are determined by the **StdOutput** and **StdError** attributes respectively. Also the number of retries is specified. It is the number of times the WMS will try to run the job.

2.3. Job Submission

A simple job can be submitted by the command:

```
glite-wms-job-submit -d <delegationId> -o <jobidfile> <jdlname>
```

The **-d** option uses the delegationID. This is the string which corresponds to your previous credential delegation. In our case it will be $USER.

As we submit a job, an identifier will be returned. It is needed to get more information about the job and to retrieve output when it is ready. Once you lose this identifier your job is pretty much lost.

The **-o** option is used to save the identifier of the job.

When you created the hostname.jdl file on the server, you can submit this job as follows:

```
$ glite-wms-job-submit -d $USER -o myjobs hostname.jdl

Connecting to the service https://wms.grid.sara.nl:7443/glite_wms_wmproxy_server


====================== glite-wms-job-submit Success ======================

The job has been successfully submitted to the WMProxy
Your job identifier is:

https://wms.grid.sara.nl:9000/T1JVMOHOgl3xCIv_cN4bVg

The job identifier has been saved in the following file:
/home/student101/myjobs

==========================================================================
```

The file *myjobs* contains the jobID(s) returned by the submission process, as a result of the **-o** option. Here the string

https://wms.grid.sara.nl:9000/T1JVMOHOgl3xCIv_cN4bVg

is the Job Identifier. This URL can also be used in a web browser. A prerequisite is that your Grid certificate is loaded in your web browser[2].

[2]For more information on loading your certificate in a browser see Appendix A.

If another job is submitted using the same **-o** value, its jobID is appended to the same jobID file. Try it yourself.

NOTE: Omitting the **-o** option means that the jobID is not saved in a file. It is returned in the output exactly as in the above example. Note that you need the jobID to get your status and retrieve your output. When you do not save this identifier you will effectively lose the output of your job!

In order to know the status of your submitted jobs use

```
glite-wms-job-status <JobID>
```

This command queries the Logging and Bookkeeping service (LB) for the status of the job whose jobID is used as an argument.

If you have saved your jobIds into a file you can use the **-i** option and use the filename as an argument. This is exemplified below with a file called *myjobs*. This file contains two job identifiers. A menu will pop up asking for the user's input. Here you can choose of which jobs you want to see the status. In the example below, the status of all listed jobs is chosen by typing **a**.

```
student101$ glite-wms-job-status -i myjobs

------------------------------------------------------------------
1 : https://wms.grid.sara.nl:9000/8Ey04s3x9xIEE_6hCwb8hw
2 : https://wms.grid.sara.nl:9000/L4kprOVqd9QwdSTg2cbGCw
a : all
q : quit
------------------------------------------------------------------


Choose one or more jobId(s) in the list - [1-2]all:a


*************************************************************
BOOKKEEPING INFORMATION:

Status info for the Job : https://wms.grid.sara.nl:9000/8Ey04s3x9xIEE_6hCwb8hw
Current Status:     Scheduled
Status Reason:      Job successfully submitted to Globus
Destination:        ce.gina.sara.nl:2119/jobmanager-pbs-short
Submitted:          Wed Mar 12 13:23:43 2008 CET
*************************************************************


*************************************************************
BOOKKEEPING INFORMATION:

Status info for the Job : https://wms.grid.sara.nl:9000/L4kprOVqd9QwdSTg2cbGCw
Current Status:     Scheduled
Status Reason:      Job successfully submitted to Globus
Destination:        ce.gina.sara.nl:2119/jobmanager-pbs-short
Submitted:          Wed Mar 12 13:23:49 2008 CET
*************************************************************
```

Note that this command doesn't require a delegation identifier. The status of the two jobs shown above is **Scheduled**.

Note: In order to know the status of your jobs you have to use this polling mechanism of actively querying for the status. Please do NOT do this every second, but wait at least several minutes before trying again.

When your job has been succesfully completed, the result can be retrieved by the command:

```
glite-wms-job-output <JobID>
```

example: **glite-wms-job-output** *https://wms.grid.sara.nl:9000/1QubWarBs8gX86X_QkL*

Again, you don't need to specify a delegation identifier.

```
student101$ glite-wms-job-output -i myjobs
------------------------------------------------------------------
1 : https://wms.grid.sara.nl:9000/DMx2JLLbkV6DofzRLG_sow
2 : https://wms.grid.sara.nl:9000/TuubaTab7s6ehol9OTcO1g
a : all
q : quit
------------------------------------------------------------------

Choose one or more jobId(s) in the list - [1-2]all (use , as separator or - for a range): 1

Connecting to the service https://wms.grid.sara.nl:7443/glite_wms_wmproxy_server


================================================================================

                        JOB GET OUTPUT OUTCOME

Output sandbox files for the job:
https://wms.grid.sara.nl:9000/DMx2JLLbkV6DofzRLG_sow
have been successfully retrieved and stored in the directory:
/tmp/glite/glite-ui/student101_DMx2JLLbkV6DofzRLG_sow

================================================================================
```

In order to inspect the job output, list the files in the indicated directory and show the content of the output file(s).

```
student101$ cd /tmp/glite/glite-ui/student101_DMx2JLLbkV6DofzRLG_sow
student101$ ls -l
total 4
-rw-r--r--    1 student101 users           0 Sep 27 21:25 hostname.err
-rw-r--r--    1 student101 users          19 Sep 27 21:25 hostname.out
student101$ cat hostname.out
wn1-ams.grid.sara.nl
```

The output directory can be choosen by the user by the **–dir** option:

```
student101$ glite-wms-job-output --dir ./op2 -i myjobs
------------------------------------------------------------------
1 : https://wms.grid.sara.nl:9000/DMx2JLLbkV6DofzRLG_sow
2 : https://wms.grid.sara.nl:9000/TuubaTab7s6ehol9OTcO1g
a : all
q : quit
------------------------------------------------------------------

Choose one or more jobId(s) in the list - [1-2]all (use , as separator or - for a range): 2

Connecting to the service https://wms.grid.sara.nl:7443/glite_wms_wmproxy_server


================================================================================

                        JOB GET OUTPUT OUTCOME

Output sandbox files for the job:
https://wms.grid.sara.nl:9000/TuubaTab7s6ehol9OTcO1g
have been successfully retrieved and stored in the directory:
/home/student101/op2

================================================================================
```

2.4. Exercise JS-1: “Hello World”

In this example you will run a simple “Hello World” job on the Grid. The job is described in the Job Description Language (JDL) in the *HelloWorld.jdl* file, which is in the *JSexersise1* directory [3]:

[3] To obtain the exercises and directories see 1.3.

```
[JSexercise1]$ cat HelloWorld.jdl
Executable = "/bin/echo";
Arguments = "Hello World";
Stdoutput = "message.txt";
StdError = "stderror";
OutputSandbox = {"message.txt","stderror"};
```

The job description consists of a number of attribute value pairs. The first line mentions the attribute **Executable** and contains as value the Unix **/bin/echo** command. This is the command that will be run as soon as the job lands on a worker node. If this worker node is unable to execute **/bin/echo** the job fails. As arguments to this command we supply the string **"Hello World"**. Standard output will be written to the *message.txt* as indicated by the fourth line. Standard error will be written to the file *stderror*.

The last line mentions the **OutputSandbox** attribute. This contains the list of files that will be migrated back to the user when the execution has finished. This is only appropriate for small files.

2.4.1. EXERCISE

1. Change to the *JSExercise1* directory.
2. Delegate your credentials by using the command: **glite-wms-job-delegate-proxy -d $USER**
3. Run the job on the Grid using the **glite-wms-job-submit** command.
4. Read and try to understand the output on the screen.
5. Request the status of the job using the **glite-wms-job-status** command.
6. Get the output from this job using the **glite-wms-job-output** command when the Current Status is in the Succeeded state.
7. Check that the job has run correctly by looking into the *message.txt* and *stderror* files.

2.4.2. THE JOB DESCRIPTION LANGUAGE

As you've seen, in gLite job description files (*.jdl* files) are used to describe jobs for execution on the Grid. These files are written using a Job Description Language (JDL). The JDL adopted within the EGEE Grid is the *Classified Advertisement (ClassAd) language* defined by the *Condor Project*, which deals with the management of distributed computing environments, and whose central construct is the *ClassAd*, a record-like structure composed of a finite number of distinct attribute names mapped to expressions. A ClassAd is a highly flexible and extensible data model that can be used to represent arbitrary services and constraints on their allocation. The JDL is used in gLite to specify the desired job characteristics and constraints, which are used by the match-making process to select the resources that the job can use.

The fundamentals of the JDL are given in this subsection. A detailed description of the JDL syntax is outside the scope of this handout. The JDL syntax consists of statements like:

```
attribute = value;
```

Note: The JDL is sensitive to blank characters and tabs. NO blank characters or tabs should follow the semicolon at the end of a line.

In a job description file, some attributes are mandatory, while some others are optional. Essentially, one must at least specify the name of the executable, the files where to write the standard output, and the standard error of the job (they can even be the same file). For example:

```
Executable = "test.sh";
StdOutput = "std.out";
StdError = "std.err";
```

Lines starting with # are not interpreted and can be used to remove certain options temporarily.

If needed, arguments can be passed to the executable:

```
Arguments = "hello 10";
```

Files to be transferred between the UI and the WN before (Input Sandbox) and after (Output Sandbox) the job execution can be specified:

```
InputSandbox = {"test.sh","std.in"};
OutputSandbox = {"std.out","std.err"};
```

Wildcards are allowed only in the **InputSandbox** attribute. The list of files in the Input Sandbox is specified relatively to the current working directory. Absolute paths cannot be specified in the **OutputSandbox** attribute. Neither the Input Sandbox nor the Output Sandbox lists can contain two files with the same name (even if in different paths) as when transferred they would overwrite each other.

Note: The executable flag is not preserved for the files included in the Input Sandbox when transferred to the WN. Therefore, for any file needing execution permissions a `chmod u+x` operation should be performed by the initial script specified as the **Executable** in the JDL file (the `chmod u+x` operation is done automatically for this script).

The environment of the job can be modified using the **Environment** attribute. For example:

```
Environment = {"CMS_PATH=$HOME/cms",
               "CMS_DB=$CMS_PATH/cmdb"};
```

To express any kind of requirement on the resources where the job can run, there is the **Requirements** attribute. Its value is a Boolean expression that must evaluate to true for a job to run on that specific CE. For that purpose all the attributes in the information system can be used.

To run on a CE using PBS as the LRMS, whose WNs have at least two CPUs and the job can run for more than two hours then in the job description file one could put:

```
Requirements = other.GlueCEInfoLRMSType == "PBS" &&
               other.GlueCEInfoTotalCPUs > 1 &&
               other.GlueCEPolicyMaxCPUTime > 120;
```

The WMS can also be asked to send a job to a particular CE with the following expression:

```
Requirements = other.GlueCEUniqueID ==
               "lxshare0286.cern.ch:2119/jobmanager-pbs-short";
```

If the job must run on a CE where a particular experiment software is installed and this information is published by the CE, something like the following must be written:

```
Requirements = Member("CMSIM-133",
                      other.GlueHostApplicationSoftwareRunTimeEnvironment);
```

Note: The **Member** operator is used to test if its first argument (a scalar value) is a member of its second argument (a list). In this example, the **GlueHostApplicationSoftwareRunTimeEnvironment** attribute is a list.

Note: Requirements on attributes of a CE are written prefixing **other.** to the attribute name in the Information System schema.

It is also possible to use regular expressions when expressing a requirement. Let us suppose for example that the user wants all his jobs to run on CEs in the domain *cern.ch*. This can be achieved putting in the JDL file the following expression:

```
Requirements = RegExp("cern.ch", other.GlueCEUniqueId);
```

The opposite can be required by using:

```
Requirements = (!RegExp("cern.ch", other.GlueCEUniqueId));
```

The choice of the CE where to execute the job, among all the ones satisfying the requirements, is based on the *rank* of the CE; namely, a quantity expressed as a floating-point number. The CE with the highest rank is the one selected.

The user can define the rank with the **Rank** attribute as a function of the CE attributes, like in the following (which is also the default definition):

```
Rank = other.GlueCEStateFreeCPUs;
```

2.4.3. Pool Accounts

Every user is mapped onto a local user account on the various Computing Elements all over the Grid. This mapping depends on the VO the user is a member of. The VO is determined from the voms proxy generated by the user, and verified using the voms server certificate.

For each VO a set of numbered accounts is available on the Grid resources. When a user gets to this machine one of these accounts is leased to the user. This lease is temporary and you may get a different account the next time you use the resource.

2.4.4. More on exercise JS-1: ”Hello World” at a different source

In this exersise you will run the same HelloWorld job but now on a pre-selected site. There exists a command that returns the result from the match making job the WMS does on the basis of the job description in the JDL file. From the list of Computing Elements that could run your job you can select one and use one of the options in the JDL syntax to send your job to that site.

The relevant command for this exercise is:

```
glite-wms-job-list-match -d $USER --vo <vo> <job.jdl>
```

This command provides information about the CE’s on which the job could run.

2.4.5. exercise

1. Find out which option in the JDL syntax you can use to choose your favorite CE to run your job.
2. Find out were your job could possibly run by using the **glite-wms-job-list-match** command.
3. Choose your favourite CE and submit the “HelloWorld.jdl” job to this site using an extra line in the “HelloWorld.jdl” file.
4. Check the status of your job and verify your job was indeed run at the site of your choice.
5. When the data is ready, get your output back and check that the job was executed correctly.

It is possible to see which CEs are eligible to run a job specified by a given JDL file using the command **glite-wms-job-list-match**:

```
$ glite-wms-job-list-match -d $USER --vo alice HelloWorld.jdl

Selected Virtual Organisation name (from proxy certificate extension): alice
Connecting to host boswachter.nikhef.nl, port 7772

**************************************************************************
                         COMPUTING ELEMENT IDs LIST
 The following CE(s) matching your job requirements have been found:

                   *CEId*
 farm012.hep.phy.cam.ac.uk:2119/jobmanager-lcgpbs-Lq
 farm012.hep.phy.cam.ac.uk:2119/jobmanager-lcgpbs-Mq
 farm012.hep.phy.cam.ac.uk:2119/jobmanager-lcgpbs-Sq
 ...
 zeus02.cyf-kr.edu.pl:2119/jobmanager-lcgpbs-long
 zeus02.cyf-kr.edu.pl:2119/jobmanager-lcgpbs-short
 tbn18.nikhef.nl:2119/jobmanager-pbs-qlong
**************************************************************************
```

When specifying something like:

```
Requirements = RegExp("nikhef.nl", other.GlueCEUniqueId);
```

in the JDL file a selection of sites can be made.

In addition, you may also want to know which CE's are available to your VO and if they are busy running jobs at the moment. This can be done by using the `lcg-infosites` command. You can type the following to know more about its use:

```
lcg-infosites --help
```

As an example consider the following use of `lcg-infosites`. It queries for all the CE's for the VO `lsgrid`. The output shows all CE's and their submission queues, together with current use statistics.

```
$ lcg-infosites --vo lsgrid ce
#CPU    Free    Total Jobs      Running Waiting ComputingElement
----------------------------------------------------------
1152    1141       6               6       0    trekker.nikhef.nl:2119/jobmanager-pbs-qlong
1152    1141       2               2       0    trekker.nikhef.nl:2119/jobmanager-pbs-qshort
 124     124       0               0       0    ce.grid.rug.nl:2119/jobmanager-pbs-short
 808     790      17              17       0    ce.gina.sara.nl:2119/jobmanager-pbs-medium
 808     790       1               1       0    ce.gina.sara.nl:2119/jobmanager-pbs-short
   3       3       0               0       0    gb-ce-ams.els.sara.nl:2119/jobmanager-pbs-medium
  17      17       0               0       0    gb-ce-lumc.lumc.nl:2119/jobmanager-pbs-medium
  13       1     133              12     121    gb-ce-wur.els.sara.nl:2119/jobmanager-pbs-medium
  17       1     145              16     129    gb-ce-kun.els.sara.nl:2119/jobmanager-pbs-medium
  17       1     121              16     105    gb-ce-uu.science.uu.nl:2119/jobmanager-pbs-medium
 124     124       0               0       0    ce.grid.rug.nl:2119/jobmanager-pbs-long
1152    1142       2               2       0    gazon.nikhef.nl:2119/jobmanager-pbs-qshort
1152    1142       6               6       0    gazon.nikhef.nl:2119/jobmanager-pbs-qlong
 124     124       0               0       0    ce.grid.rug.nl:2119/jobmanager-pbs-medium
  17       1     112              16      96    gb-ce-nki.els.sara.nl:2119/jobmanager-pbs-medium
  17       1     105              16      89    gb-ce-amc.amc.nl:2119/jobmanager-pbs-medium
```

A similar use of the same command will come up in the chapter about data management.

2.5. EXERCISE JS-2: PING A HOST FROM A NODE; THE SUBMISSION OF SHELL SCRIPTS TO THE GRID

Now we will ping a host from a Worker Node to exercise again the execution of simple operating system commands on the nodes. In this particular case we will execute the **ping** command in two ways: directly calling the **/bin/ping** executable on the node and by executing a simple shell script (*pinger.sh*) which does the same thing. This will teach us how to use shell scripts on the Grid.

2.5.1. EXERCISE

1. Go to the directory *JSexercise2*.
2. Execute the **ping** command on host *www.sara.nl* from a shell on the User Interface machine to see what it does. Depending on the installation ping ends by itself or has to be stopped with Ctrl-c (If you want to find out more about the ping command type `man ping`).
3. Have a look at the *pinger1.jdl* file and try to understand what it does.
4. Submit the "pinger1" job on the Grid and retrieve the output when the job has finished and see if the output is what you expected.
5. Now have a look at the *pinger2.jdl* file and notice the differences.
6. Submit the "pinger2" job on the Grid and retrieve the output when the job has finished and verify if the output is the same.

As you may have noticed, the executable in the second job was the bash shell itself. The parameters for this executable are then the **ping** command and the **hostname**. In this case one uses a Unix fork and executes the command in the new shell. This may be usefull in case you know your script only works within a specific shell. Without a fork this should also work but then you have to be sure your script can run in the default shell of the worker node. In that case the executable becomes the *pinger.sh* script and the argument to be passed to the executable is just the **hostname**.

In the first case we directly call the ping executable (the JDL file is *pinger1.jdl*):

```
Executable    = "/bin/ping";
Arguments     = "-c 5 www.sara.nl";
RetryCount    = 7;
Stdoutput     = "pingmessage1.txt";
StdError      = "stderror";
OutputSandbox = {"pingmessage1.txt","stderror"};
Requirements  = other.GlueHostOperatingSystemName == "Scientific Linux";
```

Whereas in the second case we call the **bash** executable to run a shell script, giving as input argument both the name of the shell script and the name of the host to be pinged, as required by the shell script itself (the JDL file is *pinger2.jdl*):

```
Executable    = "/bin/bash";
Arguments     = "pinger.sh www.sara.nl";
RetryCount    = 7;
Stdoutput     = "pingmessage2.txt";
StdError      = "stderror";
InputSandbox  = "pinger.sh";
OutputSandbox = {"pingmessage2.txt","stderror"};
Requirements  = other.GlueHostOperatingSystemName == "CentOS";
```

where the *pinger.sh* shell script, to be executed in bash, is the following one:

```
#!/bin/sh
/bin/ping -c 5 $1
```

Note the use of a requirement on an operating system on the nodes. We also make use of the **RetryCount** mechanism, which will cause the job to be resubmitted when it for some reason fails on a certain site. Note that this will not resubmit your job when there is a bug in your job script.

2.5.2. EXERCISE

1. Make your own *pinger3.jdl* file where you make *pinger.sh* the executable.
2. Run this new job on the Grid and verify the output
3. Make you own *student.sh* script in which you don't ping a host but executes some other commands like for example **/bin/pwd** or **/usr/bin/who**.
4. Submit this script to the Grid make sure that the output you get back is what you expected.

2.6. EXERCISE JS-3: RENDERING IMAGES

An useful application of the Grid is rendering images, for example for scientific visualisation. In this example we will make use of a raytracer program called POV-Ray to create the images.

2.6.1. EXERCISE

1. Go to the directory *JSexercise3*.
2. Look at the file *povray_morphine.jdl* and make sure you understand what it does. Note that the job execution itself is done in a shell script *start_povray_morphine.sh* and that the POV-Ray software is required to be present on the node.
3. Look at the *start_povray_morphine.sh* script and see what it does.
4. Look for yourself which CEs are able to accept this job
5. Submit the job to the Grid and copy the output to the UI after the job has finished
6. View the *morphine.png* file with the **display** tool.

The JDL file (*povray_morphine.jdl*) for this exercise looks like:

```
Executable = "/bin/sh";
StdOutput = "povray_morphine.out";
StdError = "povray_morphine.err";
InputSandbox = {"start_povray_morphine.sh","morphine.pov"};
OutputSandbox = {"povray_morphine.out","povray_morphine.err","morphine.png"};
RetryCount = 7;
Arguments = "start_povray_morphine.sh";
Requirements = Member("POVRAY",other.GlueHostApplicationSoftwareRunTimeEnvironment);
```

The executable to be used in this case is the **sh** shell executable. Its input argument is the name of the shell script we want to be executed (*start_povray_morphine.sh*):

```
#!/bin/bash
/usr/bin/povray -D +W800 +H800 morphine.pov
```

We can finally, after having retrieved the output of the job, examine the produced image using for example **display**. If you do not see any out put make sure you have connected to the UI using **ssh -X**, alternatively the `$DISPLAY` variable should be set to your current terminal).

Since we require a special software package called POV-Ray, we need to specify this as a requirement in our JDL file. The availability of POV-Ray is published in the Grid Run Time environment by a tag called "POVRAY". We can specify this tag in the Requirements classAd, in order to let the WMS only consider only those CEs which have this software installed.

2.7. Exercise JS-4: Checksum on a large InputSandbox transferred file

In this exercise you will transfer via the **InputSandbox** a file whose checksum is known to a worker node. On the worker node you will check that the file was transferred correctly by performing a checksum again. You will use the shell script *ChecksumShort.sh*, which exports in an environment variable (`$CSTRUE`) the value of the checksum of the file before the transfer. In the script the checksum is performed again on the worker node by issueing the **cksum** command on the file and the result is stored in the `$CSTEST` variable. When `$CSTEST` is equal to `$CSTRUE` the file was not corrupted during the transfer.

2.7.1. Exercise

1. Go to the directory *JSexercise4* and perform the checksum locally on the file *short.dat*, which is present in that directory. Make sure you understand the **cksum** command. More information about this command you get by typing `cksum --help`.
2. Look at the *ChecksumShort.jdl* file and note that no arguments need to be passed with this job and that only the shell script *ChecksumShort.sh* will be executed.
3. Look at the *ChecksumShort.sh* file and note that indeed all parameters needed to run the job are specified in here. Try to understand what this script does and try to predict what you will see in the output file after the job has finished.
4. Submit the job to the Grid, check its status and retrieve the output and verify that the answer is what you expected.
5. Manually change the value for the checksum. Submit the job again and verify you understand the result.

The JDL file (*ChecksumShort.jdl*) is the following one:

```
Executable     = "ChecksumShort.sh";
StdOutput      = "std.out";
StdError       = "std.err";
InputSandbox   = {"ChecksumShort.sh", "short.dat"};
OutputSandbox  = {"std.out", "std.err"};
```

If everything worked fine, and the GridFTP InputSandbox transfer was OK, the *std.out* should read:

```
True checksum:'2933094182 1048576 short.dat'
Test checksum:'2933094182 1048576 short.dat'
Done checking.
Goodbye. [OK]
```

2.8. Exercise JS-5: A small cascade of “Hello World” jobs

Very often you do not want to submit just one job but a whole series of jobs with the same executable but with different input files (e.g. when you want to run a reconstruction file on all data files from the month January). In this exercise you will submit a small cascade of “Hello World” jobs by a special type of job, called a parametric job.

A parametric job causes a set of jobs to be generated from one JDL file. This is invaluable in cases where many similar (but not identical) jobs must be run. This is achieved by the parametric job having one or more parametric attributes described in the JDL. These attributes are identified by use of the key word `_PARAM_` in its value; that value will be replaced by the actual value of Parameters during the jdl expansion. The JobType in the JDL is Parametric.

In this exercise a set of values of the parameters is defined by the three attributes **Parameters**, **ParameterStep** and **ParameterStart**. **ParameterStart** defines the starting value for the variation; **Parameter-**

Step the step for each variation and **Parameters** defines the value where the submission of jobs will stop (that value itself is not used). The number of jobs that will be submitted **(Parameters - ParameterStart) / ParameterStep**. An example will clarify this.

```
[student101/exercises]$ cat parametric.jdl
 [
    JobType = "Parametric";
    Executable = "/bin/echo";
    Arguments = "Hello World";
    Parameters= 6;
    ParameterStep =2;
    ParameterStart = 0;
    StdOutput = "myoutput_PARAM_.txt";
    StdError = "myerror_PARAM_.txt";
    OutputSandbox  = {"myoutput_PARAM_.txt", "myerror_PARAM_.txt"};
    ShallowRetryCount = 1;
]
```

Here `_PARAM_` acts very much like a variable. It will be expanded starting with 0. Then the value is increased until it is equal or smaller than 6. Note that 6 itself is not reached. In this case, three jobs will be generated. The standard output of these files will be written to the file *myoutput0.txt*, *myoutput2.txt* and *myoutput4.txt*. Likewise the matching standard error will be written to the files *myerror0.txt*, *myerror2.txt*, *myerror4.txt*.

2.8.1. EXERCISE

1. Go to the directory *JSexercise5* and submit the *parametric.jdl* file. Use the **-o** option to save the job identifiers to a file of your choice.
2. When all jobs are finished retrieve the results using the **-i** option of the **glite-wms-job-output command.**
3. Check the different output files.

You can experiment with other values for the attributes Parameters, ParameterStep and ParameterStart. You can also use the `_PARAM_` keyword in the names of file that go in the **InputSandbox**, like in the following example:

```
[
    JobType = "Parametric";
    Executable = "/bin/sh";
    Arguments = "message_PARAM_.sh";
    InputSandbox = "message_PARAM_.sh";
    Parameters= 6;
    ParameterStep =2;
    ParameterStart = 0;
    StdOutput = "myoutput_PARAM_.txt";
    StdError = "myerror_PARAM_.txt";
    OutputSandbox  = {"myoutput_PARAM_.txt", "myerror_PARAM_.txt"};
    ShallowRetryCount = 1;
]
```

In this case you have to provide three input files to be submitted with the job: *message0.sh*, *message2.sh* and *message4.sh*. In this way you can easily perform computation of the same procedure with different input parameters.

3. DATA MANAGEMENT

3.1. INTRODUCTION

In a Grid environment, data files can be replicated, possibly on a temporary basis, to many different sites depending on where the data is needed. The users or applications do not need to know where the data is physically located: they use logical names for the files. Data Management services are responsible for locating and accessing the data. Data on the Grid is stored on so called *Storage Elements (SEs)*. The data on a Storage Element is stored per VO, and only users of the same VO have access to the data. In order to optimise data access and to introduce fault-tolerance and redundancy, data files can also be replicated to multiple SEs. To be able to easily find the stored data the LCG File Catalog (LFC) is used to keep track of all the data. Access to the LFC is controlled by your certificate, and therefore you need to be registered with a VO.

The files in the Grid are referenced by different names:

- *Grid Unique IDentifier (GUID)*; a file can always be identified by its GUID, which is assigned at data registration time and is based on the *Universal Unique IDentifier (UUID)* standard to guarantee unique IDs. A GUID is of the form:
  guid:<unique_string>
  and all the replicas of a file will share the same GUID. An example GUID is:
  `guid:3cb13190-ab23-11d8-bc9c-d39c21caf9ab`

- *Logical File Name (LFN)*; in order to locate a Grid accessible file, the human user will normally use a LFN. LFNs are usually more intuitive, human-readable strings, since they are chosen by the user as GUID aliases. LFNs are organised in a directory structure within the LFC. Special lfc commands are available to see the LFNs. Their form is:
  lfn:<any_alias>
  An example LFN is:
  `lfn:importantResults/Test1240.dat`

- *Storage URL (SURL)*; the SURL is used by the LFC to find where a replica is physically stored, and by the SE to locate it. The SURL is of the form:
  srm:// <SE_hostname><SE_Accesspoint><VO_path><filename>
  An example SURL is:
  `srm://tbed0101.cern.ch/flatfiles/SE00/dteam/generated/2004-02-26/`
  `file3596e86f-c402-11d7-a6b0-f53ee5a37e1d`

While the GUID or LFN refer to files and not replicas, and say nothing about locations, the SURLs give information about where a physical replica is located. Figure 1 shows the relation between the different file names.

As a tool to access the LFC and to store and retrieve data form the Grid the LCG Replica Management tools have been written. These tools assist in storing and retrieving data, and also in creating and deleting replicas. There is also the possibility to store some metadata in the LFC, although special databases for metadata exist as well.

3.1.1. GLITE DATA MANAGEMENT TOOLS

In this chapter, exercises will be presented to make you familiar with the gLite Data Management tools. These are high level tools used to upload files to the Grid, replicate data and locate the best replica available.

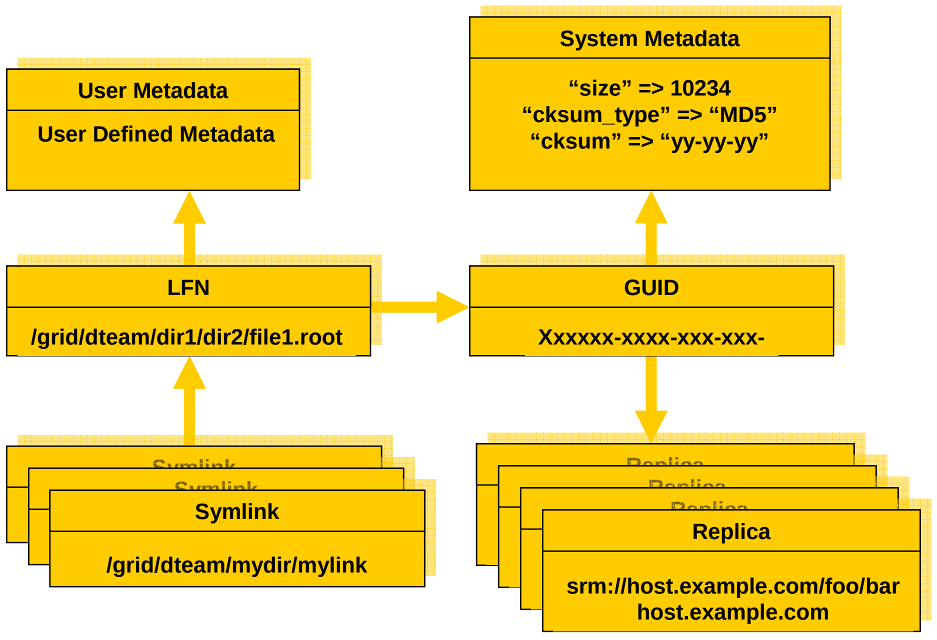


Figure 1: The mapping between GUID, LFNs and physical file names is maintained in the LCG File Catalog.

3.2. Exercise DM-1: Discover Grid storage

In general, all Storage Elements registered with the Grid publish information about themselves through the Grid information system. This information contains the VOs they support, the location of the VO storage directory, the amount of space available, etc. In order to get information about Storage Elements the **lcg-infosites** command can be used. This command can query the information system about several things.

To retrieve information about SEs we can issue:

```
$ lcg-infosites --vo tutor se
```

The output presents information about all Storage Elements available to the VO.

3.2.1. Exercise

1. Issue the command to retrieve information about Storage Elements.
2. Read and try to understand the output on the screen.
3. Find out how many Storage Elements support the tutorial VO "tutor" (i.e. how many SEs you can use).

3.3. Exercise DM-2: Looking in the LCG File Catalog

The LCG File Catalog (LFC) stores the Logical Filenames (LFNs) in a directory structure. It is therefore possible to do an **ls** on the files in the LFC, to see what files are available. Another command that is useful is the **mkdir** command to create a new directory in the LFC. When you are storing LFNs in the LFC it is best to keep your files separated from files from other people. It is therefore wise to create your own subdirectories where you store your LFNs. Note that subdirectories have to be created in advance, before storing LFNs in them. So before you register a Grid file (see next section), you have to create the directory first.

A file can have multiple LFNs. Extra LFNs are created as symbolic links in the LFC. Each file has one main entry, and can have multiple symbolic links pointing to that file.

In order to access the LFC directly, an environment variable **LFC_HOST** has to be set. The best way to set it is to make use of lcg-infosites to obtain the name of the LFC server.

```
$ export LFC_HOST=`lcg-infosites --vo tutor lfc`
```

Note the special quotes around the command.

After **LFC_HOST** has been set, one can issue the commands **lfc-ls**, **lfc-ln**, and **lfc-ln** (amongst others) to study and work with the file catalog.

3.3.1. EXERCISE

1. Set the **LFC_HOST** environment variable to the correct value.
2. Do an **lfc-ls** on the file catalog, note that the toplevel directory for tutor is /grid/tutor. So the command is `lfc-ls /grid/tutor`.
3. Use **lfc-mkdir** to create one or more subdirectories to use during the rest of the exercises.

3.4. EXERCISE DM-3: FILE REPLICATION WITH THE REPLICA MANAGER

In this exercise we will use the Replica Manager for replicating files between various SEs and get familiar with the basic catalogue commands to list and delete replicas. The following commands can be used:

- **lcg-cr** copy and register a file
- **lcg-lr** lists the replicas for a given LFN, GUID or SURL
- **lcg-lg** lists the GUID for a given LFN or SURL
- **lcg-rep** copy a file from one SE to another SE and registers it in the LFC
- **lcg-aa** add an alias in the LFC for a given GUID
- **lcg-ra** remove an alias in the LFC for a given GUID
- **lcg-rf** register a file residing on an SE in the LFC
- **lcg-uf** unregister a file residing on an SE in the LFC
- **lcg-cp** copy a Grid file to a local destination
- **lcg-la** lists the aliases for a given LFN, GUID or SURL

Extra information concerning each command can be obtained by inspecting the man-pages. Information about the **lcg-cr** command is shown with `man lcg-cr`, for example.

3.4.1. copyAndRegisterFile; Uploading a file from the UI to the Grid

In order to upload a file to the Grid, i.e., to transfer it from the local machine to a Storage Element where it must reside permanently, the **copyAndRegisterFile** (**lcg-cr**) command can be used (on a system with a valid proxy):

```
$ lcg-cr --vo tutor -l lfn:/grid/tutor/exercises/testfile.dat -d srm.grid.sara.nl \
 file://$(pwd)/testfile.dat
guid:11c00016-cec0-4530-be19-ff42644da0b0
```

where the only argument is the local file to be uploaded (a fully qualified URL) and the **-l** option indicates an LFN for it. The command returns the unique GUID for the file. If no LFN is provided, the file will not be registered in the catalog with a logical file name. With this method you should always provide an LFN, and your ability to retrieve the file now fully depends on the LFN.

The **-d** <**destination**> option selects the specified SE as the destination for the file. There the file will be stored and given a ”random” name. This file name is not known to the user. You need the LFN in order to retrieve the file.

A better way of storing a file on the Grid is to use a ”destination path” instead of just the SE name after the **-d** flag. Instead of only the SE hostname a complete SURL, including the SE hostname, the path (accesspoint plus VO-specific directory) and a chosen filename, can be used as the destination. This is illustrated by the following commands:

```
 $ lcg-cr -d srm://srm.grid.sara.nl/flatfiles/SE00/tutor/testfile.dat \
 > --vo tutor -l lfn:/grid/tutor/exercises/testfile file://$(pwd)/testfile.dat
```

This allows the user to retrieve the file without using the LFN and use the SURL instead. Notice that in this way the LFC is effectively bypassed. Users that keep precious data on the Grid should not depend fully on the LFC and keep a list of SURLs of their valuable files.

Finally, in this and other commands the **-n** <**#streams**> option can be used to specify the the number of parallel streams to be used in the transfer. This option can speed up data transfers over long distances.

3.4.2. listReplicas & listGUID; Listing replicas and GUIDs

The Replica Manager allows users to list all the replicas of a file that have been successfully registered with the Replica Location Service. For that purpose the **listReplicas** (**lcg-lr**) command is used:

```
$ lcg-lr --vo tutor lfn:/grid/tutor/exercises/testfile.dat
srm://srm.grid.sara.nl/flatfiles/SE00/tutor/generated/2005-05-17/filee4028505-9f4a-436b-84fe-f2691dbddac8
```

Note: Instead of LFN the GUID or SURL can be used to specify the file for which all replicas must be listed. The SURLs of the replicas are returned.

Reciprocally, the **listGUID** (**lcg-lg**) return the GUID associated with a specified LFN or SURL:

```
$ lcg-lg --vo tutor lfn:/grid/tutor/exercises/testfile.dat
guid:11c00016-cec0-4530-be19-ff42644da0b0
```

3.4.3. replicateFile; Replicating a file

Once a file is stored on an SE and registered with the Replica Location Service, the file can be replicated using the **replicateFile** (**lcg-rep**) command, as in:

```
$ lcg-rep -d se.grid.rug.nl --vo tutor lfn:/grid/tutor/exercises/testfile.dat
```

where the file to be replicated can be specified using a LFN, GUID or even a particular SURL, and the **-d** option is used to specify the SE where the new replica will be stored (and, as with **CopyAndRegisterFile**, using either the SE hostname or a complete SURL). If this option is not set, then the an SE is chosen automatically.

Note: For one GUID, there can be only one replica per SE. If the user tries to use the **replicateFile** command with a destination SE that already holds a replica, the existing SURL will be returned, and no new replica will be created.

3.4.4. COPYFILE; COPYING FILES OUT OF THE GRID

The **copyFile** (**cp**) command can be used to copy a Grid file to a non-Grid storage resource. This is useful to have a local copy of the file. The command accepts the LFN, GUID or SURL of the file as its first argument and a local filename as the second, as is shown in the following example:

```
$ lcg-cp --vo tutor lfn:/grid/tutor/exercises/testfile file://$(pwd)/testfile.dat
```

Note: Although this command is designed to copy files from a SE to a non-Grid resources, if the proper URL is used, a file could be transferred from one SE to another, or from out of the Grid to an SE. *This should not be done*, since it has the same effect as using **replicateFile** but *skipping the file registration*, making this replica invisible to Grid users.

3.4.5. DELETEFILE; DELETING REPLICAS

Once a file is stored on a Storage Element and registered with a catalogue, it can be deleted using the **deleteFile** (**del**) command. If a SURL is provided as argument, then that particular replica will be deleted. If a LFN is given instead, then the **-s** <**SE**> option must be used to indicate which one of the replicas must be erased. The same is true if a GUID is specified, unless the **-a** option is used, in which case all replicas of the file will be deleted and unregistered (on a best-effort basis).

The following commands:

```
$ lcg-del -s srm.grid.sara.nl --vo tutor lfn:/grid/tutor/exercises/testfile.dat
```

and

```
$ lcg-del -a --vo tutor lfn:/grid/tutor/exercises/testfile.dat
```

remove, from the file system and the catalog, one particular replica and all available replicas of the file, respectively.

3.4.6. EXERCISE

1. Create a file using e.g. the **touch** or **echo** commands. You can also use *testfile.dat* in the directory *DMexercise3*
2. Find an SE which you want to use to copy your file to.
3. Copy the created file to the SE and register it in the replica catalogue with a Logical File Name.
4. Check if the copy was successful and that the file is registered.
5. Create a replica of the file at a different SE.
6. Repeat this until you get bored with it...
7. Inspect all created replicas.
8. Copy the file stored on Grid Storage back to you local account.
9. Cleanup and unregister all created replicas.

3.5. EXERCISE DM-4: ACCESSING A GRID FILE FROM A JOB

A job that is submitted to the Grid can, of course, access files stored on SEs. The best way to do that is by making use of the lcg-cp command to copy the data to the WN the job runs on. In some cases it is better to have the job run close to the data that it needs to access. For that purpose, the JDL file of the job must include the name (GUID or LFN) of the files to be accessed, in the **InputData** attribute; and the protocol that will be used to access them in the **DataAccessProtocol** attribute. Currently, the only two supported protocols to access Grid files are: GridFTP (**gsiftp**) and rfio (**rfio**).

The following exercise show how to access files from a Perl script.

Note: The Logical File Names used in the exercises should be treated as exemplary. Since, the LFNs that can be registered in the replica catalogue are unique, to be able to do these exercises the lfn should be altered accordingly.

3.5.1. EXERCISE

1. Change to the *DMExercise4* directory, and study the *.jdl* and *.pl* files.
2. Copy and register the file *values* to a SE (e.g. *srm.grid.sara.nl.*) with your uniquely chosen LFN.
3. Change the LFNs in the scripts and JDL files to make them reflect your LFN.
4. Run the *gsiftp.jdl* job (Tip: first try to run the Perl scripts locally, this can save you a lot of debugging time).
5. Retrieve, inspect and try to understand the output of the job.

3.5.2. ACCESSING A FILE USING THE GRIDFTP PROTOCOL

We assume that a user has registered a data file (called *values*) within the EGEE Grid, using **lfn:unique_name** as its LFN. The contents of the file are the following:

```
 pi  = 3.141592654
 e   = 2.718281828
 tel = 020-5923000
```

The JDL file of the job (*gsiftp.jdl*) includes the LFN of the file, and the protocol (**gsiftp**) to be used when accessing it. The contents of the JDL file follows:

```
Executable="gsiftp.pl";
StdOutput="std.out";
StdError="std.err";
InputSandbox={"gsiftp.pl"};
OutputSandbox={"std.out","std.err"};
InputData={"lfn:/grid/tutor/mydir/uniquename"};
DataAccessProtocol={"gsiftp"};
```

The executable (**gsiftp.pl**) is a Perl program, that calls the **lcg-cp** command to copy the Grid file to the local filesystem of the Worker Node where the job is running. The rest of the script is simple Perl code to show the data retrieved:

```
#!/usr/bin/perl

# Copy the input data file to the WN local filesystem
system "lcg-cp --vo tutor lfn:/grid/tutor/mydir/uniquename file:///`pwd`/values";

# Open it
open(file,'values');

# Read all the lines\\
@lines=<file>;

#Show the info
print "The values stored in the input data file are:\n";
print " @lines";
```

The job is submitted as usual:

```
$ edg-job-submit -o jobid gsiftp.jdl
```

And the results retrieved with:

```
$ edg-job-get-output -i jobid
```

The *std.out* file obtained is this:

```
The values stored in the input data file are:
 pi  = 3.141592654
 e   = 2.718281828
 tel = 020-5923000
```

3.6. Exercise DM-5: Use Case - Copy and Register Job Output Data

In this exercise you are going to write a job `job1` that produces several output files that are afterwards copied and registered to various SEs. Next you are going to write a second job that reads the files of `job1` and prints the output on the screen. This is a typical use case of centralised data production where the result is distributed to various sites that are part of a particular Virtual Organisation. These production results are later analysed by users distributed all over the globe.

Note: In this exercise we only list a set of steps that will guide you through the use case. We do not provide a detailed list of commands but leave it as a challenge for you to solve the exercise in the best possible way.

The following steps are necessary for implementing this use case:

- Write a simple job `job1` that produces 5 output files with random numbers between 0 and 100.

- Copy and register these files to various SEs at the end of the job.
- Register metadata about file size, owner and description of the files.
- Write a second job `job2` that reads in the files of `job1` and prints the output on the screen. [Hint: This step is similar to Exercise 4.]

A GETTING ACCESS TO THE GRID

A1. GRID CERTIFICATES

While you are using computer systems that are scattered all over the world, the administrators of all those machines will want to know who is using their machines and storage. In the past, you had to contact each site administrator separately, and you would get a username and a password for every new site. By providing this combination, the administrator could be sure who was using the system. But the user was obliged to remember as many passwords as there were sites. This cumbersome way of working is not suitable for the Grid, where you will be accessing many different sites without you even knowing.

On the Grid, you will be using a certificate. This certificate binds together your identity (name, affiliation, etc.) and a unique piece of digital data called a public key. A third party that is trusted by all sites in the EGEE infrastructure digitally signs the combination of your name and the public key.

The use of a public key to authenticate yourself is based on a special mathematical trick, called *asymmetric cryptography*. If you would pick two large prime numbers and multiply them, it is virtually impossible to factorise the product into the two numbers again. The individual prime numbers are used to generate an encryption and a decryption function and the product of the two, and then the two numbers are destroyed. If you only have the encryption function, it is impossible to derive the decryption functions from it (and vice versa). So, if you distribute the encryption function called public key widely (e.g. you put it on the web) but keep the decryption function private (private key), everyone can send you encrypted messages, but only you can read them and even the sender cannot get the message back!

This method is quite useful if you want to authenticate yourself to a remote site without revealing any personal information: if the remote site knows your public key, it can encrypt a challenge (e.g. a random number) using this key and ask you to decrypt it. If you can, you obviously own the private key and therefore you are who you say you are but still the remote site has to know all the public keys of every one of its customers.

It all becomes simpler if we introduce a trusted third party, a human that can authenticate people in person called a *Certification Authority (CA)*. When you go to a CA you bring along your public key and an identifier containing your full name and possibly an affiliation. Now the CA has to make sure by some other means that you are indeed who you claim to be. The CA may ask for a passport or drivers license, it could contact your boss to verify your affiliation, make a phone call to your office, etc. When the CA is reasonably convinced of your identity, it will take your public key and your identifier and put those together in a certificate. As a proof of authentication, the CA will then calculate a digest (hash) of the combination of the two and encrypt it with the private key of the CA. Everyone can recalculate the digest, decrypt the signature using the public key of the CA and verify that these two are the same. If you show up at a remote site that only knows your name (identifier) and trust the CA that you got your certificate from, the site knows that whoever can decrypt the challenge sent, corresponds to the name they have in their list of allowed users.

A2. GETTING A CERTIFICATE

This subsection will try to familiarise you with the procedure of making a certificate request for a certificate that is useful in *real life*. The exact procedure is different for every CA and therefore differs per country. As employee of a Dutch institute you need a *medium-security CA* certificate from the DutchGrid CA, in order to be able to use the Grid. The website for this CA is http://www.dutchgrid.nl/ca. On this page you will find a link to a web form that will help you to generate a certificate request. When you fill in all information and make your way through the certification details, you can in the end download a shell script and an application form. You can run the shell script on for example the user interface machine. The shell script is called *makerequest.sh* by default and is usually written to your home directory or Desktop. You have to download the application form as well, print it out and fill in the missing details. Don't forget the "proof-of-possesion challenge", this information is generated when you run the script.

When you run the shell script (run it only once!), it will generate a new, unique public and private key and mail a certificate request to ca@nikhef.nl. Note that the machine has to be able to send mail for this. Otherwise you have to mail the request yourself, see the CA webpage for more information.

For large CAs, it is very difficult to contact everyone personally. Therefore, the task of authenticating people has been issued to *Registration Authorities (RA)*s. Like a CA, an RA is a real person, maybe the head of your personnel department, or your team leader. The RAs do not sign certificates themselves, but tell a CA that a particular person belongs to a particular certificate request and that they should sign the request. The task of an RA is simple, and many RAs can be appointed for one CA. On the other hand, running a proper CA is a complex task, requiring a secure environment and personnel.

You will find the Registration Authority that you have to go to mentioned on the web page with the script. The RA is also named on the application form. You will have to go to the RA in person. Bring the signed application form and a valid picture-ID (passport, national identity card, or drivers license) with you. The RA will check your identity and sign the application form as well.

You then have to send the application form, together with a photocopy of the picture-ID used to the CA.

After a while, you get a certificate back from the CA by e-mail. You need to store the certificate in a file called *usercert.pem* in the *.globus* directory, where your private key *userkey.pem* should also be stored. Note that the private and public keys should belong together, otherwise you will see all kinds of strange error messages.

It does not matter how much bogus is in the certificate file, as long as you keep the fragment between `BEGIN CERTIFICATE` and `END CERTIFICATE` intact.

A3. Registering in a Virtual Organisation

If you want to make use of the EGEE Grid, you should register with a Virtual Organisation (VO). This may be your high energy physics experiment (LHCb, Babar) or your community (vlemed, lofar, dans, lsgrid, dteam).

When registering you have to agree to the Acceptable Use Policy of the VO. In order to register with a VO you must authenticate yourself with your certificate to a web site, and therefore you need to have your certificate available inside your web browser.

A3.1. Importing certificate in a browser

The files you have on disk are suitable for Grid use, but need to be converted to a different format to be used in web browsers. This format is called `PKCS#12`, and files have the extension *.p12*. This format is special in the sense that a single file contains both your public and your private key, and the combination is again protected with a pass phrase (here called export password).

The **openssl** program can be used to convert between the different formats:

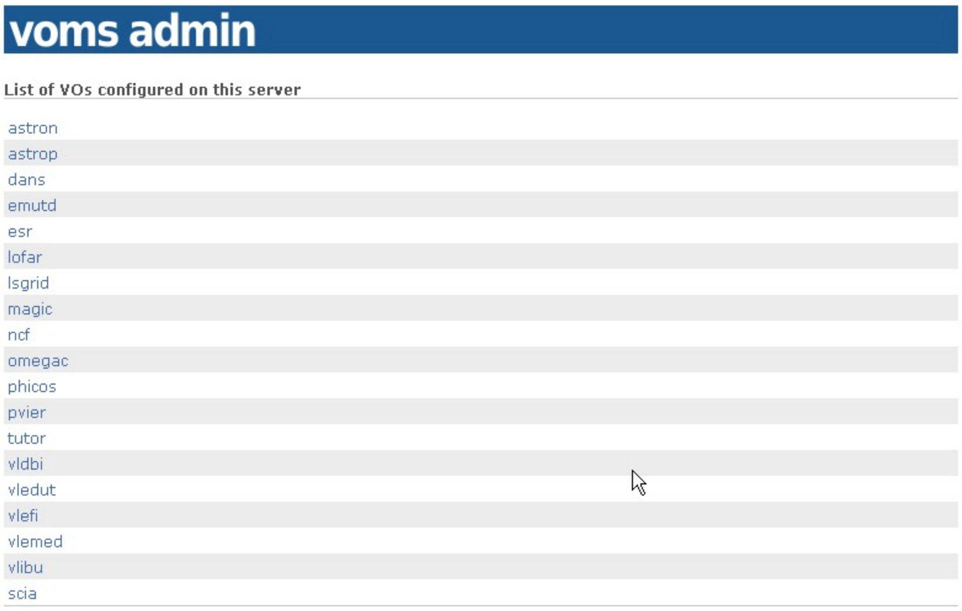
```
$ cd $HOME/.globus
$ openssl pkcs12 -export -in usercert.pem -inkey userkey.pem \
 -out packed-cert.p12
```

The file *packed-cert.p12* now contains both your certificate and your private key, and can be imported in an internet browser. In this tutorial we will use Konqueror (also installed on the UI), but Internet Explorer or Firefox would work as well. The certificate in Konqueror can be imported via: `Settings -> Configure Konqueror` which will open the Settings window. Then, on the left, click `Crypto` to open the cryptography settings page. Under the tab `Your Certificates` you can then import your certificate by pressing the Import button. In addition to importing the certificate, you need to set the 'Default Action' to `Send` under the `Authentication` tab.

Konqueror will protect its certificate store with a password as well. Enter a good password in the dialogue. If the browser does not ask you for a password, you will have to set it manually later. In the file browser window you will subsequently get, go to your *.globus* directory and select the packed-cert.p12 file. Again, you will have to provide a password, this time the export password you gave to **openssl** when you created the `PKCS#12` file. You have now successfully imported your certificate and you can close the Konqueror security window.

A3.2. Requesting access to a VO

You are now ready to apply for a VO membership. For the NL-Grid infrastructure you can register at the following website (see Figure 2):
https://voms.grid.sara.nl:8443/vomses/


voms admin

List of VOs configured on this server

astron
astrop
dans
emutd
esr
lofar
lsgrid
magic
ncf
omegac
phicos
pvier
tutor
vldbi
vledut
vlefi
vlemed
vlibu
scia

Figure 2: NL Grid VOMS server

Press OK whenever asked (the web site is protected with a certificate from the DutchGrid CA, which is not recognised by default in Firefox). Using your personal certificate, you can authenticate to the web site.

You will see a list of VOs supported by this server. Select the tutor VO and then you follow the directions shown on your screen. You will see that all the data from your certificate is already filled in. You also have to agree to the usage guidelines shown.

A3.3. Excercises

1. Convert your certificate and private key into a `PKCS#12` file.
2. Import the certificate into a browser.
3. Register for the tutor VO at the VOMS web page. Note that you need to supply a valid e-mail adress for this. Ask one of the tutorial assistants if you are not able to receive mail during the tutorial.
4. Ask one of the tutorial assistants to approve your request.

A4. Setting up the authentication environment

In reality, applying for a certificate may take a day or two. Remember that it requires action by real human beings. The temporary tutorial certificates can be generated on the fly however. The only thing you have to do now is get it and install it in the proper directory.

In this tutorial you will be working from a *User Interface (UI)*. So, first you have to login to the UI, if you have not done so already.

The certificate and private key file should be installed in the *.globus* directory. Note that the the private key file should be read-only and only readable for yourself.

```
$ cd $HOME/.globus
$ ls -l
total 24
-rw-r--r--    1 demo07   demo          249 Aug 10 13:43 certreq18629.cnf
-rw-r--r--    1 demo07   demo         2513 Aug 10 13:43 certreq18629.txt
-rw-r--r--    1 demo07   demo         4499 Aug 10 13:47 usercert.pem
-r--------    1 demo07   demo          963 Aug 10 13:43 userkey.pem
-rw-r--r--    1 demo07   demo         2077 Aug 10 13:43 userrequest.pem
```

Note the protection set on your private key file *userkey.pem*. The permissions are very restrictive and are set this way for a reason: your possession of the private key is the only proof remote sites have that they are indeed talking to you. If you would give that key to someone else (or if it gets stolen), you will be held liable for any damage that may be done to the remote site! In any case, if the user key is world readable or worse, it will not be trusted by the Grid. In case the permission of this file is not read-only for the owner of the file only, please change it using **chmod 400 userkey.pem**.

The private key must also be protected with a pass phrase. You have given this pass phrase when running the makerequest.sh script. If the key gets stolen and you did not set a pass phrase anyone can pretend to be you.

You can always see what is in a certificate using the **openssl** command. This is a toolkit for handling certificates, keys and requests. The table below lists a few useful commands:

show the contents of a certificate:

```
openssl x509 -text -noout -in <usercert.pem>
```

show the contents of a certificate request:

```
openssl req -text -noout -in <userrequest.pem>
```

writes a new copy of the private key with a new pass phrase:

```
openssl rsa -in private_key_file -des3 -out new_private_key_file
```

In principle you are now ready to start with the exercises for working on the Grid (e.g. job submission, data management . . .). But the certificates you have obtained for this tutorial are only useful for the duration of the tutorial (plus some extra days). For prolonged use of the Grid you have to make a request for a real certificate and register with a *Virtual Organisation (VO)*.

A4.1. EXERCISES

1. If you have not yet retrieved your certificate, retrieve your tutorial certificate from the CA server, and store it in the *.globus* directory.

2. Look in your certificate directory, and look inside your certificate using the openssl command. What is your subject name?

B JOB STATUS DEFINITION

As already mentioned in chapter 3, a job can find itself in one of several possible states, the definition of which is given in this table.

Status	*Definition*
SUBMITTED	The job has been submitted by the user but not yet processed by the Network Server
WAITING	The job has been accepted by the Network Server but not yet processed by the Workload Manager
READY	The job has been assigned to a Computing Element but not yet transferred to it
SCHEDULED	The job is waiting in the Computing Element's queue
RUNNING	The job is running
DONE	The job has finished
ABORTED	The job has been aborted by the WMS (e.g. because it took too long, or the proxy certificate expired, etc.)
CANCELLED	The job has been cancelled by the user
CLEARED	The Output Sandbox has been transferred to the User Interface

Only a limited set of transitions between states is allowed. These transitions are depicted in Figure 3.

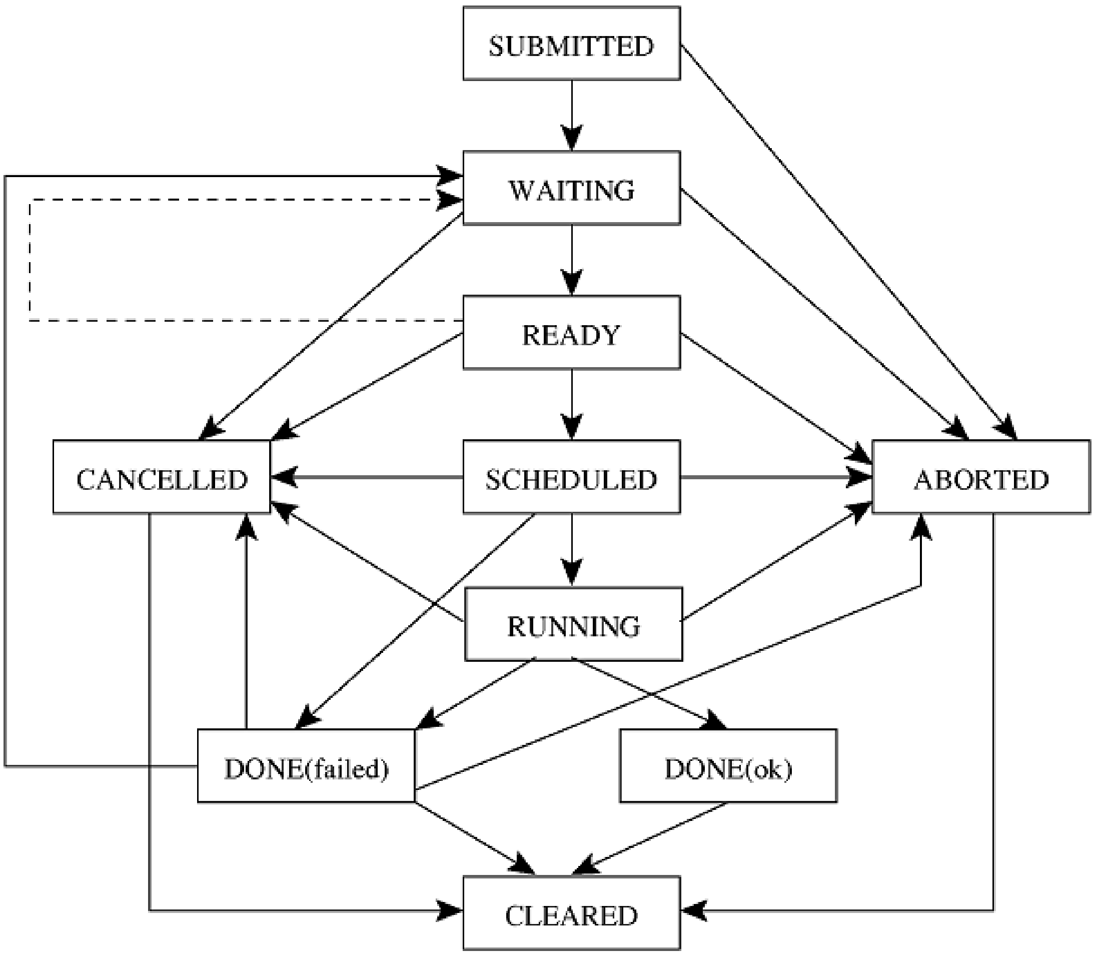


Figure 3: Possible job states in the LCG-2