EGEE

Grid Middleware

HANDOUTS FOR STUDENTS

Date: **June 20, 2006**

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

Abstract: These handouts are provided for people to learn how to use the gLite middleware components to submit jobs on 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, data management and information systems. It is assumed that you are familiar with the basic Linux/UNIX user environment (bash, shell etc.). 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 & Kurt Stockinger, Antony Wilson.

1.1. WORKING ENVIRONMENT

During this tutorial you will make use of a so called User Interface machine. You will have to login to this machine in order to perform the exercises. This should be done using ssh. The user name and password will be provided by one of the tutorial assistants.

Example:

```
$ ssh demo07@ui.matrix.sara.nl
demo07@ui.matrix.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

mu7.matrix.sara.nl:~
demo07$
```

It may be convenient to have a browser running on this same machine. Firefox has been installed on the User Interface, and can be used for the exercises.

2. GETTING ACCESS TO THE GRID

2.1. 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 of 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.

2.2. GETTING A CERTIFICATE

This subsection will try to familiarise you with the procedure of making a certificate request. For the tutorial certificates have already been created for you, but these are only valid for the duration of the tutorial. In this subsection we will show you how to request 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. 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, a 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 use 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.

2.2.1. EXERCISES

1. Request an EDG tutorial certificate at the CA webpage.
2. Run the makerequest.sh script at the User Interface machine. Note that you only have to fill in the paper form when requesting a medium security certificate.
3. Upload your request at the CA webpage. Give "Grid4All" as token.
4. Ask one of the tutorial assistants to approve your request.
5. Retrieve the certificate from the CA webpage (http://certificate.nikhef.nl/edgtutorial/, choose Retrieve on the left hand side).

2.3. 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 (dteam, EarthOb).

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.

2.3.1. 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 Firefox or Internet Explorer in this tutorial we will use Firefox (also installed on the UI), but Internet Explorer will work as well. The certificate in Firefox (version 1.0.x) can be reached from: `Edit -> Preferences -> Advanced -> Security -> View Certificates` In the `Certificate Manager` You can now import your certificate by pressing the Import button.

Firefox will protect its certificate store with a password as well. Enter a good password in the dialogue. 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 Firefox security window.

2.3.2. REQUESTING ACCESS TO A VO

You are now ready to sign the Guidelines and apply for a VO membership. For the NL-Grid infrastructure you can register at the following website (see Figure 1):

https://mu4.matrix.sara.nl:8443/vomses/

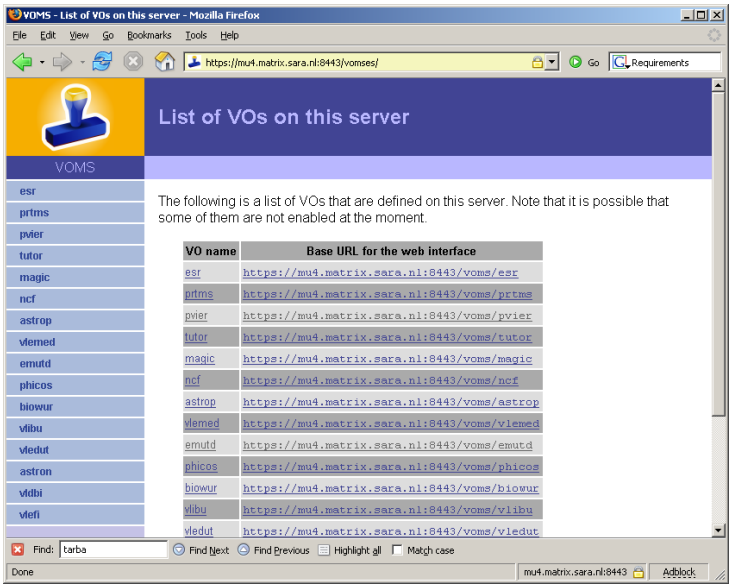

Figure 1: NL Grid VOMS server

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 can go to "New user registration". You will see that all the data from your certificate is already filled in. You also have to agree to the usage guidelines shown.

2.3.3. 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.

2.4. 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 worthless 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.

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 with 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)*.

2.4.1. 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?

2.5. GETTING A PROXY

When you have a certificate you can now request a certificate proxy to be allowed to do the exercises that follow in this manual. The proxy you get will be valid for several hours, long enough for a hands-on afternoon at least. First you have to get onto a machine that understands grid commands. Such computers are called the User Interface (UI) machines and you may have one in your own home institute for which you have an account. In any case, your tutorial organizer should have provided an account for you to use during the course.

In order to get and work with a proxy the following commands can be used:

voms-proxy-init	to get a key, a pass phrase will be required
voms-proxy-info	all gives information of the ticket in use
voms-proxy-destroy	destroys the ticket 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  mu4.matrix.sara.nl:30014 [/O=dutchgrid/O=hosts/OU=sara.nl/CN=mu4.matrix.sara
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=mu4.matrix.sara.nl
attribute : /tutor/Role=NULL/Capability=NULL
timeleft  : 11:59:21
$voms-proxy-destroy
Would remove /tmp/x509up_u503
```

2.6. 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 from:
http://mu7.matrix.sara.nl/tutorial/exercises.tgz

The files can be extracted with:

```
tar xzf exercises.tgz
cd exercises
```

3. JOB SUBMISSION

3.1. INTRODUCTION

The components of the *Workload Management System (WMS)* are shown in Figure 2. It consist of a *User Interface (UI)*, a *Resource Broker (RB)* with an associated *Information Index (II)* and a *Job Submission System (JSS)*, the *Globus Gatekeeper (GK)* with its associated *Local Resource Management System (LRMS)*, a *Worker Node (WN)* and a *Logging and Bookkeeping (LB)* system. The Resource Broker, the Job Submission System and the Logging and Bookkeeping system are central services in the Grid and do not have to be geographically at the same place as the user and/or the User Interface machine. The Gatekeeper and the Local Resource Management System are services that each center providing compute and storage resources to the grid will have. Logically the Gatekeeper, the Local Resource Management System and the Worker Nodes are called a *Computing Element (CE)*. As depicted in Figure 2 the steps to run a job are:

1. User submits the job from the UI to the RB. The RB does the matchmaking to find out where the job might be executed. After having found a suitable Computing Element the Job is transferred to the Job Submission System. At the JSS a file is created in *Resource Specification Language (RSL)*. Also at this stage the *Input SandBox* is created in which all files are specified that are needed by the job.
2. This RSL file, together with the Input SandBox, is then transferred to the Gatekeeper of the Computing Element and the Gatekeeper submits the job to the Local Resource Management System.
3. The LRMS will then send the job to one or more of the free Worker Nodes of the Computing Element.
4. When the job has finished, the files produced by the job are available on the LRMS. The job manager running on the CE notifies the Resource Broker that the job has completed.
5. The RB subsequently retrieves those files specified in the *OutputSandBox*.
6. The RB sends the results (the OutputSandBox) back to the user on the User Interface machine.
7. Queries by the user on the status of the job are sent to the Logging and Bookkeeping Service.

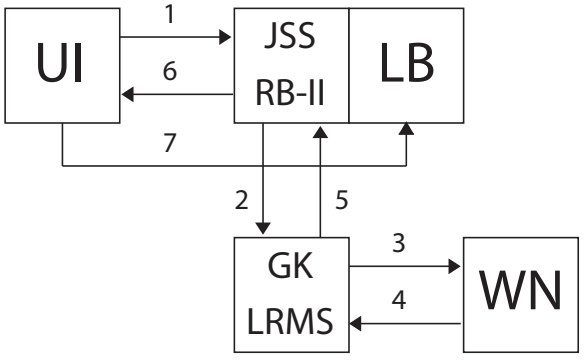

Figure 2: The main Work Load Management System (WMS) components and their operation.

Users access the GRID through a UI machine that allows the user to submit a job, monitor its status, and retrieve the output from the worker node back to a local directory on the UI machine. To do so, a simple *Job Description Language (JDL)* file is compiled. In this file all parameters to run the job are specified.

The relevant commands for submitting a job, querying its status, retrieving the output, and cancelling a job are:

edg-job-submit <job.jdl>	submits a job for which the description is in *job.jdl*
edg-job-status <jobId>	returns the status of a job with job identifier jobId
edg-job-get-output <jobId>	returns the place where tho output of the job can be found
edg-job-cancel <jobId>	cancels the job with job identifier jobId
edg-job-xxx --help	shows the usage of command **edg-job-xxx**

In the next subsections a series of exercises will be presented to familiarise you with the WMS of the Grid.

3.2. 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 [1]:

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

This is close to the minimum job description to be able to run a job on the grid. The Executable in this case is a simple Unix **echo** command and the Argument to this command is the "Hello World" text. You have to specify at least two files: an output file and a file where the possible error messages go. The OutputSandbox contains the files that will be migrated back to the user when the execution has finished.

3.2.1. EXERCISE

1. Run this job on the grid using the **edg-job-submit** command.
2. Read and try to understand the output on the screen.
3. Request the status of the job using the **edg-job-status** command.
4. Get the output from this job using the **edg-job-get-output** command when the job is in the Output Ready state.
5. Check that the job has run correctly by looking into the *message.txt* and *stderror* files.

To submit the "Hello World" job to the EGEE Grid, you must have a valid proxy certificate in the User Interface machine. Obtain one by issuing:

```
$ voms-proxy-init --voms tutor
```

and submit the job:

```
$ edg-job-submit HelloWorld.jdl
```

If the submission is successful, the output is similar to:

[1] To obtain the exercises and directories see 2.6..

```
Selected Virtual Organisation name (from proxy certificate extension): tutor
Connecting to host mu3.matrix.sara.nl, port 7772
Logging to host mu3.matrix.sara.nl, port 9002


**************************************************************************************
                              JOB SUBMIT OUTCOME
 The job has been successfully submitted to the Network Server.
 Use edg-job-status command to check job current status. Your job identifier (edg_jobId)

 - https://mu3.matrix.sara.nl:9000/cOGTsaFRNYhYN9UPApK5yA


**************************************************************************************
```

In case of failure, an error message will be displayed instead, and an exit status different from zero returned.

The command returns the job identifier to the user (*jobId*), which uniquely defines the job and can be used to perform further operations on the job, like interrogating the system about its status, or cancelling it. The format of the jobId is:

```
https://Lbserver_address[:port]/unique_string
```

where `unique_string` is guaranteed to be unique and `Lbserver_address` is the address of the Logging and Bookkeeping server for the job, and usually (but not necessarily) also the Resource Broker.

Note: The jobId does NOT identify a web page.

After a job is submitted, it is possible to see its status and its history, and to retrieve logging information about it. Once the job is finished the jobs output can be retrieved, although it is also possible to cancel it beforehand. The status information of our “Hello World” job can now be obtained by issuing:

```
$ edg-job-status https://mu3.matrix.sara.nl:9000/cOGTsaFRNYhYN9UPApK5yA
```

a possible output could be:

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

Status info for the Job : https://mu3.matrix.sara.nl:9000/cOGTsaFRNYhYN9UPApK5yA
Current Status:     Scheduled
Status Reason:      Job successfully submitted to Globus
Destination:        mu6.matrix.sara.nl:2119/jobmanager-pbs-medium
reached on:         Fri Aug 11 12:46:06 2006
*************************************************************
```

where the current status of the job is showed, along with the time when that status was reached, and the reason for being in that state (which may be especially helpful for the **ABORTED** state). The possible job states are summarised in Appendix A. Finally, the **destination** field contains the ID of the CE where the job has been submited.

Note: Much more information is provided if the verbosity level is increased by using **-v 1** or **-v 2** with the command.

After our job has finished (it reaches the **Done** status), its output can be copied to the UI with the command **edg-job-get-output**:

```
$ edg-job-get-output https://mu3.matrix.sara.nl:9000/cOGTsaFRNYhYN9UPApK5yA

Retrieving files from host: mu3.matrix.sara.nl ( for https://mu3.matrix.sara.nl:9000/cOG

********************************************************************************
                        JOB GET OUTPUT OUTCOME

 Output sandbox files for the job:
 - https://mu3.matrix.sara.nl:9000/cOGTsaFRNYhYN9UPApK5yA
 have been successfully retrieved and stored in the directory:
 /tmp/jobOutput/demo07_cOGTsaFRNYhYN9UPApK5yA

********************************************************************************
```

By default, the output is stored under */tmp/jobOutput*, but it is possible to specify in which directory to save the output using the −**dir** <**path_name**> option.

3.2.2. THE JOB DESCRIPTION LANGUAGE

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 on 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";
```

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 **Output-Sandbox** 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 GLUE attributes of the IS can be used. For a list of GLUE attributes, see Appendix **??**.

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

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

The WMS can be also 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;
```

3.2.3. 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 this accounts is leased to the user. This lease is temporary and you may get a different account the next time you use the resource.

3.2.4. MORE ON EXERCISE JS-1: "HELLO WORLD" ON 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 RB does on the basis of the job description in 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:

edg-job-list-match <job.jdl>	returns the CEs where the job could run

3.2.5. EXERCISE

1. Find out which option of the **edg-job-submit** command you can use to choose your favorite CE to run your job.
2. Find out were your job could possibly run by using the **edg-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 **edg-job-list-match**:

```
$ edg-job-list-match 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("rug.nl", other.GlueCEUniqueId);
```

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

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

In this exercise we 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.

3.3.1. EXERCISE

1. Execute the **ping** command on host *www.sara.nl* 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`)
2. Have a look at the *pinger1.jdl* file and try to understand what it does.
3. 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.
4. Now have a look at the *pinger2.jdl* file and notice the differences.
5. 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
```

3.3.2. 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.

3.4. 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 povray to create the images.

3.4.1. EXERCISE

1. Look at the *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 POVRAY software is required to be present on the node.
2. Look at the *start_povray_morphine.sh* script and make sure you understand what it does.
3. Look for yourself which CEs are able to accept this job
4. Submit the job to the grid and copy the output to your home directory after the job has finished
5. Look at 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, giving as an input argument to it 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 **kview** (after having exported the `$DISPLAY` variable to our current terminal).

Since we require a special software package called Povray, we need to specify this as a requirement in our JDL file. The availability of Povray 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 RB only consider only those CEs which have this software installed.

3.5. EXERCISE JS-4: CHECKSUM ON A LARGE INPUT SANDBOX TRANSFERRED FILE

In this exercise you will transfer via the Input Sandbox a file whose bit wise 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 environmental variable (`$CSTRUE`) the value of the Checksum for the file before the transfer. In the script the Checksum is performed again on the worker node by issuing the **cksum** command on the file and the result is stored in the `$CSTEST` variable. When `$CSTEST` is equal to `$CSTRUE` the file was not corrupted during the transfer.

3.5.1. 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. Change the value for the checksum and then submit the job again and verify you understand the answer that comes back now.

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"};
Arguments     = "none";
```

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]
```

3.6. 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 a different input files each time (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. This exercise will show how to allow you to look up the status of any of the jobs you submitted or to retrieve the output of any of those jobs without having to remember the JobId's of each individual job that was submitted.

3.6.1. EXERCISE

1. Look at the *submitter.sh* script and note that in this case we don't submit this script to the grid but instead run this script locally and it submits jobs to the grid. Note that the scripts takes 2 parameters, the first one is the number of times you want to submit the "HelloWorld" job and the second one is the name of the file where you want to store the JobId's in.
2. Note that in the script the **-o** option of the **edg-job-submit** command is used. Find out what this parameter does.
3. Have a look at the *HelloWorld.jdl* and note that it is the same simple job as in JS exercise 1.
4. Run the script by typing **`./submitter.sh 4 HelloWorld.jid`**
5. Look in the *HelloWorld.jid* file and make sure you understand what it contains.
6. Make sure you understand how this works. Note, if you run the job again you may want to use a different name for the file with the JobId's (e.g. *HelloWorld.jid2* etc).

The shell script (*submitter.sh*) that loops a given amount of times submitting a single job each occasion is the following one:

```
#!/bin/bash
i=0
while [ $i -lt $1 ]
  do edg-job-submit -o $2 HelloWorld.jdl
  i=`expr $i +1`
done
```

From the UI we can now issue the command:

```
$ ./submitter.sh 4 HelloWorld.jid
```

The *HelloWorld.jid* file is a file containing all JobIds for the 4 submitted Jobs. To get info on the Jobs status we can issue `edg-job-status -i HelloWorld.jid`. To collectively retrieve the output we can issue a `edg-job-get-output -i HelloWorld.jid` and then examine the content of the files on the local temporary directories on the UI, where files are retrieved.

3.7. EXERCISE JS-6: MPI JOBS

Running parallel programs has become very important in recent years because of the large increase in the number of available processors. Using message passing between multiple processes with the MPI library is a common way to construct parallel programs. Usually multiple copies of a program are started that know about their rank within this group. This rank can then be used to decide about the distribution of the work, and special routines are used to communicate between the processes. When using the gLite middleware it is also possible to run jobs that make use of this *MPI* message passing library.

In order to run a parallel MPI job the **JobType** attribute has to be specified, and set to *MPICH* (mpich is a freely available version of the mpi library). When specifying `JobType="MPICH"` the number of nodes to use must also be specified using the attribute **NodeNumber**.

An example is:

```
JobType = "MPICH";
NodeNumber = 4;
```

When specifying `JobType="MPICH"` the job will be sent to a CE that supports MPICH. This requirement is automatically added to the job.

The executable given with the Executable attribute will be run in parallel using multiple processors on one or more worker nodes.

3.7.1. EXERCISE

1. Look at the *linpack4.jdl* file in the JSExercise8 directory. Try to understand the contents of the file.
2. Submit the job to the grid.
3. Look at the output and determine the Gigaflop rate for the run.
4. Look at the *linpack1.jdl* file, and try to understand the contents of this file.
5. Submit the job to the same CE you used for the previous job.
6. Look at the output and determine the Gigaflop rate for the run. Do you understand the difference?

In the directory JSExercise8 an example MPI job is available. The job will run the linpack parallel benchmark over 4 processors. The benchmark measures the time needed to solve a large linear system. The *linpack4.jdl* file looks like follows:

```
JobType = "MPICH";
NodeNumber = 4;
Executable = "xhpl";
StdOutput = "linpack.out";
StdError = "std.err";
OutputSandbox = {"linpack.out","std.err"};
InputSandbox = {"xhpl","hpl4/HPL.dat"};
Environment = "P4_GLOBMEMSIZE=128000000";
```

Note the **JobType** and **NodeNumber** attributes. The Enviroment variable is needed to increase the amount of memory available for shared memory communication within a multi-processor node.

The executable **xhpl** and input file *hpl4/HPL.dat* will be transferred tot the worker nodes and run. Because the processor layout is specified in the *HPL.dat* file we need separate input files for single and four processor runs.

Within the outputfile the GFlop rate for the four processor run will be shown like:

```
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR12L8R4       10000   240     2     2              64.59           1.032e+01
--------------------------------------------------------------------------------
||Ax-b||_oo / ( eps * ||A||_1  * N        ) =        0.1014441 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =        0.0240047 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =        0.0053655 ...... PASSED
```

==========

3.7.2. THE GRAPHICAL USER INTERFACE

The EGEE WMS GUI is a Graphical User Interface composed of three different applications:

- a Job Description Language Editor (**edg-wl-ui-jdleditor.sh**);
- a Job Monitor (**edg-wl-ui-jobmonitor.sh**);
- a Job Submitter (**edg-wl-ui-jobsubmitter.sh**).

The three GUI components are integrated although they can be used as standalone applications so that the JDL Editor and the Job Monitor can be invoked from the Job Submitter, thus providing a comprehensive tool covering all main aspects of job management in a Grid environment: from creation of job descriptions to job submission, monitoring and control up to output retrieval.

Note: To be able to use these applications set the `$DISPLAY` environment variable appropriately.

4. DATA MANAGEMENT

4.1. 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 located. They use logical names for the files and 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 allocated 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:
  sfn:// <**SE_hostname**><**SE_Accesspoint**><**VO_path**><**filename**>
  An example SURL is:
  `sfn://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 and TURLs give information about where a physical replica is located. Figure 3 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.

4.1.1. 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.

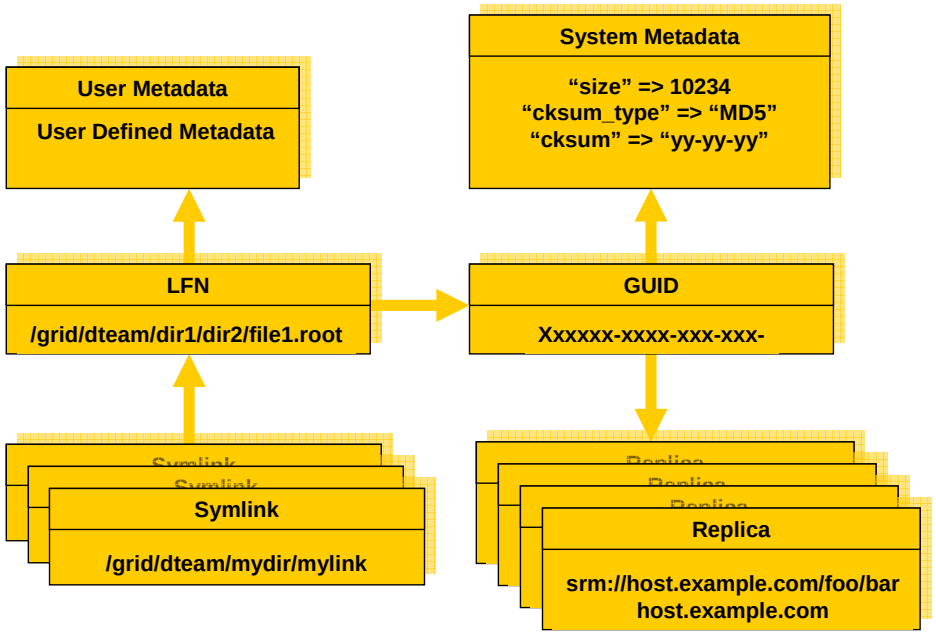

Figure 3: The mapping between GUID, LFNs and physical file names is maintained in the LCG File Catalog.

4.2. 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.

4.2.1. 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).

4.3. 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 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 is 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.

4.3.1. 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.
3. Use **lfc-mkdir** to create one or more subdirectories to use during the rest of the exercises.

4.4. EXERCISE DM-3: FILE REPLICATION WITH THE REPLICA MANGER

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 LRC
- **lcg-aa** add an alias in RMC for a given GUID
- **lcg-ra** remove an alias in RMC for a given GUID
- **lcg-rf** register in the LRC (and optionally in the RMC) a file residing on an SE
- **lcg-uf** unregister in the LRC a file residing on an SE
- **lcg-cp** copy a Grid file to a local destination
- **lcg-gt** get the TURL for a given SURL and transfer protocol
- **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.

4.4.1. 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 (in a machine with a valid proxy):

```
$ lcg-cr --vo tutor -l lfn:/grid/tutor/exercises/hello.f -d mu2.matrix.sara.nl \
> file://$(pwd)/helloworld.f
guid:11c00016-cec0-4530-be19-ff42644da0b0
```

where the only argument is the local file to be uploaded (a fully qualified URI) 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. The **-d** <**destination**> option selects the specified SE as the destination for the file. 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 sfn://mu2.matrix.sara.nl/flatfiles/SE00/tutor/hello.f \
> --vo tutor -l lfn:/grid/tutor/exercises/hello.f file://$(pwd)/helloworld.f
```

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.

4.4.2. 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 pvier lfn:/grid/tutor/exercises/hello.f
sfn://mu2.matrix.sara.nl/flatfiles/SE00/pvier/generated/2005-05-17/filee4028505-9f4a-436
```

> **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 pvier lfn:/grid/tutor/exercises/hello.f
guid:565774f1-9a0d-41a3-9977-fb955bb51b0c
```

4.4.3. 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 teras.sara.nl --vo pvier lfn:/grid/tutor/exercises/hello.f
```

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.

4.4.4. 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, GUI or SURL of the LCG-2 file as its first argument and a local filename or valid TURL as the second, as is shown in the following example:

```
$ lcg-cp --vo pvier lfn:/grid/tutor/exercises/hello.f file://$(pwd)/helloworld.f
```

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

4.4.5. 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 teras.sara.nl --vo pvier lfn:/grid/tutor/exercises/hello.f
```

and

```
$ lcg-del -a --vo pvier lfn:/grid/tutor/exercises/hello.f
```

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

4.4.6. EXERCISE

1. Create a file using e.g. the **touch** or **echo** commands.
2. Find a 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.

4.5. EXERCISE DM-4: USING THE REPLICA CATALOG

After some preliminary replica catalogue interaction in the previous exercise, we now go a bit more into detail with using the replica catalogue with the EGEE Replica Manager and use the following commands:

- **addAlias**, **removeAlias**
- **registerFile**, **unregisterFile**

4.5.1. EXERCISE

1. Create a file using e.g. the **touch** or **echo** commands.
2. Find a SE which you want to use to copy your file to.
3. Misuse the **copyFile** command to copy the created file to the SE.
4. Become aware of your “mistake” and register the file.
5. Add an alias to the registered file.
6. Cleanup and unregister all created replicas and aliases.

4.5.2. REGISTERFILE & UNREGISTERFILE; REGISTERING AND UNREGISTERING GRID FILES

Usually, new files are introduced in LCG-2 copying them from a non-grid resource using **CopyAndRegisterFile**; they are replicated to different SEs using **replicateFile**; and can be copied out of the Grid with **copyFile**. But it is also possible that a file is copied between SEs using **copyFile** (i.e., without registering) or by physically carrying a great amount of data in tapes, or it is possible that a new storage resource that already holds files is added to the Grid (becoming a SE). These files will be in a SE (they will have a valid SURL), but will not be registered in the LCG2 catalogs (i.e., they will not have an associated GUID).

For this situation, the **registerFile** (**lcg-rf**) commands may be useful, which creates a new GUID for a given SURL,

An example of the **lcg-rf** command:

```
$ lcg-rf --vo pvier sfn://teras.sara.nl/home/pvier/hello.f
guid:8733a8ca-e94d-41c8-91b1-6fb11a3c6e4f
```

Likewise, instead of using the **deleteFile**, which both unregister and physically deletes a replica, a user can unregister a replica from the LRC catalogue, without actually deleting it (it can still be accessed on the SE with **copyFile**, for instance). This can be achieved with the **unregisterFile** (**lcg-uf**) command, specifying both the GUID and the SURL to be unregistered, as in:

```
$ lcg-uf --vo pvier guid:8733a8ca-e94d-41c8-91b1-6fb11a3c6e4f \
sfn://teras.sara.nl/home/pvier/hello.f
```

Note: If the last replica of a file is unregistered, then the GUID is also removed from the catalogue.

4.5.3. ADDALIAS & REMOVEALIAS; MANAGING ALIASES

The **addAlias** (**lcg-aa**) command allows the user to add a new LFN to an existing GUID:

```
$ lcg-aa --vo pvier guid:63ab8df9-1bcb-4799-8083-25d7188e1722 lfn:/grid/tutor/exercises/
```

The **removeAlias** (**lcg-ra**) command allows the user to remove an LFN from an existing GUID:

```
$ lcg-ra --vo pvier guid:63ab8df9-1bcb-4799-8083-25d7188e1722 lfn:/grid/tutor/exercises/
```

In order to list the aliases of a file, the user has to use the **edg-replica-metadata-catalog** command, discussed later.

4.6. EXERCISE DM-5: 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 access of the 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.

4.6.1. 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. *mu2.matrix.sara.nl*.) with your uniquely chosen LFN.
3. Change the LFNs in the scripts and JDL files to make then reflect your LFN.
4. Run the *gsiftp.jdl* job (Tip: first try to run the Perl scrips locally, this can save you a lot of debug time).
5. Retrieve, inspect and try to understand the output of the job.

4.6.2. ACCESSING A FILE USING THE GRIDFTP PROTOCOL

We assume that a user has registered a data file (called *values*) within LCG-2, 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 contains of the JDL file follows:

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

The executable (**gsiftp.pl**) is a Perl program, that calls the **edg-rm copyFile** 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 pvier lfn: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
```

4.7. EXERCISE DM-6: TAKING A LOOK AT THE .BROKERINFO FILE

When a Job is submitted to the Grid, we do not know *a-priori* its destination Computing Element.

There are reciprocal "closeness" relationships among Computing Elements and Storage Elements (SE) which are taken into account during the match making phase by the Resource Broker and affect the choice of the destination CE according to where required input data are stored.

In general, we need a way to inform the Job of the choice made by the Resource Broker during the matchmaking so that the Job itself knows which are the physical files actually to be opened. Therefore, according to the actual location of the chosen CE, there must be a way to inform the job how to access the data.

This is achieved using a file called the *.BrokerInfo* file, which is written at the end of the matchmaking process by the Resource Broker and it is sent to the worker node as part of the *InputSandbox*.

The *.BrokerInfo* file contains all information relevant for the Job, like the destination CEId, the required data access protocol for each of the SEs needed to access the input data ("file" - if the file can be opened locally, **rfio** or **gridftp** - if it has to be accessed remotely, etc), the corresponding port numbers to be used and the physical file names (SURLs) corresponding to the accessible input files from the CE where the job is running.

The *.BrokerInfo* file provides to the application a set of methods to resolve the LFN into a set of possible corresponding SURLs (getLFN2SFN).

> **Note:** That by definition the SFN corresponds to an SURL but it does not contain the prefix "srm:". Neither the BrokerInfo API nor CLI distinguishes between SFN and SURL and thus they are identical here.

In addition, there exists a Command Line Interface (CLI) called **edg-brokerinfo** that is equivalent to the C++ API. It can be used to get information on how to access data, compliant with the chosen CE. The CLI methods can be invoked directly on the Worker Node (where the *.BrokerInfo* file actually is

held), and - similarly to the C++ API - do not actually re-perform the matchmaking, but just read the *.BrokerInfo* file to get the result of the matchmaking process.

Note: That the CLI and the API can only be successfully used on the WN where the .BrokerInfo file exists. You will not be able to use the tool on the UI.

In this example exercise we take a look at the *.BrokerInfo* file on the Worker Node of the destination CE, and examine its various fields.

The very basic JDL we are going to use is the following one (*brokerinfo.jdl*):

```
Executable    = "/bin/cat";
StdOutput     = "message.txt";
StdError      = "stderror.log";
OutputSandbox = {"message.txt", "stderror.log"};
Arguments     = " .BrokerInfo";
```

The corresponding set of commands we have to issue are as follows:

```
$ voms-proxy-init --vo tutor
$ edg-job-submit brokerinfo.jdl
$ edg-job-get-output <JobId>
```

The BrokerInfo file is a mechanism by which the user job can access, at execution time, certain information concerning the job, for example the name of the CE, the files specified in the **InputData** attribute, the SEs where they can be found, etc.

The BrokerInfo file is created in the job working directory (that is, the current directory on the WN for the executable) and is named *.BrokerInfo*. Its syntax is, as in job description files, based on Condor ClassAds and the information contained is not easy to read; however, it is possible to get it by means of a CLI, whose description follows.

The **edg-brokerinfo** command has the following syntax:

```
edg-brokerinfo [-v] [-f <filename>] function [parameter] [parameter] ...
```

where **function** is one of the following:

getCE	returns the name of the CE the job is running on.
getDataAccessProtocol	returns the protocol list specified in the **DataAccessProtocol** JDL attribute.
getInputData	returns the file list specified in the **InputData** JDL.
getSEs	returns the list of the storage elements with contain a copy of at least one file among those specified in **InputData**.
getCloseSEs	returns a list of the storage elements close to the CE.
getSEMountPoint <**SE**>	returns the access point for the specified <**SE**>, if it is the list of close SEs of the WN.
getSEFreeSpace <**SE**>	returns the free space on <**SE**>.
getLFN2SFN <**LFN**>	returns the storage file name of the file specified by <**LFN**>, where <**LFN**> is a logical file name of a GUID specified in the **InputData** attribute.
getSEProtocols <**SE**>	returns the list of the protocols available to transfer data in the storage element <**SE**>.
getSEPort <**SE**> <**Protocol**>	returns the port number used by <**SE**> for the data transfer protocol <**Protocol**>.
getVirtualOrganization	returns the name of the VO specified in the **VirtualOrganization** JDL attribute.
getAccessCost	not supported at present.

The **-v** option produced a more verbose output, and the **-f** <**filename**> option tells the command to parse the BrokerInfo file specified by <**filename**>. If the **-f** option is not used, the command tries to parse the file `$EDG_WL_RB_BROKERINFO`.

There are basically two ways for parsing elements from a BrokerInfo file.

The first one is directly from the job, and therefore from the WN where the job is running. In this case, the `$EDG_WL_RB_BROKERINFO` variable is defined as the location of the **.BrokerInfo** file, in the working directory of the job, and the command will work without problems. This can be accomplished for instance by including a line like the following in a submitted shell script:

```
/opt/edg/bin/edg-brokerinfo getCE
```

where the **edg-brokerinfo** command is called with any desired function as its argument.

If, on the contrary, **edg-brokerinfo** is invoked from the UI, the `$EDG_WL_RB_BROKERINFO` variable will be usually undefined, and an error will occur. The solution to this is to include an instruction to generate the *.BrokerInfo* file as ouput of the submitted job, and retrieve it with the rest of generated output, when the job finishes. This can be done for instance with:

```
#!/bin/sh
cat $EDG_WL_RB_BROKERINFO
```

in a submitted shell script.

Then, the file can be accessed locally with the **-f** option commented above.

4.8. EXERCISE DM-7: 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 9.]

A 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 was too long, or the proxy certificated expired, etc.)
CANCELLED	The job has been cancelled by the user
CLEARED	The Output Sandbox has been transferred to the User Interface

Only some transitions between states are allowed. These transitions are depicted in Figure 4.

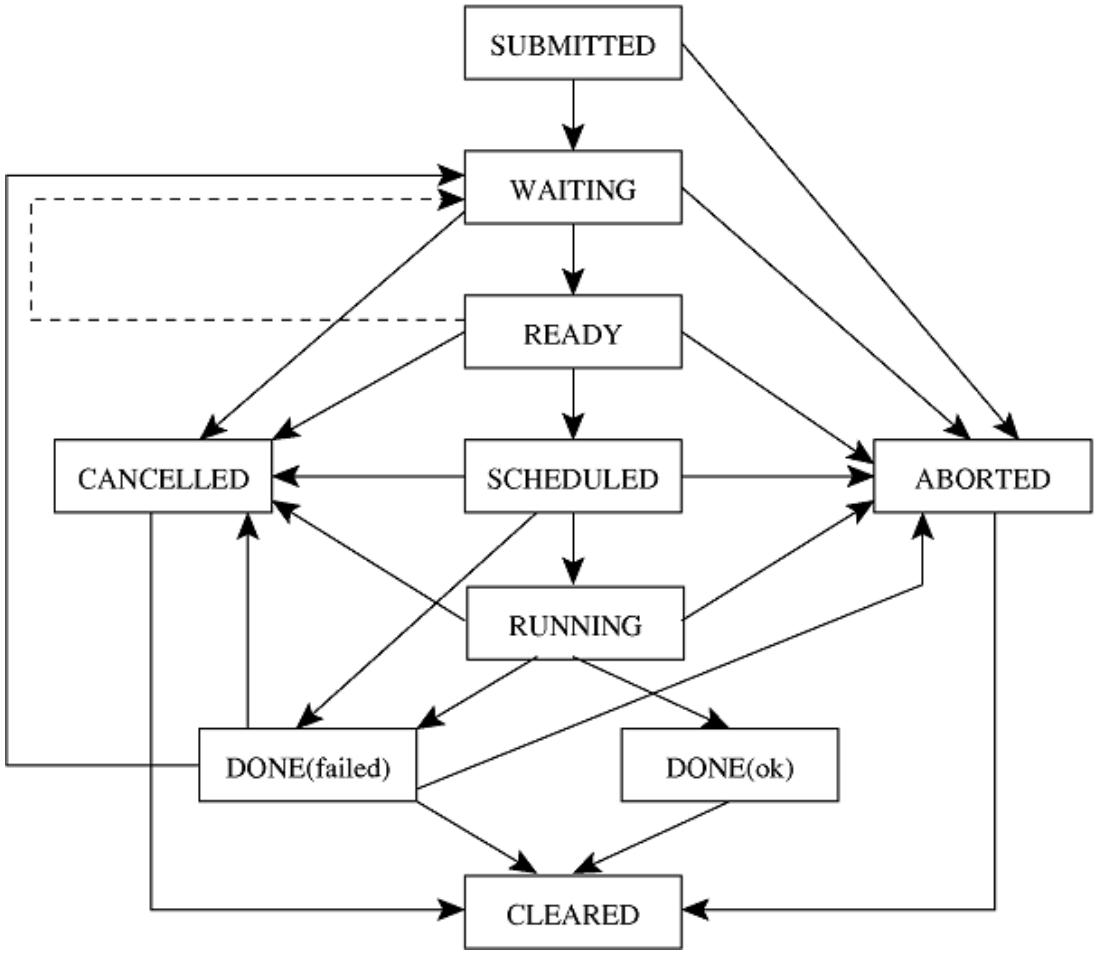

Figure 4: Posible job states in the LCG-2