DataGrid

EDG TUTORIALS

HANDOUTS FOR PEOPLE

Document identifier:	**DataGrid-08-TUT-02.4**
Date:	**25/10/2002**
Work package:	**WP6: Integration Testbed** **WP8: HEP applications**
Partner(s):	**EDG Collaboration**
Lead Partner:	**EDG**
Document status :	**version 2.4**
Author(s):	Mario Reale, Elisabetta Ronchieri Antony Wilson Elisabetta.Ronchieri@cnaf.infn.it A.J.Wilson@rl.ac.uk Mario.Reale@cnaf.infn.it
File: EDG-HANDOUTS-v2.4.DOC	

Abstract

These handouts are provided for people to learn how to use the EDG 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 operating system and know basic editor commands and shell commands..

CONTENT

1. INTRODUCTION

1.1 OVERVIEW

This document leads you through a number of increasingly sophisticated exercises covering aspects of job submission, data management and information systems
The document assumes you are familiar with the basic Linux user environment (bash shell etc.) and that you have obtained a security certificate providing access to the EDG testbed.
This document is designed to be accompanied by a series of presentations providing a general overview of grids and the EDG tools.
Solutions to all the exercises are available online.
The exact details of the EDG testbed (e.g. hostnames and file directories) referred to in this document may change over time so please consult your tutorial leader.

1.2 GETTING A PROXY

Once you have a certificate you can request a ticket to be allowed to do the exercises that follow in this manual. The ticket you receive 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. If so you can use this machine. If you don't know of such a machine you can use the one at CERN which has been set up for this tutorial: testbed010.cern.ch and on which tutor accounts have been created. Your instructor will tell you which account you can use and what your password is.
Now one can get a ticket that allows you to use the testbed. The following commands are available:

grid-proxy-init to get a ticket, a pass phrase will be required
grid-proxy-info –all gives information of the ticket in use
grid-proxy-destroy destroys the ticket for this session
grid-proxy-xxx –help shows the usage of the command grid-proxy-xxx

Examples:

```
[bosk@testbed010 bosk]$ grid-proxy-init
Your identity: /O=dutchgrid/O=users/O=nikhef/CN=Kors Bos
Enter GRID pass phrase for this identity:
Creating proxy ................................................ Done
Your proxy is valid until Thu Sep  5 21:37:39 2002

[bosk@testbed010 bosk]$ grid-proxy-info -all
subject  : /O=dutchgrid/O=users/O=nikhef/CN=Kors Bos/CN=proxy
issuer   : /O=dutchgrid/O=users/O=nikhef/CN=Kors Bos
type     : full
strength : 512 bits
timeleft : 11:59:43

[bosk@testbed010 bosk]$ grid-proxy-destroy -dryrun
Would remove /tmp/x509up_u2899
```

1.3 GETTING THE EXERCISES

Now you are logged onto the testbed and have a ticket so you can start to run some. Some material for the exercises has been prepared in advance and you can copy it (with "wget") to your home directory on the UI machine from:
http://hep-proj-grid-tutorials.web.cern.ch/hep-proj-gridtutorials/jobsubmission.tgz and
http://hep-proj-grid-tutorials.web.cern.ch/hep-proj-grid-tutorials/datamanagement.tgz

Example of what you may see on the screen:

```
[bosk@testbed010 temp]$  wget http://hep-proj-grid-tutorials.web.cern.ch/hep-proj-grid-
tutorials/jobsubmission.tgz
--11:31:44--  http://hep-proj-grid-tutorials.web.cern.ch:80/hep-proj-grid-
tutorials/jobsubmission.tgz
           => `jobsubmission.tgz'
Connecting to hep-proj-grid-tutorials.web.cern.ch:80... connected!
HTTP request sent, awaiting response... 200 OK
Length: 2,031,924 [application/x-compressed]
0K ->   .......... .......... .......... .......... .......... [  2%]
50K -> .......... .......... .......... .......... .......... [  5%]
…etc…
1900K -> .......... .......... .......... .......... .......... [ 98%]
1950K -> .......... .......... .......... ..…… ……...[100%]
11:31:45 (9.55 MB/s) - `jobsubmission.tgz' saved [2031924/2031924]

[bosk@testbed010 temp]$ ls
jobsubmission.tgz
[bosk@testbed010 temp]$ gunzip jobsubmission.tgz
[bosk@testbed010 temp]$ ls
jobsubmission.tar
[bosk@testbed010 temp]$ tar -xvf  jobsubmission.tar
JSexercise1/
JSexercise1/HelloWorld.jdl
…etc…
JSexercise9/HelloWorld.jdl


[bosk@testbed010 temp]$ ls
JSexercise1   JSexercise11  JSexercise3  JSexercise5  JSexercise7  JSexercise9
JSexercise10  JSexercise2   JSexercise4  JSexercise6  JSexercise8  jobsubmission.tar
[bosk@testbed010 temp]$ ls JSexercise1
HelloWorld.jdl

Similarly on can get the Data Management exercises issuing:

[bosk@testbed010 temp]$    wget http://hep-proj-grid-tutorials.web.cern.ch/hep-proj-grid-
tutorials/datamanagement.tgz
```

2. JOB SUBMISSION EXERCISES

2.1 EXERCISE JS-1 : "HELLO WORLD"

In this example we do the simplest job submission to the GRID.
We will involve the basic components of the Workload Management System (WMS). Namely, we will submit a job which simply prints "Hello World", using the */bin/echo* command and takes the "Hello World" string as an argument to this command.(See ref.doc[R2]).
The WMS components are shown in Figure 1. They consist of the User Interface (UI), the Resource Broker (RB, associated to an Information Index (II) LDAP server), the Job Submission System (JSS), the Globus Gatekeeper (GK), the Worker Node (WN) and the Logging and Bookkeeping (LB) system.
Users access the GRID through the User Interface machine, which by means of a set of binary executables written in Python, allows us to submit a job, monitor its status and retrieve its output from the worker node where it has been executed to a local directory on the UI machine.
To do so, we write a simple JDL (Job Description Language) file and issue a `dg-job-list-match <JDL-file-name>` , to check which are the available computing elements to have it executed.
We submit it to the GRID by means of the `dg-job-submit` <JDL-file-name> command.
The system should accept our job and return a unique job identifier (*JobId*).
We verify the status of the execution of the job using `dg-job-status <JobId>`.
After the jobs gets into the *OutputReady* status, we retrieve the output by issuing a `dg-job-get-output <JobId>`, and then verify the output file is in the corresponding local temporary directory on the user interface and that no errors occurred.
Figure 1 shows the complete sequence of operations performed, after having compiled the JDL file and having verified the availability of matching computing elements:

- User submits the job from the UI to the RB [1]
- The RB performs the matchmaking to find the best available CE to execute the job.
- RB transfers the Job to the Job Submission System (JSS), after having created a Condor RSL (Resource Specification Language) file to submit the to the Local Resource Management System (LRMS or batch system such as LSF, PBS etc.)
  The JSS transfers the job (and files specified in the *InputSandbox*) to the Globus Gatekeeper [2]
- The GK sends the Job to the LRMS, which handles the job execution on the available local farm worker nodes. [3]
- After execution on the WN, the produced output is transferred back to the RB and to the UI, using the *OutputSandBox.* [4], [5], [6]
- Queries of the status of the job are addressed to the LB MySql database from the UI machine . [7]

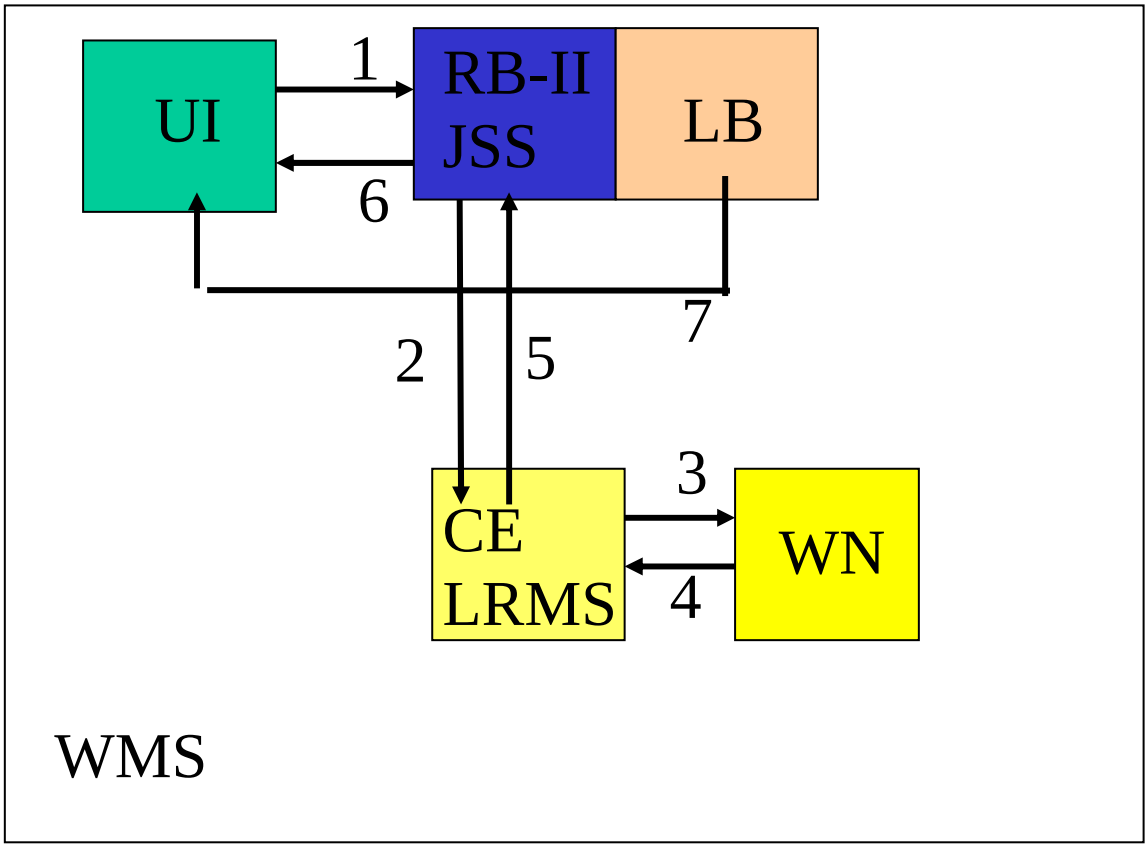


Figure 1: The main WMS components and their operation

The JDL file we will be using is the following one:

```
Executable = "/bin/echo";
Arguments = "Hello World";
StdOutput = "message.txt";
StdError = "stderror";
OutputSandbox = {"message.txt","stderror"};
```

The issued command sequence will be:

```
grid-proxy-init
dg-job-submit HelloWorld.jdl
dg-job-status JobId
dg-job-get-output JobId
```

2.2 EXERCISE JS-2 : LISTING CONTENT OF CURRENT DIRECTORY ON THE WORKER NODE – GRID-MAP FILE

In this example **we will list the files on the local directory of the Worker Node.**
Every user is mapped onto a local user account on the various Computing Elements all over the GRID. This mapping is controlled by the
/etc/grid-security/grid-mapfile file on the Gatekeeper machine and is known as the *grid-mapfile mechanism*: every user (identified by their personal certificate's subject) must be listed in the *grid-mapfile* file and associated to one of the pooled accounts available on the CE for the locally supported Virtual Organization he belongs to.
(SeeFigure 2).
The *grid-mapfile* mechanism, which is part of GSI, requires that each individual user on the grid is assigned a unique local User ID.
The *accounts leasing* mechanism allows access to take place without the need for the system manager to create an individual user account for each potential user on each computing element.
On their first access to a Testbed site, users are given a temporary "leased" identity (similar to temporary network addresses given to PCs by the DHCP mechanism). This identity is valid for the task duration and need not be freed afterwards. If the lease still exists when the user re-enters the site, the same account will be re-assigned to him. (See
http://www.gridpp.ac.uk/gridmapdir/)

We therefore submit here (after `grid-proxy-init`) a job using as executable */bin/ls*, and we redirect the standard output to a file
(JDL attribute : StdOutput = "ListOfFiles.txt";) , which
is retrieved via the *OutputSandbox* to a local directory on the User Interface machine.
The result of the file listing command will be the list of the files on the $HOME directory of the local user account on the Worker Node to which we are mapped. We can issue `dg-job-status JobId` and
`dg-job-get-output JobId` (after the job is in the `OutputReady` status) to get the output.
The exercise is finished at this point. (you are not asked to print the grid-mapfile. It is mentioned here for your information and knowledge)

This very basic example shows how accessing GRID resources is therefore guaranteed only to certified users, with a valid PKI X.509 personal certificate (issued by an officially recognized Certification Authority), whose certificate's subject is listed in the grid-mapfile of the various CE resources, distributed all over the GRID.
To store all subjects of the certificates belonging to the large community of users, each Virtual Organization manages an LDAP Directory server, describing its members. Each user entry od this directory contains at least the URI of the certificate on the Certification Authority LDAP server (and the Subject of the user's certificate, in order to make the whole process faster).
Moreover, EDG users must sign the Acceptable Use Policy (AUP) document in order to receive a certificate and there is another LDAP Directory ("Authorization Directory") which collects the names of people who have signed the AUP. The *grid-map* file on the various GRID CEs is generated by a daemon called *mkgridmap*, which contacts these LDAP servers

(the VO-Authentication server, which will at its turn contact the Certification Authority LDAP server) and the Authorization Directory server, to locally generate (normally once per day) a locally updated version of the */etc/grid-security/grid-mapfile* file. This mechanism is represented in **Figure 3**.

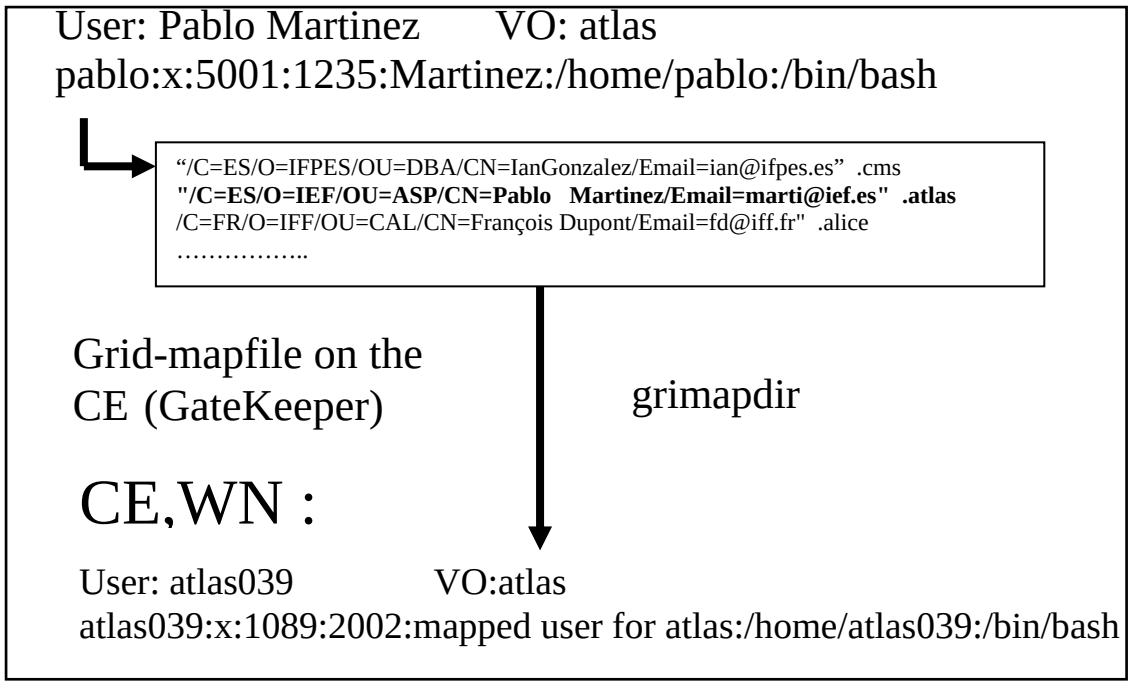


Figure 2 : the grid-mapfile mechanism

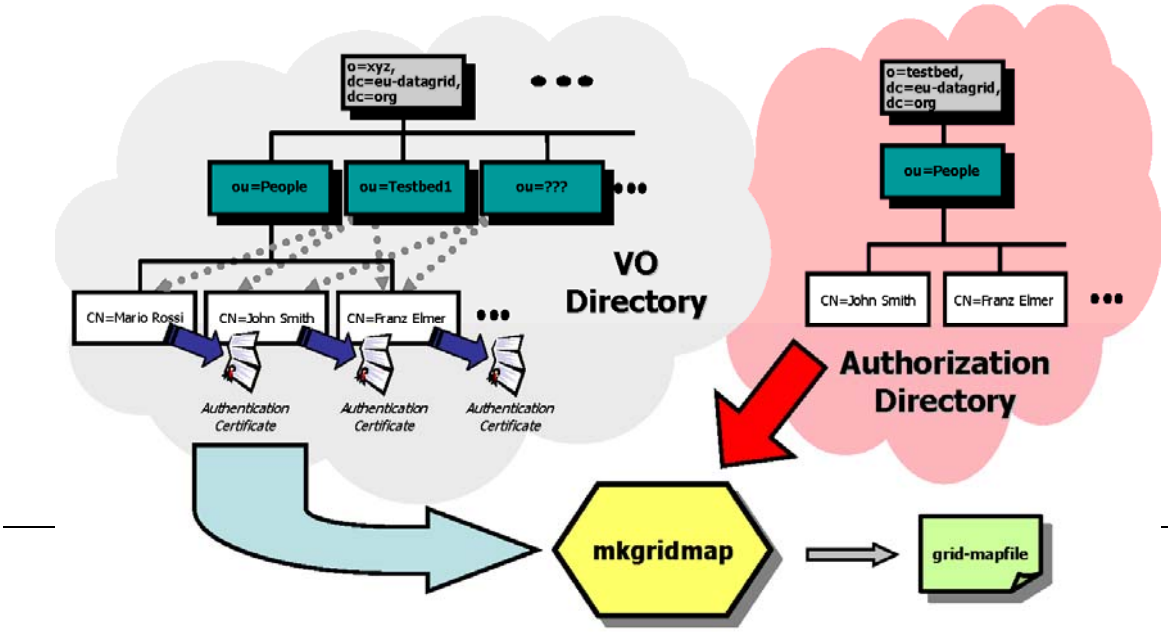


Figure 3 : the mkgidmap daemon, updating the grid-map file

2.3 EXERCISE JS-3 : A SIMPLE PAW PLOT

In this exercise we execute a simple plot on the GRID using PAW, the Physics Analysis Workstation package belonging to the CERNLIB libraries
The PAW executable is installed under the */cern/pro/bin/* directory.
We will run PAW in its batch mode, passing as an argument to the executable the "*-b testgrid*" string, which tells PAW to execute in batch a macro of instructions called *testgrid.kumac*.
Testgrid.kumac opens a file for output in the postscript format store the drawing of the components of a previously created simple vector :

```
ve/create a(10) r  1 2 3 8 3 4 5 2 10 2
ve/print a
for/file 1 testgrid.ps
metafile -1 -111
ve/draw a
close 1
```

Another macro file called *pawlogon.kum*ac sets the PAW environment and options for a given user : in this case just the date on the plots.
The produced output file is therefore *testgrid.ps*, which, after Job Output retrieval can be viewed using *ghostview*.
We need to locally transfer the two required *.kumac* files via InputSandbox to the Worker node and retrieve the produced output file, together with the standard error and standard output file via the OutputSandbox.
Therefore the JDL file (pawplot.jdl) we are going to use looks like this:

```
Executable = "/cern/pro/bin/paw";
Arguments = "-b testgrid";
StdOutput = "StdOutput";
StdError = "stderror";
InputSandbox = {"testgrid.kumac", "pawlogon.kumac"}
OutputSandbox = {"stderror", "StdOutput", "testgrid.ps" };
```

We will submit this job twice : once to the Broker, (leaving it the task of performing the matchmaking process to find the best matching CE), and once directly to an available CE (we get the list of available CEs using the dg-job-list-match command).
The sequence of commands we are going to issue is:

```
grid-proxy-init
dg-job-submit pawplot.jdl
dg-job-status  JobId
dg-job-get-output JobId
dg-job-list-match pawplot.jdl
dg-job-submit -resource  CEid  pawplot.jdl
dg-job-status JobId2
dg-job-get-output JobId2
```

Figure 4 shows the main GRID elements involved in this job's execution example.

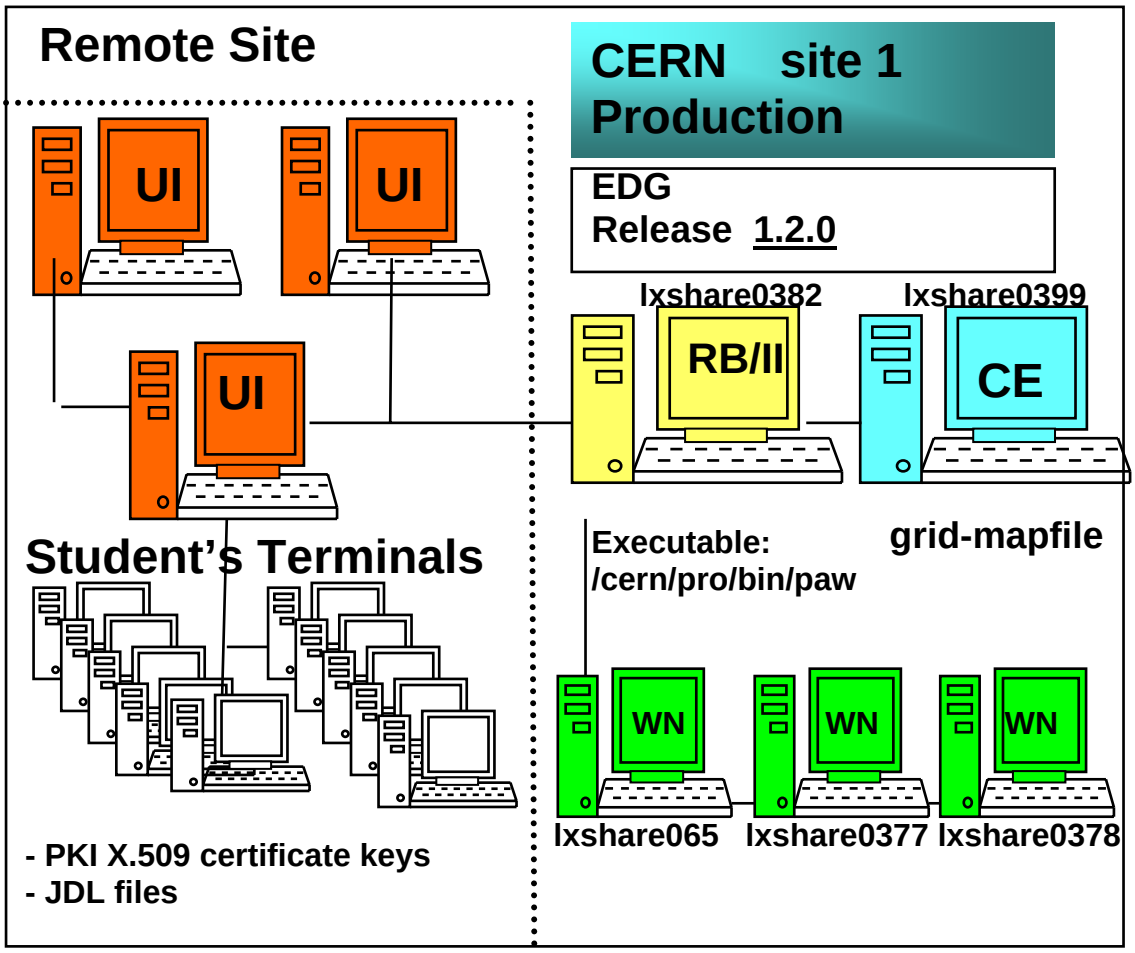


Figure 4 : the main GRID components involved in the execution of a simple PAW plot

2.4 EXERCISE JS-4 : PING OF A GIVEN HOST FROM THE WORKER NODE

In this example we run the simple ping of a remote host from the Worker Node, to start understand the execution of simple commands on the worker nodes. We will execute a ping to a given host from the Worker Node in two ways : directly calling the */bin/ping* executable on the machine and writing a very simple shell script (*pinger.sh*) which does it for us, just to understand how to use shell scripts on the GRID.

We need therefore to write two different JDL files and submit them.
In the first case we directly call the ping executable: (JDL file : *pinger1.jdl*)

```
Executable = "/bin/ping";
Arguments = "-c 5 lxshare0220.cern.ch";
RetryCount = 7;
StdOutput = "pingmessage1.txt";
StdError = "stderror";
OutputSandbox = "pingmessage1.txt";
Requirements = other.OpSys=="RH 6.2";
```

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), (JDL file *: pinger2.jdl*)

```
Executable = "/bin/bash";
Arguments = "pinger.sh lxshare0220.cern.ch";
RetryCount = 7;
StdOutput = "pingmessage2.txt";
StdError = "stderror";
InputSandbox = "pinger.sh";
OutputSandbox = {"pingmessage2.txt","stderror"};
Requirements = other.OpSys=="RH 6.2";
```

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

```
#!/bin/bash
/bin/ping –c 5 $1
```

As a related problem, try to build similar examples
for */bin/pwd* or */usr/bin/who*, in both ways : directly and via a shell script.

As usual, the set of commands we are going to issue in both cases is the following gone:
(of course changing the name of the JDL from *pinger1.jdl* to *pinger2.jdl* in the second case)

```
grid-proxy-init
dg-job-submit pinger1.jdl
dg-job-status JobId
```

`dg-job-get-output JobId`

Figure 5 shows the main difference between the two ways of operating and suggests a third one.

JDL file – 1 :
Executable = “/bin/ping”;
Arguments = “-c 5 lxshare0393.cern.ch”;

JDL file - 2:
Executable = “/bin/bash”;
Arguments = “pinger.sh lxshare0393.cern.ch”;
InputSandbox = {“pinger.sh”, .., .. }

pinger.sh:
#!/bin/bash
/bin/ping –c 5 $1

JDL file - 3: There is even a third way of executing the ping command : directing calling the pinger Shell script as executable:
Executable = “pinger.sh”;
Arguments = “lxshare0393.cern.ch”;
InputSandbox ={“pinger.sh”, .., .. }

Figure 5 : three different ways of working : 1 direct , 2 via shell script, 3 directly

2.5 EXERCISE JS-5 : RENDERING OF SATELLITE IMAGES : USING DEMTOOLS

We will launch the DEMTOOLs program on the GRID, which is a satellite images rendering program : staring from ASCII files in the .DEM format (Digital Elevation Model, usually acquired by high resolution remote sensing satellites), produces graphical virtual reality images, in the .wrl file format, which can then be browsed and rotated using the *lookat* command, after output retrieval.
We need to specify in input the satellite remote sensing data stored in the 2 files, referring to satellite views of Mont Saint Helens and the Grand Canyon, called *mount_sainte_helens_WA.dem* and *grand_canyon_AZ.dem*, and after the job's execution we need to specify the name of the 2 produced images we want returned to our UI machine. The data flow is shown in Figure 6. The JDL file (*demtools.jdl*) is the following one:

```
Executable = "/bin/sh";
StdOutput = "demtools.out";
StdError = "demtools.err";
InputSandbox =
{"start_demtools.sh","mount_sainte_helens_WA.dem","grand_canyon_AZ.dem"};
OutputSandbox = \
{"demtools.out","demtools.err","mount_sainte_helens_WA.ppm","mount_sainte_helens_WA
.wrl",
"grand_canyon_AZ.ppm", "grand_canyon_AZ.wrl"};
RetryCount = 7;
Arguments = "start_demtools.sh";
Requirements = Member(other.RunTimeEnvironment,"DEMTOOLS-1.0");
```

Note that we need to expressly require that the destination CE should have the DEMOTOOLs software installed : we do so in the last line of the JDL file.
The launching shell script (*start_demtools.sh*) used is the following one:

```
/usr/local/bin/dem2ppm mount_sainte_helens_WA.dem  mount_sainte_helens_WA.ppm
/usr/local/bin/dem2vrml -r 2 mount_sainte_helens_WA.dem mount_sainte_helens_WA.wrl
/usr/local/bin/dem2ppm grand_canyon_AZ.dem grand_canyon_AZ.ppm
/usr/local/bin/dem2vrml -r 2 grand_canyon_AZ.dem grand_canyon_AZ.wrl
```

To check the effective presence of available CEs for the job to be correctly executed, as usual, we can issue a `dg-job-list-match demtools.jdl`.
After we checked (issuing a `dg-job-status JobId`) that the Job reached the OutputReady status, we can issue a `dg-job-get-output JobId` to retrieve the output locally on the User Interface machine and take a look at the produced images going in the local directory where the output has been returned using `lookat grand_canyon_AZ.wrl` and `lookat mount_sainte_helens_WA.wrl`.

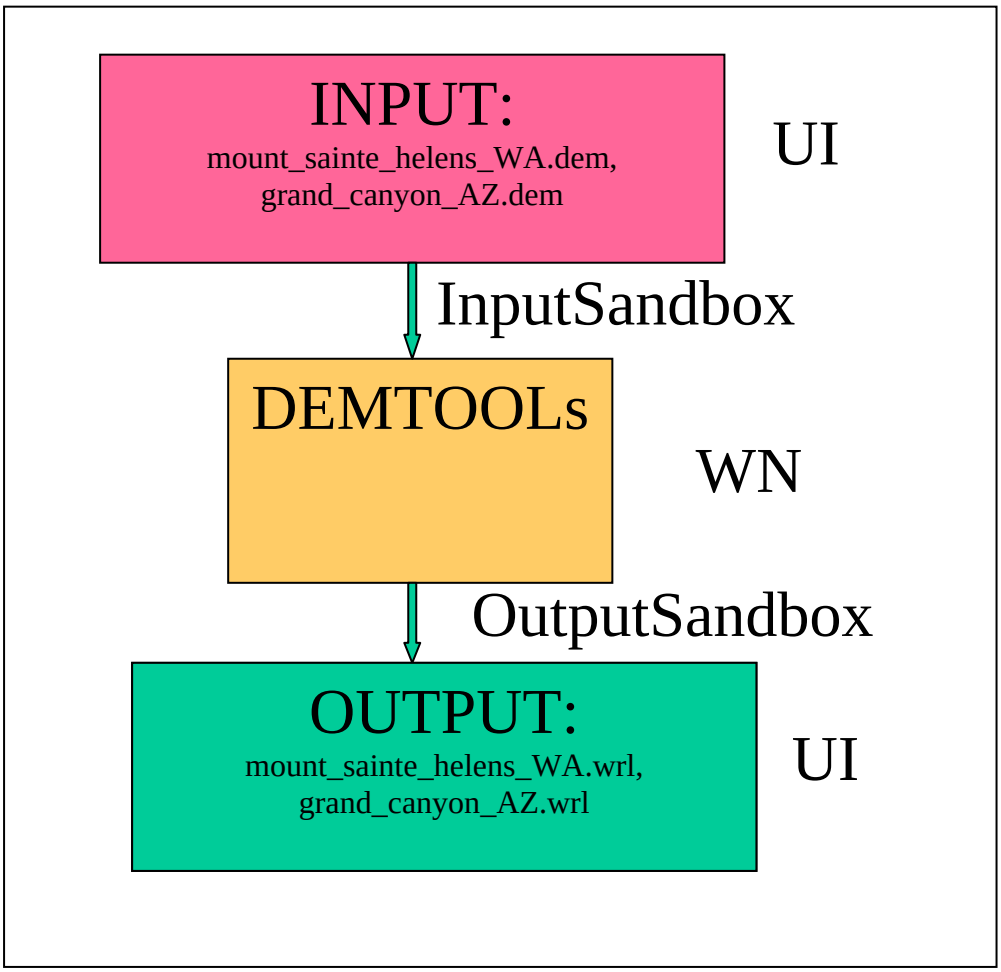


Figure 6 : Data Flow for example exercise JS-5 on DEMTOOLs

2.6 EXERCISE JS-6 : USING POVRAY TO GENERATE VISION RAY-TRACER IMAGES.

We want to launch POVRAY (http://www.povray.org), a graphical program, which starting from ASCII files (in this specific example - the *pipeset.pov* file) in input, creates in output Vision Ray-Tracer images in the .png file format.
We will do it using the GRID, submitting a proper JDL file which executes an ad-hoc shell script file. In this example the out coming image is the one of a pipe duct.
We need therefore to compile our JDL file, specifying in the InputSandbox all the required ASCII files to be used by the program and the corresponding shell script. Then we submit it to the WMS system.
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_pipe.sh*):

```
#!/bin/bash
mv pipeset.pov OBJECT.POV
/usr/local/bin/x-povray /usr/local/lib/povray31/res640.ini
mv OBJECT.png pipeset.png
```

We can finally, after having retrieved the Job, examine the produced image using Netscape or Explorer or using *xv* (after having exported the $DISPLAY variable to our current terminal)

The JDL file we are going to use is the following one (*povray_pipe.jdl*):

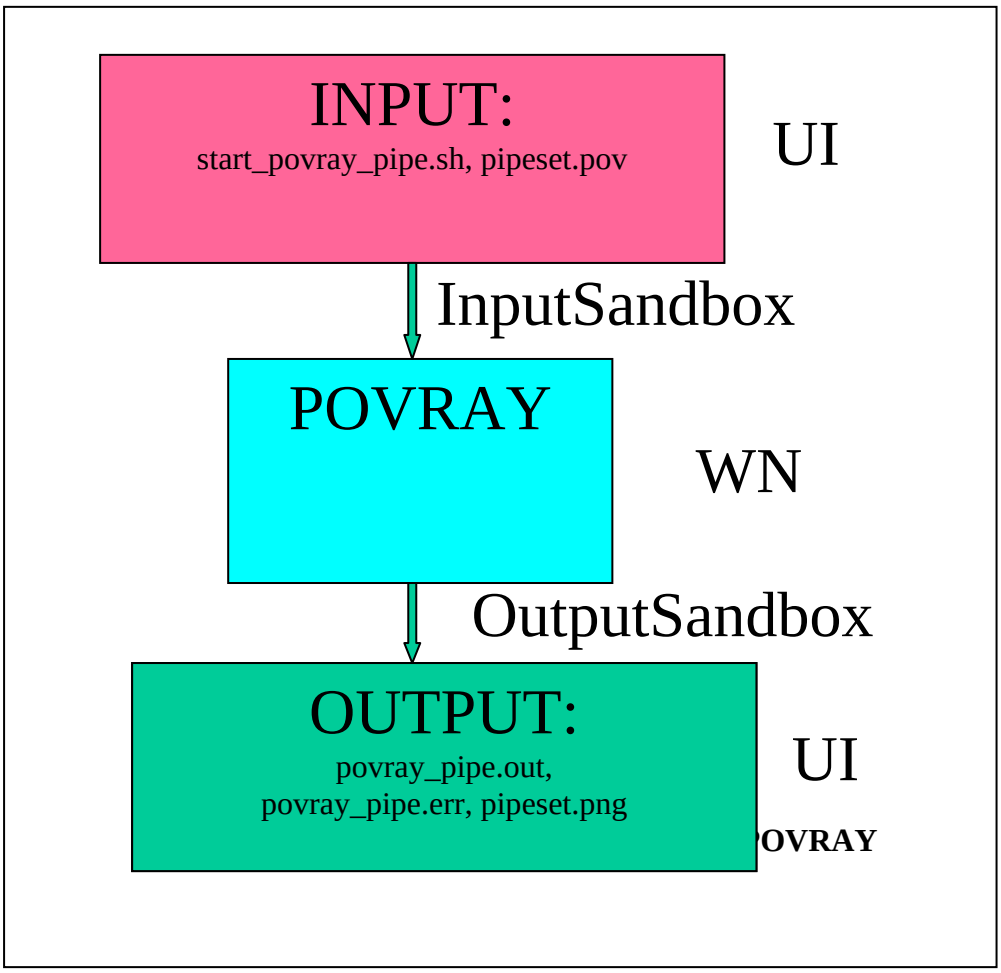
```
Executable = "/bin/sh";
StdOutput = "povray_pipe.out";
StdError = "povray_pipe.err";
InputSandbox = {"start_povray_pipe.sh","pipeset.pov"};
OutputSandbox = {"povray_pipe.out","povray_pipe.err","pipeset.png"};
RetryCount = 7;
Arguments = "start_povray_pipe.sh";
Requirements = Member(other.RunTimeEnvironment,"POVRAY-3.1");
```

Since we are requiring a special software executable,
(*/usr/local/bin/x-povray/usr/local/lib/povray31/res640.ini*), which is identified by the GRID Run Time Environment flag called "POVRAY-3.1", we notice here that we need to specify it in the Requirements classAd, in order to consider (during the matchmaking done by the Broker to select the optimal CE to send the job to) only those CEs which have this software installed. This is done in the last line of the JDL file.

The set of sequence commands we are going to issue is the following one:

```
grid-proxy-init;
dg-job-list-match povray_pipe.jdl;
dg-job-submit povray_pipe.jdl;
dg-job-status JobId; dg-job-get-output JobId.
```

The data flow via Input and Output Sandboxes for this exercise is shown in Figure 7 . To take a look at the produced file : *lookat pipeset.png*.



2.7 EXERCISE JS-7 : GENERATE AN ALICE GEANT3 ALIROOT SIMULATED EVENT

We are going to generate an ALICE (http://alice.web.cern.ch/Alice) simulated event on the GRID.
The event is a reduced track number Lead-Lead collision, and it is generated by Aliroot (http://alisoft.cern.ch/offline/aliroot-new/howtorun.html), which is a GEANT 3 based generator for Monte Carlo simulated events.
We write therefore a JDL file using the */bin/bash* executable with in argument the name of the script we want to executed. This scripts basically sets some environmental variables and then launches Aliroot with an appropriate configuration file. We will need to transfer all required files to the WN, therefore filling them in the Input Sandbox. We will then retrieve the output and check the correct presence of the files in output, and take a look at the produced event running Aliroot in the Display mode. The required shell script to be used sets some relevant environment variables and renames one file (*rootrc*) for Aliroot; then it starts the aliroot programs, "compiling on the flight" the file *grun.C* and finally generating the event, in the *galice.root* file.

```
#!/bin/sh
mv rootrc $HOME/.rootrc
echo "ALICE_ROOT_DIR is set to: $ALICE_ROOT_DIR"
export ROOTSYS=$ALICE_ROOT_DIR/root/$1
export PATH=$PATH:$ROOTSYS/bin
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
export ALICE=$ALICE_ROOT_DIR/aliroot
export ALICE_LEVEL=$2
export ALICE_ROOT=$ALICE/$ALICE_LEVEL
export ALICE_TARGET=`uname`
export LD_LIBRARY_PATH=$ALICE_ROOT/lib/tgt_$ALICE_TARGET:$LD_LIBRARY_PATH
export PATH=$PATH:$ALICE_ROOT/bin/tgt_$ALICE_TARGET:$ALICE_ROOT/share
export MANPATH=$MANPATH:$ALICE_ROOT/man
$ALICE_ROOT/bin/tgt_$ALICE_TARGET/aliroot -q -b grun.C
```

The JDL file we need is the following one:

```
Executable = "/bin/sh";
StdOutput = "aliroot.out";
StdError = "aliroot.err";
InputSandbox = {"start_aliroot.sh","rootrc","grun.C","Config.C"};
OutputSandbox = {"aliroot.err","aliroot.out","galice.root"};
RetryCount = 7;
Arguments = "start_aliroot.sh 3.02.04 3.07.01";
Requirements = Member(other.RunTimeEnvironment,"ALICE-3.07.01");
```

After output retrieval, we can take a look at the event launching aliroot in the Display mode by issuing *aliroot display.C* , after having copied the *rootrc* file to our home directory and having renamed it to *.rootrc* and having sourced the aliroot.sh file. The generated event looks like the one reported in Figure 8.
Figure 9 reports the data flow (Input/Output) for this aliroot example run.

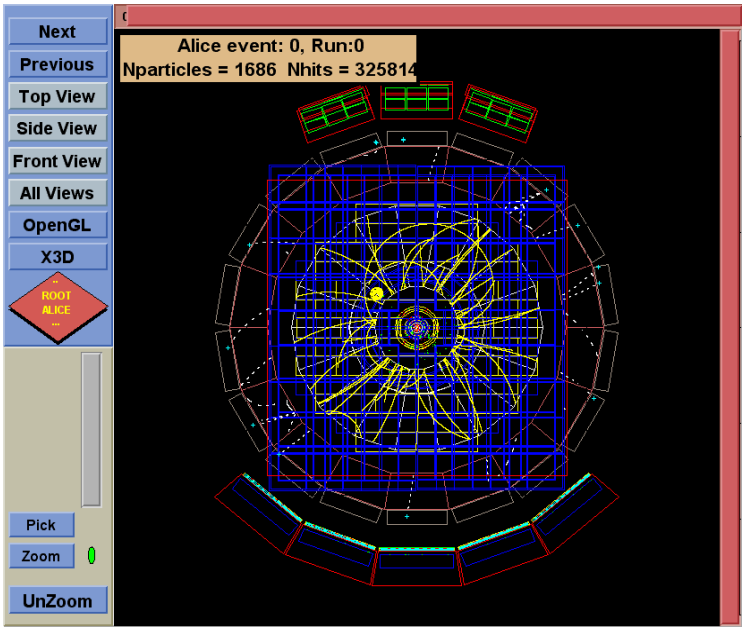


Figure 8: The ALICE Aliroot GEANT 3 simulated event

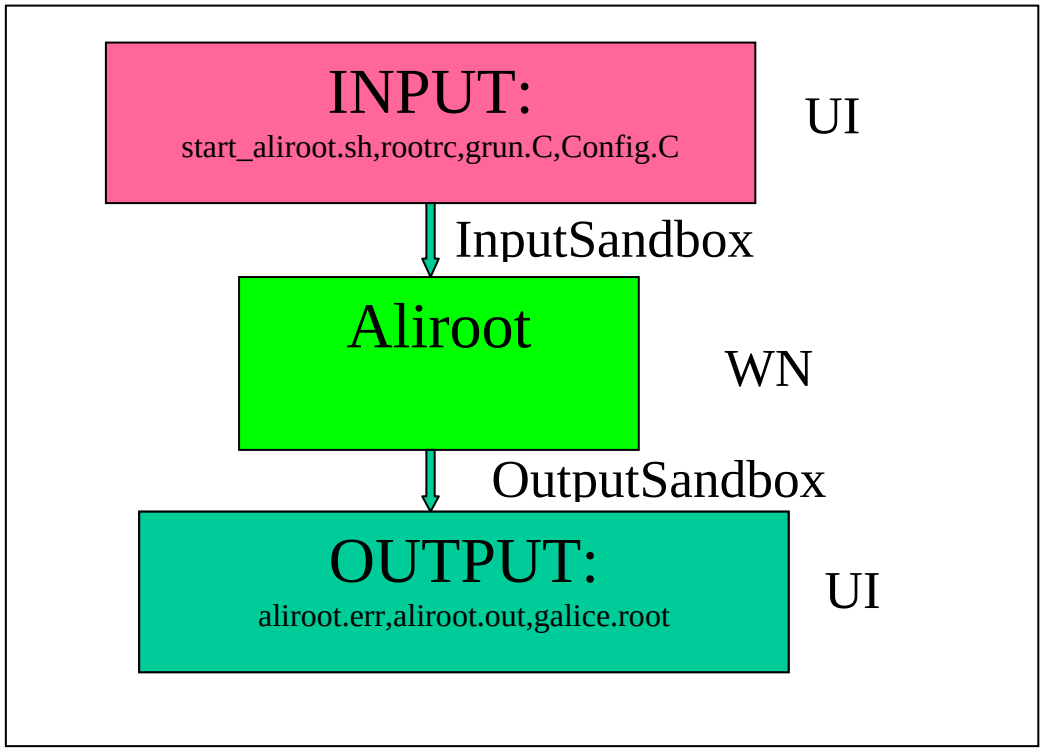


Figure 9 : Input/Output data flow to the Worker Node for the Generation of an ALICE simulated event.

2.8 EXERCISE JS-8 : CHECKSUM ON A LARGE INPUTSANDBOX TRANSFERRED FILE

In this example exercise we transfer via InputSandbox a large file (~20MB) , whose bit wise checksum is known, and check that the file transfer didn't corrupt by any mean the file by performing again the checksum on the worker node and comparing the two results.
We will use a shell script (*ChecksumShort.sh*), which exports in an environmental variable ($CSTRUE) the value of the Check Sum for the file before file transfer, and then performs again the check locally on the Worker node issuing the *cksum* command on the file *short.dat* and exporting the result in the $CSTRUE environmental variable: The test result is then OK if the two values are equal:

```
#!/bin/sh
# The true value of the checksum.
export CSTRUE="2933094182 1048576 short.dat"

# Create a 20MB file with the given seed.
echo "True checksum: '${CSTRUE}'"
export CSTEST="`cksum short.dat`"
echo "Test checksum: '${CSTEST}'"
echo "Done checking."
if [ "${CSTRUE}" = "${CSTEST}" ]; then
   export STATUS=OK;
else
   export STATUS=FAIL;
fi
# Finished.
echo "Goodbye. [${STATUS}]"
```

The JDL file we are going to use is the following one (*ChecksumShort.jdl*) :

```
Executable    = "ChecksumShort.sh";
Arguments     = "none";
StdOutput     = "std.out";
StdError      = "std.err";
InputSandbox  = {"ChecksumShort.sh", "short.dat"};
OutputSandbox = {"std.out","std.err"};
```

If everything works fine (and the gridFTP InputSandbox transfer was OK) in the std.out file we should find this content:

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

The data flow for this exercise is shown in

Figure 10.

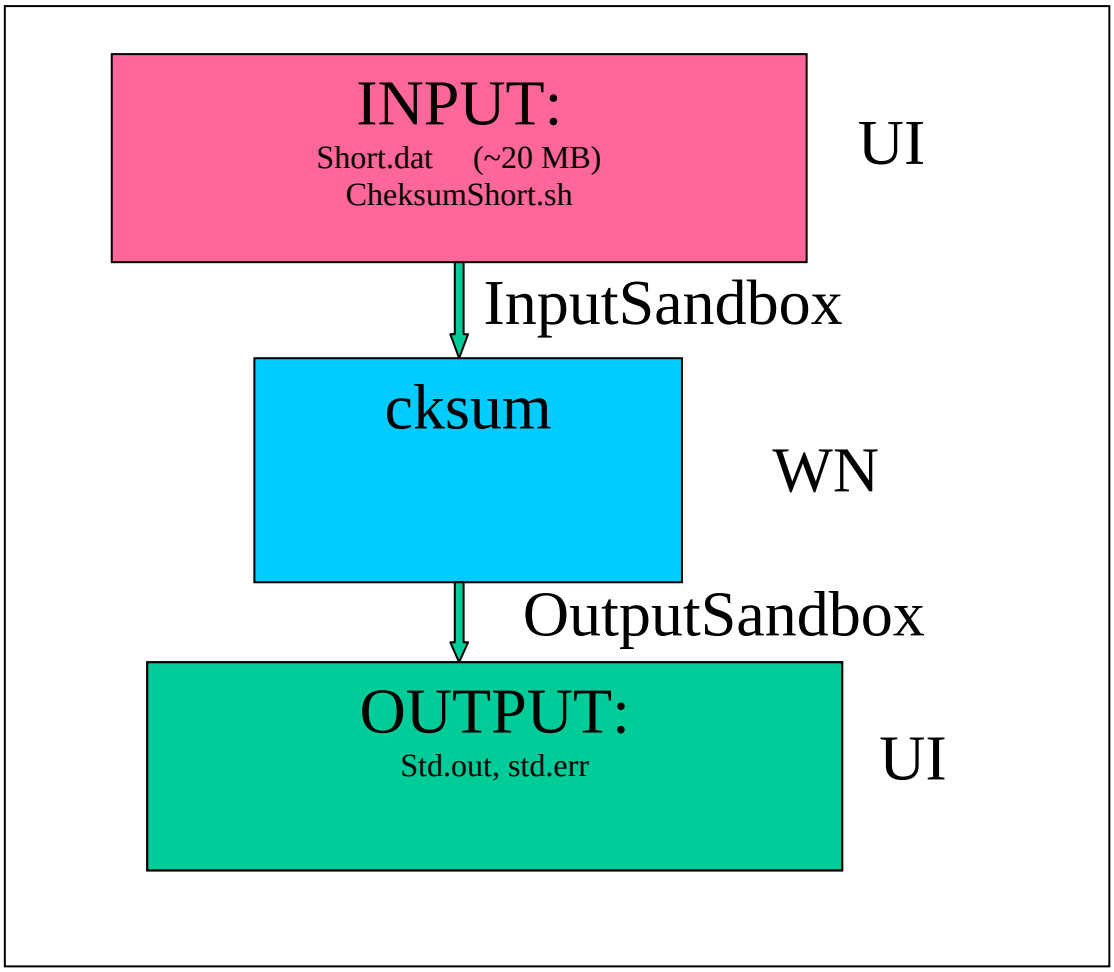


Figure 10: Checksum example: data flow

As usual, the sequence of commands we are going to issue is the following one:

```
dg-job-submit ChecksumShort.jdl
dg-job-status JobId
dg-job-get-output JobId
```

We finally need then to change directory to the local directory where files have been retrieved and examine the std.out to check the result of the checksum test.

2.9 EXERCISE JS-9 : A SMALL CASCADE OF HELLO WORLD JOBS.

Goal of this exercise is to submit a small cascade of elementary “Hello World” jobs,
To explore the tools provided to handle numerous JobIds during the parallel execution of a large set of Jobs.
For our purpose we can log in twice on the User Interface, in order to have at our disposal two simultaneous sessions from which to submit jobs.
We than use a shell script that loops a given amount of times submitting a single job on each occasion (*submitter.sh*) :

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

We can, from each UI session, after grid-proxy-init, issue the command `./submitter.sh 4 HelloWorld.jid`. The “.jobids” file is a file containing all JobIds for the 4 submitted Jobs. This can be done using the “`-o filename`” option in the `dg-job-submit` command to submit jobs.
We can then use the provided shell script called *analysis.sh*
(which requires in input the name of the .jobids file) to issue the `dg-job-status` for all jobs and extract in a formatted way some relevant information. Otherwise we can also issue directly a `dg-job-status –i HelloWorld.jobids` to get info on the Job’s status.
To collectively retrieve the output we can issue a `dg-job-get-output –i HelloWorld.jobids` and then examine the content of the files on the local temporary directories on the UI , where files are retrieved. The JDL file we use is the simplest one of all :

```
Executable = "/bin/echo";
Arguments = "Hello World";
StdOutput = "message.txt";
StdError = "stderror";
OutputSandbox = {"message.txt", ”stderror”};
```

2.10 EXERCISE JS-10 : A SMALL CASCADE OF ALICE ALIROOT MC EVENTS JOBS.

The aim of this exercise is to submit a small cascade of 5 ALICE aliroot jobs to represent a mini production of ALICE simulated events on the GRID.
We will therefore use a submitter shell script, which will produce a loop in which the submission of the *aliroot.jdl* (see exercise JS-7) is performed.
Like in the previous exercise (JS-9), we will store all JobIds in a file (*aliroot.jobids*), and use this file to handle the jobs (getting their status and retrieving the output).

We run the *alirootSubmitter.sh* shell script, that will issue the `dg-job-submit -o aliroot.jobids` commands. We then monitor the status of the jobs using the analysis.sh script. After all jobs have been executed, we will issue a `dg-job-get-output -i aliroot.jobids` to retrieve the output of all the executed jobs.
We require the presence of the ALICE
Experiment software installed on the Computing Element's worker nodes: this is done in the last line of the following JDL file
(*aliroot.jdl*) :

```
Executable = "/bin/sh";
StdOutput = "aliroot.out";
StdError = "aliroot.err";
InputSandbox = {"start_aliroot.sh","rootrc","grun.C","Config.C"};
OutputSandbox = {"aliroot.err","aliroot.out","galice.root"};
RetryCount = 7;
Arguments = "start_aliroot.sh 3.02.04 3.07.01";
Requirements = Member(other.RunTimeEnvironment,"ALICE-3.07.01");
```

The corresponding shell script (*start_aliroot.sh*) is (like in exercise JS-7) :

```
#!/bin/sh
mv rootrc $HOME/.rootrc
echo "ALICE_ROOT_DIR is set to: $ALICE_ROOT_DIR"
export ROOTSYS=$ALICE_ROOT_DIR/root/$1
export PATH=$PATH:$ROOTSYS/bin
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
export ALICE=$ALICE_ROOT_DIR/aliroot
export ALICE_LEVEL=$2
export ALICE_ROOT=$ALICE/$ALICE_LEVEL
export ALICE_TARGET=`uname`
export
LD_LIBRARY_PATH=$ALICE_ROOT/lib/tgt_$ALICE_TARGET:$LD_LIBRARY_PATH
export PATH=$PATH:$ALICE_ROOT/bin/tgt_$ALICE_TARGET:$ALICE_ROOT/share
export MANPATH=$MANPATH:$ALICE_ROOT/man
$ALICE_ROOT/bin/tgt_$ALICE_TARGET/aliroot -q -b grun.C
```

and the *alirootSubmitter.sh* shell script is the following one:

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

Finally, we need to issue from the User Interface machine the following commands to start the production:

```
grid-proxy-init
./alirootSubmitter.sh 5 aliroot.jobids
./analysis aliroot.jobids
dg-job-get-output -i aliroot.jobids
```

2.11 EXERCISE JS-11 : TAKING A LOOK AT THE .BROKERINFO FILE

When a Job is submitted to the GRID, we don't know a-priori its destination Computing Element.
There are reciprocal "closeness" relationships among Computing Elements and Storage Elements 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.
For jobs which require accessing input data normally resident on a given GRID SE, the first part of the matchmaking is a query to the Replica Catalog to resolve each required LFN into a set of corresponding PFNs. Then once a set of job-accessible SEs is available, the matchmaking process selects the optimal destination CE – i.e. the "closest" to the most accessed Storage Element.
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 CE-id, 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 corresponding to the accessible input files from the CE where the job is running.

The .BrokerInfo file is used by C++ APIs called the *RCb APIs* (Replica Catalog – BrokerInfo) which parse the file content (to extract Broker Info information) and provide to the application a set of methods to resolve the LFN into a set of possible corresponding PFNs, (getPhysicalFileNames), or getting the optimal physical file name starting from a list of PFNs and the corresponding required protocols

(getBestPhysicalFileName).

Also, a set of line interface commands are available, doing exactly the same thing, for the application, in order to get information on how to access data, compliant with the chosen CE. This CLI tool is called `edg-brokerinfo` (aka BIcli - BrokerInfo command line interface), its methods can be invoked directly on the Worker Node (where the .BrokerInfo file actually is held), and – similarly to the RCb API – do not actually re-perform the matchmaking, but just read the *.BrokerInfo* file to get the result of the matchmaking process.

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 (*brokerinfo.jdl*) :

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

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

```
grid-proxy-init
dg-job-submit brokerinfo.jdl
dg-job-get-output JobId
```

This exercise leads naturally to the Data Management Exercises, where we will see how to manage data over the GRID.

3. DATA MANAGEMENT EXERCISES

So far we have seen (exercises JS-1 to JS-11) how to submit jobs, query the job's status, retrieve the output, essentially packaging everything needed by our job in the Input and Output sandboxes. But Input and Output Sandboxes isn't all about Data Management on the GRID.
On the contrary, the idea of using sandboxes is more thought as for providing small auxiliary files to the jobs, like data cards for the various Monte Carlo events generating programs, or small required libraries for the executables.
We have still to discover and understand how to handle large files, distributed all over the GRID, how to make other users aware of the availability of files owned by us on our Storage Elements and how to be able to find out where to find useful replicas for us to be used in our jobs.
For this purposes there is a set of tools conceived to replicate files from one Storage Element to another, to register files on Catalogs, to publish these Catalogs to other users and get a copy of the files we are interested in, by subscription of our replica server (on our SE) to other sites we consider relevant for us.
The main tool conceived at this purpose is the edg-replica-manager (http://cern.ch/grid-data-management/edg-replica-manager) which has a complete set of tools set up to create replicas of files between Storage Elements and register files in the Replica Catalogs. Another special purpose replication tool is GDMP (Grid Data Mirroring Package, http://project-gdmp.web.cern.ch/project-gdmp/) which has more advanced features to replicate file sets between SEs. For simple file replication we recommend to use edg-replica-manager. For more advanced users GDMP provides a rich set of tools.In general, all Storage Elements registered to the GRID publish, through the GRID Information System, the location of the directory where the are available to store files, and this directory is usually VO dependent : each defined VO has its own one, according to what GDMP requires. This information can be obtained from the Information Index : see for example the query to the CERN Info Index LDAP servers for production and development available at URLs:

http://testbed007.cern.ch/tbstatus-bin/infoindexcern.pl and
http://testbed007.cern.ch/tbstatus-bin/infoindexcerndev.pl,

where the main relevant information concerning the various Computing Elements and Storage Elements (belonging respectively to the production and development EDG testbed) is provided in a formatted way.

To take a direct look at the registered Storage Elements on the GRID information published by the GRID information systems, one can query the MDS LDAP servers hosting the whole hierarchy of MDS GRIS/GIIS information belonging to the various national branches of MDS :
For the production testbed one can browse the content of the GRID top level MDS opening the URL:

For the EDG development testbed one can use the Netscape browser to open the URL:ldap://lxshare0376.cern.ch:2135/mds-vo-name=edg,o=grid??sub?objectclass=*

to get a (quite impressive) view of all information about all branches of MDS in EDG.
For the production testbed the node name is lxshare0225.

More specifically one can query the Replica Catalog on the CERN testbed about the existing registered Storage Elements and the registered files, again opening Netscape at the URL:
ldap://lxshare0226.cern.ch:9011/dc=lxshare0226,dc=cern,dc=ch??sub?objectclass=globusreplicainfo

The LFN (Logical File Name) is currently the very last part of the complete file name entered in the RC: When files are replicated, identical files (replicas) exist at multiple locations and need to be identified uniquely. A set of identical replicas is assigned a *logical filename* (LFN) and each single physical file is assigned a *physical filename* (PFN).

For example, given the LFN *file1.file* , a typical PFN can look like

host.cern.ch/data/run1/day1/file1.file
replicating this file with GDMP creates an identical copy of this file on another SE, creating therefore a new available PFN which also corresponds to the same LFN, like the previous one, like, for example

host38.nikhef.nl/flatfiles/iteam/file1.file

A query to the Replica Catalog to resolve the LFN "file1.file" to its corresponding PFNs will therefore lead as a result

host.cern.ch/data/run1/day1/file1.file
host38.nikhef.nl/flatfiles/iteam/file1.file

NOTE:

Filenames used in these examples are to be taken as example names.
When registering files into Replica Catalogs, LFNs must be **unique**; moreover, you are all sharing the same directories.
Therefore try to use fancy and imagination to use your own filenames in order to avoid conflicts. A good way could be using filenames containing your complete name.

3.1 EXERCISE DM-1 : TRANSFERRING FILES WITH *GLOBUS-URL-COPY*

In this set of examples about the Data Management we will see how to use the available tools to handle data and we start here using *globus-url-copy* , a secure file transfer application making use of
ftp and the *GSI* security framework of Globus to copy files for authenticated users over the GRID.

In this example we issue a grid-proxy-init to get a valid proxy and then we will transfer a file from the CNAF to the CERN Storage Elements, just to start learning how to move files around the GRID.

We will first copy a file (file1.file) from our User Interface to a Storage Element at CERN and then we copy a file from the SE at CERN: lxshare0393.cern.ch. to the SE at CNAF : grid007g.cnaf.infn.it .
To use globus-url-copy, we have to specify the so called Physical File Name (PFN) of both files: The complete path, including the hostname , to both source and destination files, Prefixed by the protocol we plan to use : "file" for files locally available and
"gsiftp" for grid GSI ftp accessed files.

Therefore the commands we are going to issue (after grid-proxy-init) from the User Interface are the following ones:

```
globus-url-copy file://`pwd`/file1.file   \
gsiftp://lxshare0393.cern.ch/flatfiles/SE00/tutor/file1.file
```

(UI to SE-at-CERN copy, - please note "SE00" is SE-zero-zero here)

```
globus-url-copy  \
gsiftp://lxshare0393.cern.ch/flatfiles/SE00/tutor/file1.file \
gsiftp://grid007g.cnaf.infn.it/shared/tutor/file1.file
```

(SE-at-CERN to SE-at-CNAF copy)

Figure 11 shows the basic dynamics of this GRID file transfer using globus-url-copy.

To check the correctness of our file copy, we can issue the following command:

```
globus-job-run grid007g.cnaf.infn.it /bin/ls -la /shared/tutor/
```

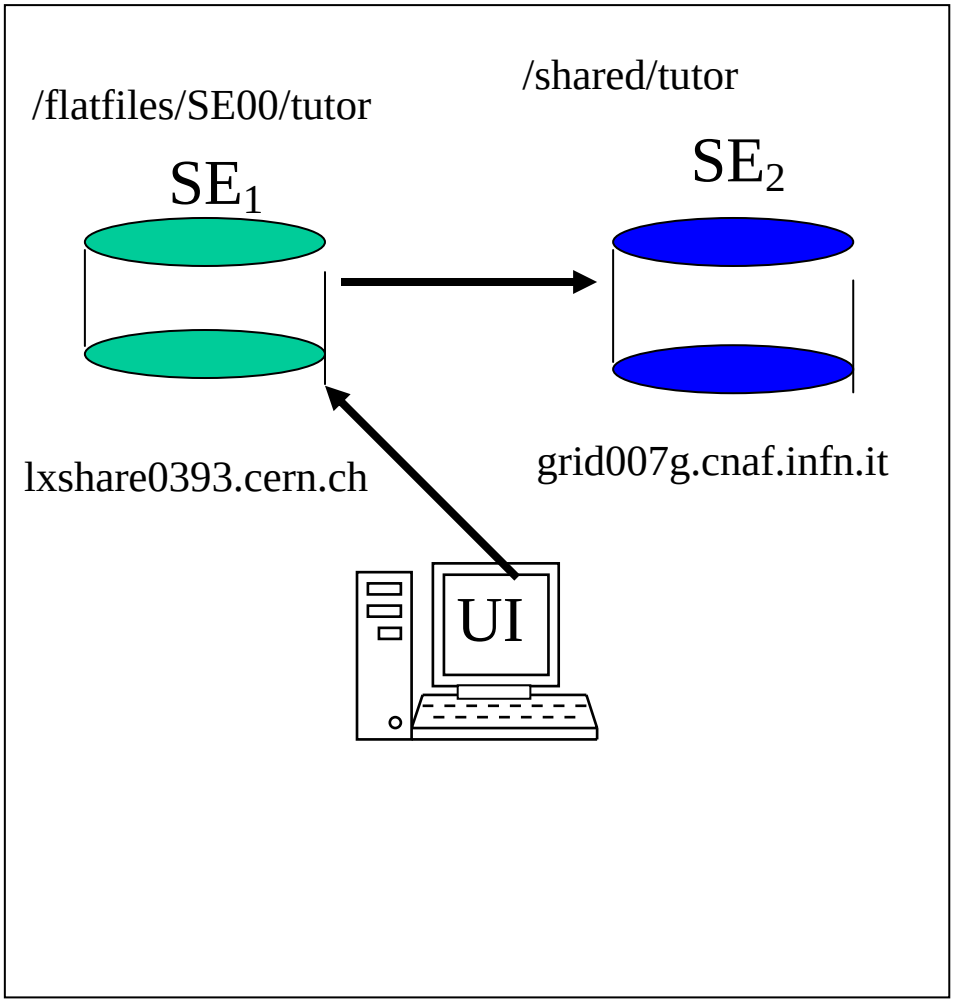


Figure 11: basic globus-url-copy GRID file transfer

3.2 EXERCISE DM-2 : START USING THE EDG REPLICA MANGER

In this example we will be using the EDG Replica Manager and the EDG Replica Catalog interface : we will start using the EDG Replica Manager commands and gain experience with its first basic methods. .(See ref.doc[R4]).
We will use the main defined functions like *getPhysicalFileName* and *copyAndRegisterFile.*
As usual we will be issuing commands from the UI node.

We need to write our *rc.conf* configuration file, and use it to have the EDG RM properly configured for our purposes.

This means essentially to specify the Replica Catalog we need to use and some other relevant information. As an example, the *rc.conf* file for the CERN RC looks like this (*rcCERN.conf*):

```
RC_REP_CAT_MANAGER_DN=cn=RCManager,dc=lxshare0226, dc=cern, dc=ch
RC_REP_CAT_MANAGER_PWD=Testbed1RC
RC_REP_CAT_URL=ldap://lxshare0226.cern.ch:9011/ rc=Testbed1 Replica
Catalog, dc=lxshare0226, dc=cern, dc=ch
RC_LOGICAL_COLLECTION=ldap://lxshare0226.cern.ch:9011/lc=test0,rc=Testbed1
Replica Catalog, dc=lxshare0226, dc=cern, dc=ch
```

After having done that, we will resolve a logical file name (LFN) into its corresponding set of Physical File Names (PFNs), making a query to the Replica Catalog. This is done using the `edg_rc_getPhysicalFileNames` command, which queries the Catalog and resolves a given already registered logical file name in a set corresponding physical file names, i.e., the complete path (including the SE host names) to the locations where that file is available from.

Preliminary to that, we will use the `edg-replica-manager-copyAndRegisterFile` method to copy a file from the UI to the CERN Storage Element, so that we make sure that a physical file (*testRM.file* for CERN and *testRMcnaf.file* for CNAF) will be actually there on the SE, registered inside the Replica Catalog with a corresponding LFN.
In our case therefore the LFN will be testRM.file, destination PFN will be lxshare0393.cern.ch/flatfiles/SE00/tutor/testRM.file.
(source PFN will be UI host name/path: ourHostName.cern.ch/`pwd`/testRM.file)

At our purposes we can use either the main EDG Tutorial RC at NIKHEF or the Replica Catalog at CERN (lxshare0226.cern.ch); in all cases we need to specify an access password and the name of an available existing logical collection by writing them in the rc.conf file we are going to use.

To register the file at CERN we need to issue the following commands:

```
edg-replica-manager-copyAndRegisterFile -l testRM.file  -s       \
`hostname`/`pwd`/testRM.file -d lxshare0393.cern.ch/flatfiles/SE00/tutor/testRM.file  -c
rcCERN.conf

globus-job-run lxshare0393.cern.ch /bin/ls -la /flatfiles/SE00/tutor/

 edg_rc_getPhysicalFileNames -l testRM.file  -c rcCERN.conf
```

To do the same thing at NIKHEF :

```
edg-replica-manager-copyAndRegisterFile -l testRMnikhef.file  -s \
`hostname`/`pwd`/testRMnikhef.file    \
-d grid007g.cnaf.infn.it/shared/tutor/testRMnikhef.file  -c  rcNIKHEF.conf

globus-job-run grid007g.cnaf.infn.it /bin/ls -la /shared/tutor/

edg_rc_getPhysicalFileNames -l testRMnikhef.file  -c rcNIKHEF.conf
```

NOTE THAT LFNS HAVE TO BE UNIQUE INSIDE REPLICA CATALOGUES. THEREFORE TAKE THE NAMES USED HERE ONLY AS EXAMPLE NAMES. USE YOUR OWN ONES.

3.3 EXERCISE DM-3 : A QUERY TO THE NIKHEF, CERN AND CNAF REPLICA CATALOGS

This is a simple Data Management exercise : we will query the Replica Catalogs of CERN, NIKHEF and CNAF, in order to see which are the registered Logical File Names and the corresponding Physical File Names of existing registered files.
We want to make a query to the LDAP servers hosting the Replica Catalogs to have a snapshot of their content

3.3.1 Simple Query Using the RC Command line Tool

An easy way to query the Replica Catalog for location of files is using the RC Command Line Interface (edg_rc-*). For instance, we want to find all locations for a given LFN.

```
edg_rc_getPhysicalFileNames -l testfile1 –c /opt/edg/etc/tutor/rc.conf
configuration file:     /opt/edg/etc/tutor/rc.conf
logical file name:      testfile1
gppse05.gridpp.rl.ac.uk/flatfiles/05/tutor/testfile1
lxshare0393.cern.ch/flatfiles/SE00/tutor/testfile1
```

3.3.2 Advanced Query Using LDAP

We will query both CERN and CNAF Replica Catalogs, using the *ldapsearch* command.

The Replica Catalog is an LDAP server for which object classes are defined.
Classes are organized in a hierarchical way, so that they normally derive one from the other, like in a classical object oriented environment. Base class is a class called *GlobusReplicaCatalog*, from which
others are derived.
Logical Collections are required to host the directory tree of the filenames inside the Catalog.
We will issue an ldapsearch command using openLDAP: no JDL file will be required and we will issue commands from the UI.

To perform the queries we have to issue:
Query the RC at NIKHEF :

```
ldapsearch -L -b "lc=EDGtutorial WP1 Repcat,rc=EDgtutorialReplicaCatalog, dc=eu,
dc=datagrid, dc=org" -H "ldap://grid-vo.nikhef.nl:10389" –x  -P 2
```

Query the RC at CERN:

```
ldapsearch -L -b "rc=Testbed1 Replica Catalog, dc=lxshare0226, dc=cern,  \
dc=ch" –H  "ldap://lxshare0226.cern.ch:9011" -x -P 2
```

Query the RC at CNAF:

```
ldapsearch -h grid005f.cnaf.infn.it -p 9011 -b "lc=EDG tutorials collection,rc=EDG
Tutorials Replica Catalog,dc=grid005f, dc=cnaf, dc=infn, dc=it"
'(ObjectClass=*)' -x -P2
```

or

```
ldapsearch -h grid005f.cnaf.infn.it -p 9011 -b "dc=grid005f, dc=cnaf,          \
dc=infn, dc=it" '(ObjectClass=*)' -x -P2
```

We can see the result of the query all registered LFN and PFN corresponding to the selected Logical Collections.(test0 and INFN Test Collection)
We can issue also from the Netscape Browser , requiring in the URL location bar:

For the CERN RC:

```
ldap://lxshare0226.cern.ch:9011/lc=test0,rc=Testbed1 Replica Catalog,      \
dc=lxshare0226, dc=cern, dc=ch
```

and for the CNAF RC:

```
ldap://grid005f.cnaf.infn.it:9011/lc=EDG tutorials collection,rc=EDG Tutorials Replica
Catalog,dc=grid005f,dc=cnaf, dc=infn, dc=it
```

3.4 EXERCISE DM-4 : GDMP BASICS

In this example, we will get familiar with the first basic commands of GDMP: `gdmp_ping`, `gdmp_host_subscribe`, `gdmp_register_local_file`, `gdmp_publish_catalog`.
Please refer to the online GDMP documentation available from
http://project-gdmp.web.cern.ch/project-gdmp/documentation.html.
In particular take a look at the description of the mechanism of GDMP at
http://cmsdoc.cern.ch/cms/grid/userguide/gdmp-3-0/node37.html. .(See ref.doc[R3]).

GDMP works through GDMP servers running locally on the Storage Elements belonging to the various GRID sites. It works through subscription of a site A to various other different sites (B,C,…), whose content we declare to be interested in, issuing a `gdmp_host_subscribe` to them.
We get a valid proxy by issuing the usual `grid-proxy-init`. Then we will first check that we can correctly `gdmp_ping` the SE we are interested in (in this case the CERN Storage Element lxshare0393.cern.ch.).
Then we will register a dummy test file (*testfile1.txt*) in the local file catalogue of the machine.
GDMP works by means of a distributed client-server architecture, therefore we can issue all relevant commands from the User Interface node : there's no need to necessarily submit jobs, no need for JDL files, even though this is of course always possible.
Preliminary to any GDMP operation, we need both to issue a `grid-proxy-init` and to export the value of the GDMP_CONFIG_FILE variable, to tell GDMP which is the configuration set up we need to use.
Only to show how the command works, we issue here also a `gdmp_host_subscribe`.
(We will use the subscription itself in the next examples DM-5 and DM-6 to get replicas.) :
For the moment we just want to register a file in the local file catalog of the GDMP server running on a CERN SE and publish the catalog of files on that SE.
(We can already however say that all sites (SEs) that have already subscribed to CERN will have their import catalogs of files from CERN updated, once we have published a new entry in the CERN Catalog). See Figure 12.
Usually for each defined VO there is an already available configuration file for GDMP under /opt/edg/etc/<VO-name>/gdmp.conf. In our case the VO we are going to use is the one called *tutor*, explicitly set up for the EDG students.

```
export GDMP_CONFIG_FILE=/opt/edg/etc/tutor/gdmp.conf

gdmp_ping -S lxshare0393.cern.ch -p 2000 -

gdmp_ping -S grid007g.cnaf.infn.it -p 2000

gdmp_host_subscribe -r lxshare0393.cern.ch -S \
        grid007g.cnaf.infn.it

globus-url-copy file://`hostname``pwd`/testfile1.txt \
gsiftp://lxshare0393.cern.ch/flatfiles/SE00/tutor/testfile1.txt
```

```
gdmp_register_local_file -S lxshare0393.cern.ch -P 2000 -R -p  \
                /flatfiles/SE00/tutor/testfile1.txt  -V tutor
gdmp_publish_catalogue -S lxshare0393.cern.ch -P 2000 -V tutor  -C
```

Here "–S" specifies the GDMP server we want to submit jobs to (default is read from the file gdmp.shared.conf) from the UI machine where we are working from.
"– r" in the *gdmp_host_subscribe* command stands for the remote GDMP server we want to specify (we want our GDMP server to subscribe to). –R specifies that the file is local to the GDMP server we are considering.
To locally register all files in our VO directory we could have issued also :

```
gdmp_register_local_file -R -d /flatfiles/SE00/tutor  \
                -S lxshare0393.cern.ch -P 2000 -V tutor
```

Of course, before trying to register a file in the local catalog on a given SE, we have to make sure the file is there : if file *testfile1.txt* is not already there on /flatfiles/SE00/tutor on the CERN SE, we need to copy it there using the globus-url-copy command (see previous example).
Issuing these commands we have therefore in conclusion published a new entry in the local catalog of a CERN SE : we can verify what we have done taking a look at the catalog, using globus-job-run (on every SE there is also a Globus gatekeeper component running) to have a look at the file on that machine or, much better, using an ad hoc GDMP command:

```
gdmp_job_status -S lxshare0393.cern.ch  -c  local_file_catalogue
```

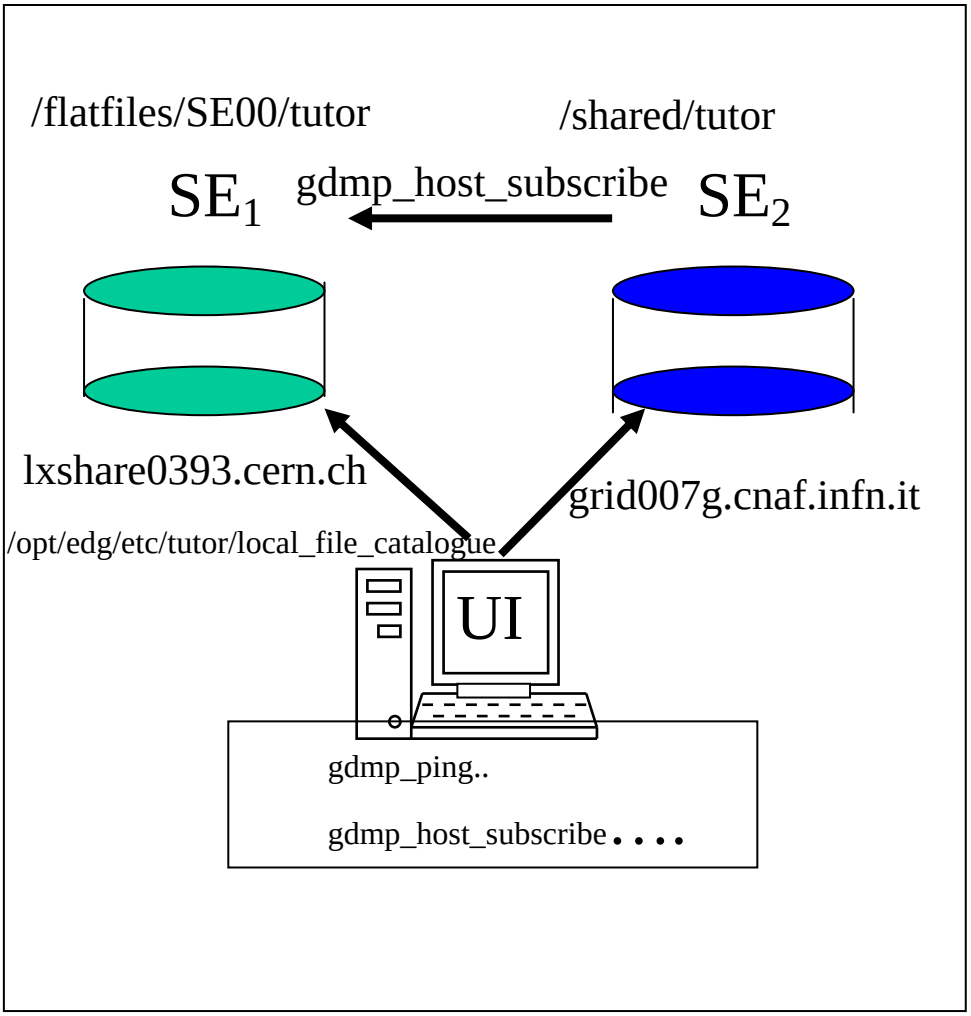


Figure 12 : GDMP server on SE2 subscribes to SE1 and SE1 publishes a new local catalog : we issue all commands from our UI node.

3.5 EXERCISE DM-5 : GDMP ADVANCED(1): FILE REPLICATION

In this exercise (DM-5, GDMP advanced) we will perform the CNAF to CERN replication of a newly created test file.
We will make a replica from the CERN SE to the CNAF SE of a test file using GDMP : we will perform the following operations: (issuing all commands from our UI machine)

-1. Verify the install and correctness of the connection using gdmp_ping from the UI to both the gdmp servers at CNAF and CERN.
(if everything is fine we should get as answer:
The local GDMP server grid007g.cnaf.infn.it:2000 is listening and you are an authorized user).

-2. Make the CERN GDMP server subscribe to the CNAF GDMP server (so that any new file appearing in the CNAF export catalog will become a new entry in our CERN local import catalog).

-3. Copy a new file in the CNAF SE published SE mount point;

-4. make CNAF GDMP server register the new created file;

-5. publish the catalog at CNAF, ask the CNAF GDMP server to do that;

-6. get a copy from our GDMP server running on our SE - at CERN –issuing a gdmp_replicate_get:

Now everything is ready: a new file is there at CNAF, we have registered and published (at CNAF), so that our import catalog at CERN is now filled with the new entry : we just need to issue a gdmp_replicate_get at CERN (with -R grid007g.cnaf.infn.it/"mountpoint" as input argument) to get a copy of that file.
After having performed these operations we will have therefore replicated a test file from CNAF to CERN having used GDMP.

As usual we start by pinging the two hosts we want to work with:

```
gdmp_ping -S lxshare0393.cern.ch

gdmp_ping -S grid007g.cnaf.infn.it
```

We then make CERN subscribe to CNAF issuing a gdmp_host_subscribe command:

```
gdmp_host_subscribe -S lxshare0393.cern.ch  -r grid007g.cnaf.infn.it
```

We copy the file from our UI machine to the CNAF Storage Element:

```
globus-url-copy  file://`pwd`/testfile.test  \
gsiftp://testbed007.cnaf.infn/shared/tutor/testfile.test
```

We register locally on the local catalog of the CNAF SE:

```
gdmp_register_local_file -S grid007g.cnaf.infn.it -P 2000 -R -p \
   /shared/tutor/testfile.test -V tutor
```

We can check the status of the GDMP job registering the file issuing :

```
gdmp_job_status  -S grid007g.cnaf.infn.it –L GDMPJobId
```

(We get the GDMP JobId from the system, after having issued the registration command). We then need to make the CNAF SE publish its file new file catalog,
creating a new export catalog starting from the (updated) local catalog:

```
gdmp_publish_catalogue -S grid007g.cnaf.infn.it -P 2000 -V tutor -C
```

Finally the job is done issuing the replicate_get command:

```
gdmp_replicate_get -S lxshare0393.cern.ch -P 2000 -V tutor -C
```

We can of course verify that the file has been finally correctly copied, as usual, issuing a globus-job-run to the CERN SE:

```
globus-job-run    lxshare0393.cern.ch    /bin/ls -la   /flatfiles/SE00/tutor
```

As a related very similar problem try to perform to the same operations using the NIKHEF SE instead of CNAF one.
Get information about the mount points and supported VOs from the II:

http://testbed007.cern.ch/tbstatus-bin/infoindexcern.pl
http://testbed007.cern.ch/tbstatus-bin/infoindexcerndev.pl

You can in principle use either URL, although it is recommended that you use the production testbed running the EDG 1.2.0 release instead of the development one for your exercises.

3.6 EXERCISE DM-6: GDMP ADVANCED(2) -MORE ON REPLICATING FILES WITH GDMP

Goal of this exercise is to experience file replication from a GRID Storage Element to another SE, using the GDMP provided functions at this purpose, to gain further experience with GDMP. Although this exercise is rather similar to the previous one, we will get further experience with replicating files through GDMP and we will describe things a little more in detail.

As we have already said, GDMP uses import and export catalogues to store a list of files which have to be imported (through their GDMP server running) from other Storage Elements [import] and which are available for other Storage Elements to be copied [export].
In the following we assume the main replication mechanism of GDMP is clear:
GDMP works through subscription (`gdmp_host_subscribe`) of a given site A to a certain number of sites (B,C..)
in which site A declares to be interested : whatever "new" will happen on these sites, site A will be informed. (a site is here a Storage Element, running the GDMP server).
Note that the import catalogs are normally created on a given site A by site B's GDMP server whenever site B publishes its catalog of export files.
There is a set of hosts to which site A is subscribed : if any of these hosts publishes new files in its export catalog, the GDMP server will automatically modify the import catalog of site A, so that finally the issuing of `gdmp_replicate_get` will perform the replication and site A will get all the new files.
Every single file needs to be registered first before it can be replicated : at this purpose `gdmp_register_local_file` needs to be used.
Therefore to correctly replicate a file from site A to site B, one needs first to register locally the file on site A (of course make sure the files is already there or we copy it there)(`gdmp_register_local_file`), then to publish the catalog on site A (`gdmp_publish_catalog`), finally to get a copy of the file from site A on site B issuing a `gdmp_replicate_get` command [on site B].

We will proceed in this way: Site A is for us CNAF, site B is CERN. We want to replicate from A to B (working from the UI machine : issuing commands from the UI). After grid-proxy-init, we first copy (by hand, using `globus-url-copy`) a file to CNAF on the published mount point for our virtual organization (it will be the published mount point by the http://testbed007.cern.ch/tbstatus-bin/infoindexcern.pl GRID info systems + "VO-name", where in our case "VO-name" is “tutor”); we then check that our configuration file is correct so that a corresponding replica catalog to be used is defined and we make sure that we have exported the GDMP_CONFIG_FILE variable pointing at the right gdmp.conf file (either the default */opt/edg/etc/tutor/gdmp.conf* or our own one).
We can issue a `globus-job-run` to the SE's gatekeeper to check if the file has been correctly copied there.

We then make the CERN GDMP server subscribe to the CNAF one.
After this, we make the CNAF GDMP server locally register the file and publish its new export catalog;
Finally we issue a gdmp_replicate_get to get a copy of the files.

Similarly to the previous exercise, the command we are going to issue are the following ones: the preliminary commands are :

```
grid-proxy-init

export GDMP_CONFIG_FILE=/opt/edg/etc/tutor/gdmp.conf

gdmp_host_subscribe -r grid007g.cnaf.infn.it  -P 2000 -S lxshare0393.cern.ch -p 2000
```

We then copy out test file (ourSimpleTestFile.txt) to the SE at CNAF and verify the copying via globus-job-run:

```
globus-url-copy file://`pwd`/ourSimpleTestFile.txt  \
gsiftp://grid007g.cnaf.infn.it/shared/tutor/ourSimpleTestFile.txt

globus-job-run grid007g.cnaf.infn.it   /bin/ls –la  /shared/tutor/

gdmp_register_local_file -S grid007g.cnaf.infn.it  -P 2000 -R –p          \
/shared/tutor/ourSimpleTestFile.txt -V tutor

 gdmp_job_status –S grid007g.cnaf.infn.it –L GDMP-JOB-ID  -V tutor
                      (where we get the GDMP-JOB-ID from the system)

 gdmp_publish_catalog -S grid007g.cnaf.infn.it -P 2000 -V tutor  -C

 gdmp_replicate_get -S lxshare0393.cern.ch    -P 2000 -V tutor -C
```

Again, as a related similar problem, as an exercise,
try to do the same thing from a Worker Node - in a shell script. Don't forget to include all required files in the InputSanbox, including the test file to be copied on the Storage Element from the Worker Node.
As GDMP 3.0 reference, again, use the HTML GDMP guide on the Web:
http://cmsdoc.cern.ch/cms/grid/userguide/gdmp-3-0/gdmp-3-0.html.

3.7 EXERCISE DM-7 : USING THE EDG REPLICA MANAGER WITHIN A JOB.

In this exercise we are going to use the edg-replica-manager inside a Job, to copy a file from the Worker Node, where it is created, to a Storage Element, and register the file inside a Replica Catalog. The file is postscript file created by PAW on the worker node.(see also exercise JS-3).

Contrary to the previous examples on Data Management, this time we need to write a JDL file with the Job Description of the Job we want to submit to the system : we create a JDL file which runs PAW on a worker node of a given computing element, copies and registers this file using the edg-replica-manager-copyAndRegisterFile method, and we finally check that the produced file is correctly registered inside the Replica Catalog. Namely, the JDL we are going to use is the following one :

```
Executable = "/bin/sh";
Arguments = "edgRM.sh testgrid.ps";
InputSandbox = {"edgRM.sh", "pawlogon.kumac", "testgridnew.kumac",
"paw.metafile", "rcCERN.conf", “rcNIKHEF.conf” };
OutputSandbox = {"stderror.log", "StdOutput.log", "testgrid.ps"};
Stderror = "stderror.log";
StdOutput = "StdOutput.log";
```

The issued commands sequence we have to use is the following one :

```
grid-proxy-init

dg-job-submit –resource testbed001.cnaf.infn.it:2119/jobmanager-pbs-medium \
edgRM.jdl

dg-job-status JobId

dg-job-get-output JobId
```

To check the actual presence of the file on the SE:

```
globus-job-run grid007g.cnaf.infn.it /bin/ls -la /shared/tutor/
```

To check the registration inside the RC we make an LDAP query search:

```
ldapsearch -L -b "lc=EDGtutorial WP1 Repcat,rc=EDgtutorialReplicaCatalog, dc=eu,
dc=datagrid, dc=org" -H "ldap://grid-vo.nikhef.nl:10389" –x –P2
```

Then, to query the CERN Replica Catalog, we issue:

```
ldapsearch -L -b "lc=test0,rc=Testbed1 Replica Catalog, dc=lxshare0226, dc=cern,
dc=ch" -H "ldap://lxshare0226.cern.ch:9011" -x -P 2
```

As a related problem , to gain experience with RM inside jobs,
try to do the same exercise involving the Storage Elements at RAL and LYON,
and the CERN Replica Catalog (lxshare0226).

Make an LDAP query to the information index at CERN to extract information
about the mounting points to be used on these SEs .
See also the two URLs :
http://testbed007.cern.ch/tbstatus-bin/infoindexcern.pl
http://testbed007.cern.ch/tbstatus-bin/infoindexcerndev.pl

As previously suggested, it is recommended that you use the first one, i.e. the EDG production EDG 1.2.0 testbed distributed cluster.

3.8 EXERCISE DM-8 : A DATA-ACCESSING JOB (1) : A PERL SCRIPT

In this example we will submit a data accessing job, and more precisely a job which will use a file locally available from the Worker Node, because of close NFS mounting of a repository directory on the Storage Element. We want namely to print out the content of a file, whose LFN is
know, and is specified in the JDL.
We want to show here how to use all the required tools for and End User, to be able to write applications that, irrespective of the resulting destination Computing Element they end up to be executed on, are able to access all required data using the EDG provided tools.
We will make use of a simple PERL script (*LFN2INFO.pl*) to be executed on the worker node, which
parses the *.BrokerInfo* file (see also exercise JS-11) in order to get relevant information for the job
and in particular prints out the best physical file name (for that LFN), the data access protocol to be
used, the path to the file and finally its content.
We start from the following JDL file (*dataAcessPerl.jdl*) :

```
Executable = "Prova3.sh";
StdOutput = "sim.out";
StdError = "sim.err";
InputData = {"LF:tutorialEDG.txt"};
ReplicaCatalog = " ldap://grid-vo.nikhef.nl:10389/ lc=EDGtutorial WP1          \
Repcat,rc=EDgtutorialReplicaCatalog, dc=eu, dc=datagrid, dc=org“ ;
DataAccessProtocol = {"file", "gridftp"};
Rank = -other.EstimatedTraversalTime;
InputSandbox  = {"Prova3.sh", "LFN2INFO.pl"};
OutputSandbox = {"sim.out", "sim.err", "sim.search", "BrokerInfo"};
```

The PERL script we are using basically parses the *.BrokerInfo* file and gets information on which
protocol to use in order to be able to retrieve the required data in Input.
In this case we require the Logical File Name *tutorialEDG.txt* , and , during the matchmaking process,
the RB queries for the corresponding Physical File Names the RC on the CERN testbed : lxshare0226.cern.ch. This is specified directly in the Replica Catalog classAd statement inside the JDL
file. In the JDL we also specify which are the acceptable protocols for our application : in this case we
have specified that we can access files either directly (SE local to the WN: a network file system is
available) , or we can also use gridFTP (`globus-url-copy`) to copy the file from the SE and then
access it locally on the WN.

In general we have therefore to "gridify" our applications to cope with the fact that we may not know
a-priori which protocol (file, rfio, gridftp, among the one we allow in the JDL) we will actually have
to use to access the data and from which SE we will get them. In this case there's an "if-then" control
in the PERL script which decides what to do according to what is written inside the *.BrokerInfo* file.
In general we have available – apart from this specific example (PERL script) -the following tools or "working options" to handle Data Management in EDG :

- 1) We know a-priori exactly what is going to happen since we force the execution of the Job on a given Computing Element and we know which are its "closeSEs" (ldap://lxshare0376.cern.ch:2135/mds-vo-name=edg,o=grid??sub?objectclass=*)
(same for lxshare0225) to see, for each SE, which are the closeCEs, for example, or http://testbed007.cern.ch/tbstatus-bin/infoindexcern.pl, (for each SE see its closeCEs).

- 2) We use the RC C++ API inside our application to perform again a query to the RC (repeating therefore what has already happened in the matchmaking process), and given the LFNs we want to use, we use the available methods like getPhysicalFileNames(LFN) to extract the available PFNs to be accessed)

- 3) Use the BrokerInfo command line interface (edg-brokerinfo) commands on the Worker Node, or the RCb C++ API inside the application, or a "manual" parsing of the *.BrokerInfo* file – i.e. similar ways all based on the matchmaking results stored in that file again to find which are the PFNs we want to access to get our data from the WN.

The submission, the matchmaking process, data access and output retrieval are represented in Figure 13

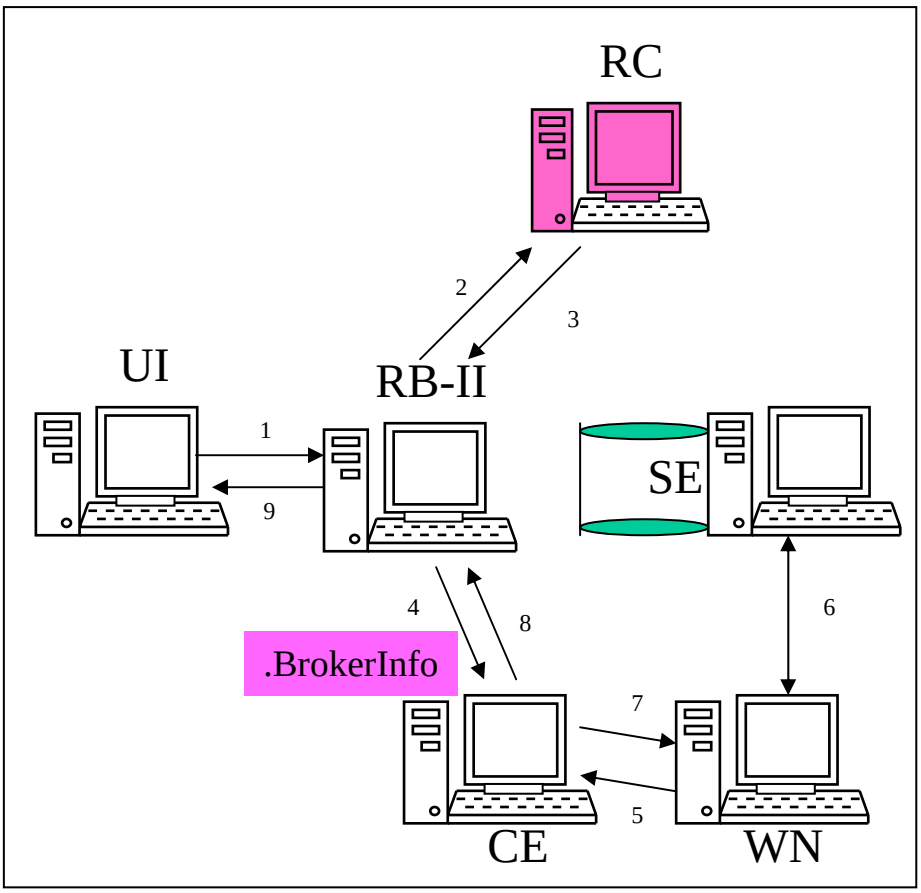


Figure 13 : Matchmaking, .BrokerInfo and Data Access for Example DM-8

3.9 EXERCISE DM-9 A DATA ACCESSING JOB (2): DIRECTLY USING PFN

Goal of this exercise is to show how to access data directly specifying the Physical File Name (PFN)
we want to use in the JDL file – we want here to print out the content of a file whose location (SE) is
known a priori.
Specifying directly a PFN (and not a LFN) in the JDL file makes sure that in the matchmaking
process only the query to the RC is skept : the Info Index is queried to make sure that both the CE-CloseSE relationship is cross-checked before selecting the destination CE and that all remaining requirements are considered.
At this purpose we will start by the following *(pfnDataAccss.jdl*) JDL file:

```
Executable = "Prova1.sh";
StdOutput = "sim.out";
StdError = "sim.err";
InputData = {"PF:tbn03.nikhef.nl/flatfiles/tutor/pippo.txt"};
DataAccessProtocol = {"gridftp", "file"};
Rank = -other.EstimatedTraversalTime;
InputSandbox  = {"Prova1.sh"};
OutputSandbox = {"sim.out","sim.err", ".BrokerInfo"};
```

The shell script here (*Prova1.sh*) executes a `globus-url-copy` from the SE to the local working WN
directory and does a simple `/bin/more` of the *pippo.txt* file.
As we already said, in this case the job assumes that the PFN JDL is known before job ‘s execution
and it is specified directly in the JDL file through the statement
InputData = {"PF:tbn03.nikhef.nl/flatfiles/tutor/pippo.txt"};

In this case we want to retrieve the output on an ad-hoc created local directory on the UI:
`mkdir exampleDM9`

The issued command sequence for this example will be:

```
grid-proxy-init
dg-job-submit pfnDataAccess.jdl
dg-job-status JobId
dg-job-get-output –d exampleDM9 JobId
```

The GRID workflow for this exercise is shown in Figure 14.

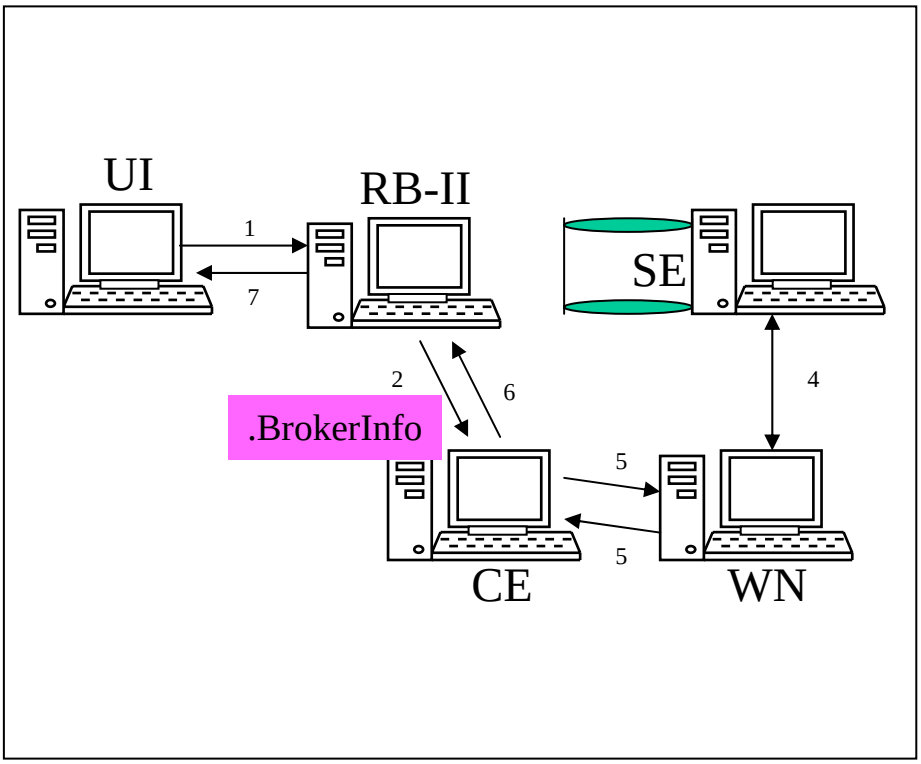


Figure 14 : data workflow for exercise DM-9

4. INFORMATION SYSTEMS EXERCISES

Information Systems provide to the GRID an updated view of its resources, to allow their management and effective usage by users and internal GRID subcomponents. (See ref.doc[R5]).
The presently implemented Information System in EDG 1.2.0 is based on the Globus MDS 2.1 (Metacomputing Directory Service – now called Monitoring and Discovery Service), which is a directory service based on the LDAP (Lightweight Directory Access Protocol). Directory Services are read-access optimized data bases; they are intended for systems with frequent read access rather than frequent write access. In Directory Services the complexity due to supporting transactions operation mode is avoided. LDAP is based on four models dealing with *Information, Naming, Functional and Security*. Data are organized in *Object Classes*, in a hierarchical object model, and all classes are derived from the *top* class.
The LDAP server can be queried specifying the particular class of objects to be returned (using filters) and specifying the starting node – in the directory structure – from which we want information to be retrieved. This is done by specifying the DN (*Distinguished Name*) of the starting node for the needed information, this is the *base DN* parameter of a LDAP search. Entries have attributes, whose name can be specified in the query string.
To describe in detail LDAP is out of the scope of this Tutorials. We want to show here just a few examples on how to get information from the GRID Information Systems. A good reference for understanding the LDAP mechanism is given by [R1].
In EDG all relevant resources run a LDAP daemon called *slapd*. As a convention Globus MDS uses port 2135.
To summarize the Information System in within EDG we can say the following:

EDG currently uses Globus MDS which is built on OpenLDAP. The Lightweight Directory Assess Protocol (LDAP) offers a hierarchical view of information in which the schema describes the attributes and the types of the attributes associated with data objects. The objects are then arranged in a Directory Information Tree (DIT).

A number of information providers have been produced by EDG. These are scripts which when invoked by the LDAP server make available the desired information. The EDG information providers include Site Information, Computing Element, Storage Element and Network Monitoring scripts.

Within MDS the EDG information providers are invoked by a local LDAP server, the Grid Resource Information Server (GRIS). "Aggregate directories", Grid Information Index Servers (GIIS), are then used to group resources. The GRISs use soft state registration to register with one or more GIISs. The GIIS can then act as a single point of contact for a number of resources, i.e. a GIIS may represent all the resources at a site. In turn a GIIS may register with another GIIS, in which case the higher level GIIS may represent a country or a virtual organisation. Within EDG we have configured MDS so that we have a hierarchy of sites, and then countries registered to the top level EDG GIIS.

As MDS is based on LDAP, queries can be posed to the current Information and Monitoring Service using LDAP search commands. An LDAP search consists of the following components:

```
$ldapsearch \
 -x \
 -LLL \
 -H ldap://lxshare0225.cern.ch:2135 \
 -b 'Mds-Vo-name=datagrid,o=grid' \
 'objectclass=ComputingElment' \
 CEId FreeCPUs \
 -s base|one|sub
```

Explanation of fields:

`-x` “simple” authentication
`-LLL` print output without comments
`-H ldap://lxshare0225.cern.ch:2135` uniform resource identifier
`-b 'Mds-Vo-name=datagrid,o=grid'` base distinguished name for search
`'objectclass=ComputingElment'` filter
`CEId FreeCPUs` attributes to be returned
`-s base` scope of the search specifying just the base object, one-level or the complete subtree

The top level GIIS MDS node is currently at CERN on node **lxshare0225.cern.ch** for the EDG production testbed and on node **lxshare0376.cern.ch** for the EDG development testbed.

4.1 EXERCISE IS-1 : DISCOVER WHICH SITES ARE AVAILABLE ON THE TESTBED

Query the top level of the information and monitoring system to discover which sites are available on the testbed.

```
ldapsearch -x -LLL -H ldap://lxshare0376.cern.ch:2135  \
-b 'Mds-Vo-name=edg,o=grid'  \
'objectclass=SiteInfo' siteName
```

4.2 EXERCISE IS-2 : DISCOVER THE AVAILABLE GRID RESOURCES

In this example exercise we perform a query to the top-level MDS hierarchy node of the development testbed at CERN to discover all available resources.

```
ldapsearch -x -LLL -H ldap://lxshare0376.cern.ch:2135  \
–b 'Mds-Vo-name=edg,o=grid'  \
"(|(objectclass=ComputingElement)(objectclass=StorageElement))" CEId SEId
```

Try to query in a similar way the production testbed top level MDS GRIS node lxshare0225.

4.3 EXERCISE IS-3 : EMULATE THE RESOURCE BROKER

In this example we perform some basic selection for the job, i.e. we emulate the Resource Broker while performing the matchmaking process.

```
ldapsearch -x -LLL -H ldap://lxshare0376.cern.ch:2135  \
–b 'Mds-Vo-name=edg,o=grid'  \
"(&(objectclass=ComputingElement)(RunTimeEnvironment="CMS-1.1.0")(TotalCPUs>  \
 =2))" CEId TotalCPUs
```

As an exercise, try to change the used ClassAds in the query and perform the query itself also on the production MDS GIIS node.

4.4 EXERCISE IS-4 : FIND OUT WHICH ARE THE CLOSE SEs

Now find the Storage Elements that are close to a Computing Element of your choice (substitute XXXX with a *CEId* obtained from the previous search).

```
ldapsearch -x -LLL -H ldap://lxshare0225.cern.ch:2135  \
-b 'Mds-Vo-name=edg,o=grid'  \
```

```
'(&(objectclass=CloseStorageElement)(CEId=XXXX))' CloseSE
```

4.5 EXERCISE IS-5 : FREE SPACE ON THE STORAGE ELEMENT

Next find out how much free space the Storage Element has. You will need to substitute XXXX with a *SEId*, use the value you obtained for *CloseSE* in the previous search.

```
ldapsearch -x -LLL -H ldap://lxshare0225.cern.ch:2135 \
-b 'Mds-Vo-name=edg,o=grid' \
'(&(objectclass=StorageElementStatus)(SEId=XXXX))' Sefreespace
```

4.6 EXERCISE IS-6 : QUERY A GRIS ON THE CE AT RAL

Now query the chosen Computing Element again, but this time query the resource directly. We will be querying a GRIS (at RAL) rather than a GIIS so we use a base dn of *Mds-Vo-name=local,o=grid*. The URI has to be changed to use the host name component of the *CEId*, in this example we have used *gppce06.gridpp.rl.ac.uk*, remember to replace XXXX with the selected *CEId*.

```
ldapsearch -x -LLL -H ldap://gppce06.gridpp.rl.ac.uk:2135 \
-b 'Mds-Vo-name=local,o=grid' .\
'(&(objectclass=ComputingElement)(CEId=XXXX))' FreeCPUs
```

4.7 EXERCISE IS-7 : INFORMATION ABOUT EDG RUNNING JOBS

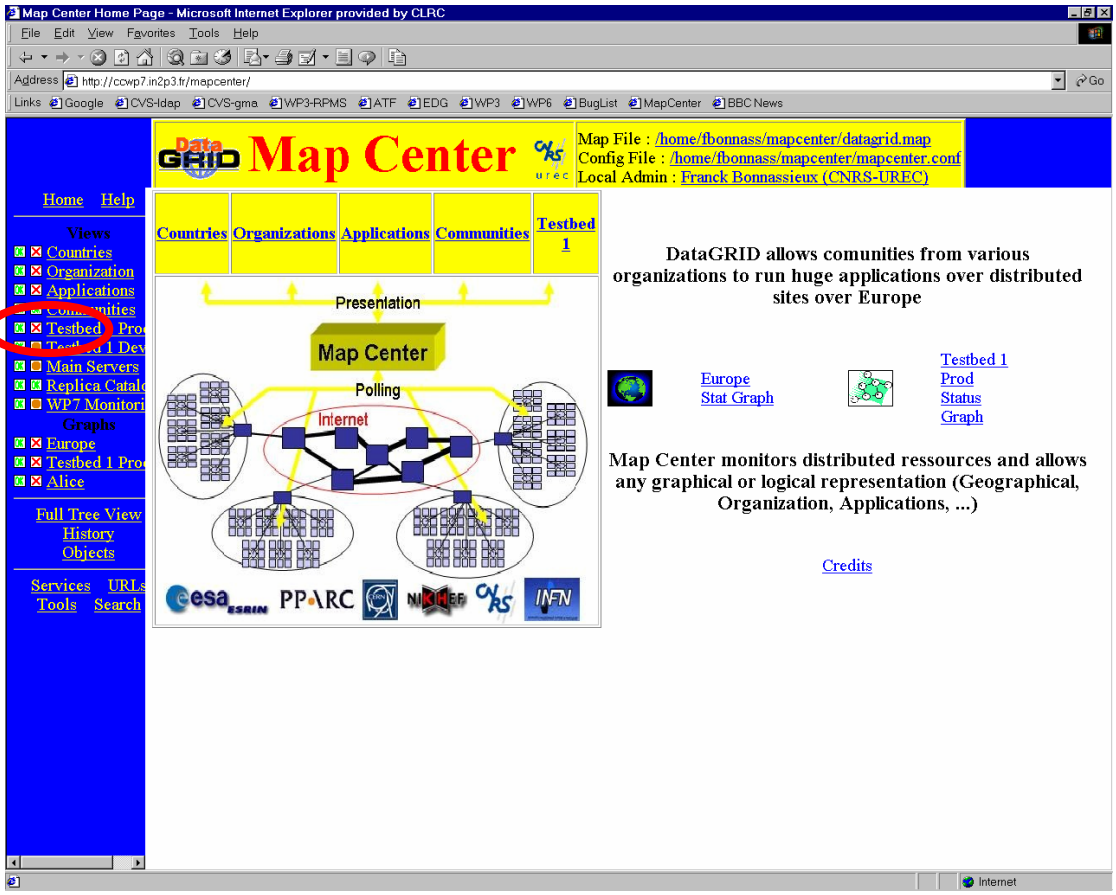
After job submission we can check the status of the running jobs all over the GRID making a query to the MDS top level LDAP server.

```
ldapsearch -x -LLL -H ldap://lxshare0376.cern.ch:2135 \
–b 'Mds-Vo-name=edg,o=grid' \
"(objectclass=ComputingElement)" CEId TotalJobs RunningJobs IdleJobs
```

4.8 EXERCISE IS-8 : MAP CENTRE

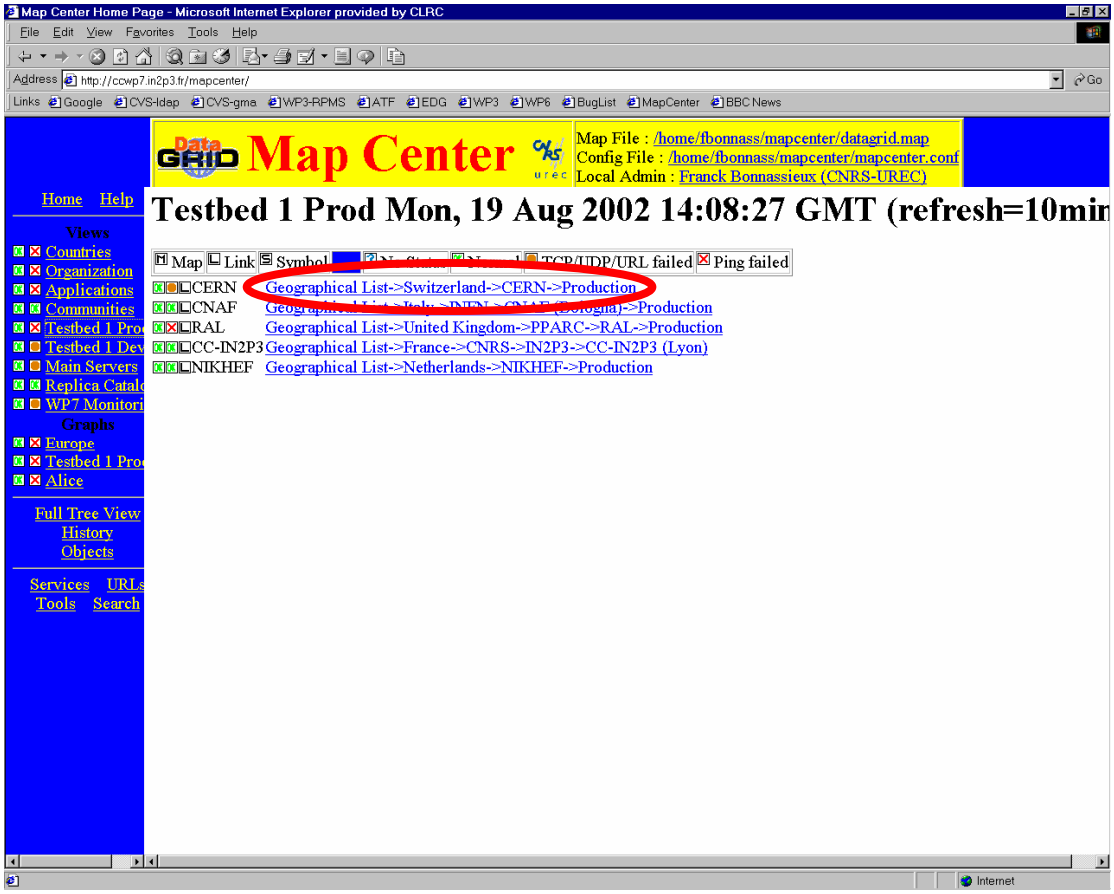
Map Centre provides a web interface to the information and monitoring system. IT makes use of LDAP searches to obtain the data.
http://ccwp7.in2p3.fr/mapcenter/

From the front page of Map Centre you can select resources from a tested or from a country



From the front page of Map Centre select

`Testbed1 Prod`



Selecting `Testbed1 Prod` will then present you with the countries registered in Testbed1

`Select a country`

You will then be taken to a point in a list of resources on the grid representing the country you have selected
From here you can either select to view the resources available to a county or view the resources on a specific machine by clicking on an instance of "mds".

`Select a countries "mds"`

This will then issue an LDAP search directed at the selected country GIIS. A list of sites registered with the GIIS will be returned.

`Select a site`

Now take some time to browse the information that is published about a site and its Computing and Storage Elements.

5. APPENDIX A : REQUIRED CONFIG FILES FOR DATA MANAGEMENT

5.1 REPLICA CATALOGS AND GDMP CONFIGURATION FILES:

5.1.1. NIKHEF – EDG Tutorials VO main Replica Catalog (rcNIKHEF.conf)

RC_REP_CAT_MANAGER_DN=cn=RCManager
RC_REP_CAT_MANAGER_PWD=EDGtutorial
RC_REP_CAT_URL=ldap://grid-vo.nikhef.nl:10389/rc=EDGtutorialReplicaCatalog,dc=eu-datagrid,dc=org
RC_LOGICAL_COLLECTION=ldap://grid-vo.nikhef.nl:10389/lc=EDGtutorial WP1 Repcat, \
rc=EDGtutorialReplicaCatalog,dc=eu-datagrid,dc=org

5.1.2. CERN (rcCERN.conf)

RC_REP_CAT_MANAGER_DN=cn=RCManager,dc=lxshare0226, dc=cern, dc=ch
RC_REP_CAT_MANAGER_PWD=Testbed1RC
RC_REP_CAT_URL=ldap://lxshare0226.cern.ch:9011/ rc=Testbed1 Replica Catalog, dc=lxshare0226, dc=cern, dc=ch
RC_LOGICAL_COLLECTION=ldap://lxshare0226.cern.ch:9011/lc=test0,rc=Testbed1 Replica Catalog, dc=lxshare0226, dc=cern, dc=ch

5.1.3. NIKHEF GDMP.CONF FILE

#NIKHEF:
# This file /opt/edg/etc/tutor/gdmp.conf is created by LCFG
GDMP_SHARED_CONF=/opt/edg/etc/gdmp.shared.conf
GDMP_SERVICE_NAME=host/tbn03.nikhef.nl
GDMP_VIRTUAL_ORG=tutor
GDMP_CONFIG_DIR=/opt/edg/etc/tutor
GDMP_VAR_DIR=/opt/edg/var/tutor
GDMP_TMP_DIR=/opt/edg/tmp/tutor
GDMP_GRID_MAPFILE=/opt/edg/etc/tutor/grid-mapfile
GDMP_SERVER_PROXY=/opt/edg/etc/gdmp_server.proxy
GDMP_PRIVATE_CONF=/opt/edg/etc/tutor/gdmp.private.conf
GDMP_STORAGE_DIR=/flatfiles/tutor
GDMP_STAGE_FROM_MSS=/opt/edg/sbin/tutor/stage_from_mss.sh
GDMP_STAGE_TO_MSS=/opt/edg/sbin/tutor/stage_to_mss.sh

5.1.4. CERN GDMP.CONF FILE

#CERN:
# This file /opt/edg/etc/tutor/gdmp.conf is created by LCFG
GDMP_SHARED_CONF=/opt/edg/etc/gdmp.shared.conf
GDMP_SERVICE_NAME=host/lxshare0393.cern.ch
GDMP_VIRTUAL_ORG=tutor

GDMP_CONFIG_DIR=/opt/edg/etc/tutor
GDMP_VAR_DIR=/opt/edg/var/tutor
GDMP_TMP_DIR=/opt/edg/tmp/tutor
GDMP_GRID_MAPFILE=/opt/edg/etc/tutor/grid-mapfile
GDMP_SERVER_PROXY=/opt/edg/etc/gdmp_server.proxy
GDMP_PRIVATE_CONF=/opt/edg/etc/tutor/gdmp.private.conf
GDMP_STORAGE_DIR=/flatfiles/SE00/tutor

5.1.5. CNAF GDMP.CONF FILE

#CNAF:
# This file /opt/edg/etc/tutor/gdmp.conf is created by LCFG
GDMP_SHARED_CONF=/opt/edg/etc/gdmp.shared.conf
GDMP_SERVICE_NAME=host/grid007g.cnaf.infn.it
GDMP_VIRTUAL_ORG=tutor
GDMP_CONFIG_DIR=/opt/edg/etc/tutor
GDMP_VAR_DIR=/opt/edg/var/tutor
GDMP_TMP_DIR=/opt/edg/tmp/tutor
GDMP_GRID_MAPFILE=/opt/edg/etc/tutor/grid-mapfile
GDMP_SERVER_PROXY=/opt/edg/etc/gdmp_server.proxy
GDMP_PRIVATE_CONF=/opt/edg/etc/tutor/gdmp.private.conf
GDMP_STORAGE_DIR=/shared/tutor

5.1.6. CC-LYON GDMP.CONF FILE

#CC LYON:
# This file /opt/edg/etc/tutor/gdmp.conf is created by LCFG
GDMP_SHARED_CONF=/opt/edg/etc/gdmp.shared.conf
GDMP_SERVICE_NAME=host/ccgridli04.in2p3.fr
GDMP_VIRTUAL_ORG=tutor
GDMP_CONFIG_DIR=/opt/edg/etc/tutor
GDMP_VAR_DIR=/opt/edg/var/tutor
GDMP_TMP_DIR=/opt/edg/tmp/tutor
GDMP_GRID_MAPFILE=/opt/edg/etc/tutor/grid-mapfile
GDMP_SERVER_PROXY=/opt/edg/etc/gdmp_server.proxy
GDMP_PRIVATE_CONF=/opt/edg/etc/tutor/gdmp.private.conf
GDMP_STORAGE_DIR=/afs/in2p3.fr/grid/StorageElement/prod/tutor

5.1.7. RAL GDMP.CONF FILE

#RAL:
# This file /opt/edg/etc/tutor/gdmp.conf is created by LCFG
GDMP_SHARED_CONF=/opt/edg/etc/gdmp.shared.conf
GDMP_SERVICE_NAME=host/gppse05.gridpp.rl.ac.uk
GDMP_VIRTUAL_ORG=tutor
GDMP_CONFIG_DIR=/opt/edg/etc/tutor
GDMP_VAR_DIR=/opt/edg/var/tutor
GDMP_TMP_DIR=/opt/edg/tmp/tutor
GDMP_GRID_MAPFILE=/opt/edg/etc/tutor/grid-mapfile
GDMP_SERVER_PROXY=/opt/edg/etc/gdmp_server.proxy
GDMP_PRIVATE_CONF=/opt/edg/etc/tutor/gdmp.private.conf
GDMP_STORAGE_DIR=/flatfiles/05/tutor

6. ACKNOWLEDGMENTS

We want to thank many different people belonging to various EDG work packages, who contributed with example material, suggestions or assistance in the set up of the exercises. We specially wish to thank Cal Loomis (IN2P3), Stephen Burke (PPARC), Flavia Donno, Roberto Barbera, Sergio Andreozzi (INFN), Heinz Stockinger, Maite Barroso Lopez, Akos Frohner, Bob Jones, Peter Kunszt, Emanuele Leonardi, Erwin Laure (CERN), Jeff Templon, Kos Bors (NIKHEF).
Special thanks to Christophe Jacquet at CERN, who has tested all exercises and has provided precious feedback and corrections.
Major reviews and corrections to this document are due to Heinz Stockinger, Bob Jones, Erwin Laure (CERN), Kos Bors (NIKHEF), to whom the authors would like to address a warm thank you.