



# DataGrid

## EDG-BROKERINFO USER GUIDE

---

Document identifier: **DataGrid-01-TEN-0135-0\_0**

Date: **13/11/2002**

Work package: **WP1-WP2**

Document status: **DRAFT**

Deliverable identifier:

---

Abstract: This document discusses about the BrokerInfo file, the edg-brokerinfo command line interface and the correspondent API.

### Delivery Slip

From	Name	Partner	Date	Signature

### Document Log

Issue	Date	Comment	Author
0_0	13/11/02	First draft	Donno F., Salconi L., Sgaravatto M.
1-0		Final draft	

### Document Change Record

Issue	Item	Reason for Change

### Software / Documents / URLs

Link	Refer to:
[U1]	<a href="http://www.pd.infn.it/~sgaravat/Grid/datagrid-01-not-0113-1_2.pdf">http://www.pd.infn.it/~sgaravat/Grid/datagrid-01-not-0113-1_2.pdf</a>
[U2]	<a href="http://datagrid.in2p3.fr/distribution/datagrid/wp2/RPMS/BrokerInfo-gcc32dbg-3.0-2.i386.rpm">http://datagrid.in2p3.fr/distribution/datagrid/wp2/RPMS/BrokerInfo-gcc32dbg-3.0-2.i386.rpm</a>

### Terminology / Glossary

EDG	Eropean DataGrid
UI	User Interface
WN	Worker node
CE	Computing element
SE	Storage element
RB	Resource broker



## Table of Contents

1. INTRODUCTION.....	5
2. BRIEF DESCRIPTION FOR THE NEED OF THE BROKERINFO FILE.....	6
3. THE BROKER INFO FILE.....	8
4. THE COMMAND LINE INTERFACE.....	11
4.1. EXAMPLE: GETTING THE CE ID USING THE EDG-BROKERINFO COMMAND FROM THE WN.....	12
4.2. EXAMPLE: PARSING THE BROKERINFO FILE FROM THE UI.....	14
5. THE BROKERINFO API - C++ CLASS.....	16
6. THE REPLICACATALOGB API - C++ CLASS.....	18





## 1. INTRODUCTION

In the BrokerInfo file (created by the RB), the various information concerning a job (chosen CE, location of the considered data, etc.) is available, so that they can be used by the applications (in particular to get the handle to the physical file names that the job has to use).

The BrokerInfo file, written relying on the Condor ClassAds, can not be so easy to be read and interpreted. We therefore provide a command line tool and a C++ API (described in the next chapters) to allow users to easily parse this file.

A first document discussing about the BrokerInfo file was [U1]: sections 2 and 3 come from that document.

The edg-brokerinfo software is available in the EDG repository ([U2]): BrokerInfo\*.rpm

## 2. BRIEF DESCRIPTION FOR THE NEED OF THE BROKERINFO FILE

When the user specifies in the *InputData* field of the JDL expression (used when a job is submitted) the input data selection, the Resource Broker finds out where (in which Storage Element(s)) these data are physically stored, and based on this info, chooses the closest most suitable Computing Element where to submit the job. Therefore the Resource Broker takes into account the replica information for scheduling.

In the *InputData* field, the user can specify lists of Logical File Names and/or Physical File Names. The Storage Element is chosen considering also the protocol that the application is able to “speak” (this is defined in the *DataAccessProtocol* field of the JDL expression) and “satisfied” by the Storage Element. Here we give a set of definitions to better understand what follows:

### ***LFN = Logical File Name***

It is an arbitrary string corresponding to the physical name of the file.

Eg.: myfile.dat

### ***PFN = Physical File Name***

It has the following form: <hostname>/<path>/<filename>, where <hostname> is the hostname of the Storage Element serving the file, <path> is a protocol independent common path and <filename> is an arbitrary string which corresponds to the physical name of the file.

Eg.: se1.cern.ch/data/myfile.dat

### ***TFN = Transport File Name***

It has the form:

<protocol>://<hostname>:<port>//<path>/<filename>

where <protocol> defines the protocol used to access the file, <hostname> is the hostname of the Storage Element serving the file, <port> is the IP port used by the defined protocol, <path> is a protocol independent common path as it appears in the corresponding PFN, <filename> is the real name of the file as it appears in the corresponding PFN.

Once the job is dispatched on the Computing Element, the application needs to get the handle to the physical filename to open. The solution to this problem is given by the following user API. In the examples we assume that se2.pd.infn.it is the storage element (SE) closest to the computing element chosen by the broker to dispatch the job in question. A GridFTP server is running on this SE, and it is listening on port 4444. As we mentioned, the protocol and port information is published in the SE MDS objectclass. The file system served by the storage element se2.pd.infn.it can also be accessed locally from the Computing Element chosen by the Broker.

The Storage Element file system where the file of interest is stored can be accessed from the Computing Element via the /mount\_point local mount point.

- **PFN[] = getPhysicalFileNames(LFN)**

eg: `getPhysicalFileNames("myfile.dat") =`  
`(se1.cern.ch/cms/mypath/myfile.dat,`  
`se2.pd.infn.it/cms2/mypath/myfile.dat,`  
`se3.in2p3.fr/mypath3/myfile.dat)`

- **PFN = `getBestPhysicalFileName(PFN[], String[] protocols)`**

eg: `getBestPhysicalFileName( (list of PFNs), ("gridftp", "file")) =`  
`se2.pd.infn.it/cms2/mypath/myfile.dat`

- **TFN = `getTransportFileName(PFN, String protocol)`**

eg: `getTransportFileName ((se2.pd.infn.it/cms2/mypath/myfile.dat),`  
`"gridftp" ) = gridftp://se2.pd.infn.it:4444//cms2/mypath/myfile.dat`

`getTransportFileName ((se2.pd.infn.it/cms2/mypath/myfile.dat),"file" )`  
`= file:///mount_point/cms2/mypath/myfile.dat`

- **filename = `getPosixFileName(TFN)`**

eg: `getPosixFileName(file:///mount_point/cms2/mypath/myfile.dat) =`  
`/mount_point/cms2/mypath/myfile.dat`

- **`getSelectedFile(LFN,String protocol, TFN, filename)`**

This is a wrapper call, calling the four above methods in sequence, if the value for the *protocol* parameter is equal to "file", otherwise the first 3 methods are invoked.

*LFN* and *protocol* are the input parameters, while *TFN* and *filename* are the output parameters.

eg: `getSelectedFile("myfile.dat","file", tfn, filnam)`  
`tfn = file:///mount_point/cms2/mypath/myfile.dat`  
`filnam = /mount_point/cms2/mypath/myfile.dat`

A way for making these APIs aware of the Resource Broker choice is needed: this is explained in the next section.

### 3. THE BROKER INFO FILE

WP1 and WP2 agreed on a possible approach to this problem: the idea is to “pack up” this information in a file (*.BrokerInfo*), which is sent to the job-working directory of the worker node with the input application sandbox. Here is the format (based on Condor ClassAds) and the content of this broker info file:

```
[  
CE = ResourceId;  
DataAccessProtocol = {proto1, proto2, ..., protop};  
InputPFNs = { PFN1,...,PFNk};  
LFNs = { LFN1, LFN2,...,LFNn };  
PFNs = {  
    {PFN1,1, PFN1,2, ..., PFN1,m1},  
    {PFN2,1, PFN2,2, ..., PFN2,m2},  
    ...  
    ...  
    { PFNn,1, PFNn,2,..., PFNn,mn}  
};  
SEs = {SE1, SE2, ..., SEq};  
SEProtocols = {  
    {proto1,1, proto1,2, ..., proto1,p1},  
    {proto2,1, proto2,2, ..., proto2,p2},  
    ...  
    ...  
    {protoq,1, protoq,2, ..., protoq,pq}  
};  
SEPorts = {  
    {port1,1, port1,2, ..., port1,p1},  
    {port2,1, port2,2, ..., port2,p2},  
    ...  
    ...  
    {portq,1, portq,2, ..., portq,pq}  
};  
CloseSEs = {SE1, SE2, ..., SEt};  
SEMountPoint = {mountpoint1, mountpoint2, ...,mountpointt};  
]
```

*CE = ResourceId* specifies the identifier of the Computing Element where the job has been dispatched.

$DataAccessProtocol = \{proto_1, proto_2, \dots, proto_p\}$  is the list of protocols that the application is able “to speak” (this is the value specified as *DataAccessProtocol* in the JDL expression).

*InputPFNs* is the list of physical file names specified in the *InputData* attribute of the JDL expression.

Then, for each logical file name specified in the *InputData* attribute, the list of all correspondent physical file names is specified: *LFNs* specifies the list of LFNs specified in the *InputData* attribute, while *PFNs* is a list of list: the first list represents the physical file names associated to the first logical file specified in the *LFNs* list, the second list corresponds to the second LFN, etc...

Then the list *SEs* is specified: these are Storage Elements storing files specified in the *PFNs* and/or *LFN2PFN* lists.

For each of these storage elements, a list of “supported” protocols (attribute *SEProtocols*), and, for each protocol, the correspondent port number (attribute *SEPorts*) are then provided.

$CloseSEs = \{SE_1, SE_2, \dots, SE_n\}$  defines the list of Storage Elements “close” to the Computing Element where the job has been submitted. We assume that this information is available in the MDS (provided by the WP4 information providers).

Each storage element is identified by the correspondent full host name.

For each of these Storage Elements, the mount point to that Storage Element from the Computing Element where the job has been dispatched (if there is “local access”) is specified: the first element of the list is the mount point for the first SE specified in the *CloseSEs* list, the second element corresponds to the second element of the *CloseSEs* list, etc...

Here is an example of a .BrokerInfo file:

```
[
CE = lxde01.pd.infn.it:2119/jobmanager-lsf-grid01;
DataAccessProtocol = {"gridftp", "file"};
InputPFNs = ["se1.pd.infn.it/data01/file762.dat"];
LFNs = {"file76.dat", "filex", "mndb"};
PFNs = {
    {"se1.pd.infn.it/data00/file76.dat",
     "se.cern.ch/cms/mn/file76.dat"},
    {"se1.pd.infn.it/data00/filex",
     "se.in2p3.fr/data/00/filex"},
    {"se.cern.ch/cms/mn/cal1.DB"}
};
SEs      = {"se1.pd.infn.it", "se.cern.ch", "se.in2p3.fr"};
SEProtocols = {
    {"gridftp", "file"},
```

```
        {"gridftp", "rfio"},
        {"gridftp"}
    };
SEPorts = {
    {"4444", undefined}, #no port number for file protocol
    {"4444", "5555"},
    {"4433"}
};
CloseSEs = {"se1.pd.infn.it", "se2.pd.infn.it"};
SEMOUNTPOINT = {"/disk1", undefined}; # no local access to se2.pd.infn.it from this CE
]
```

Both a command line tool (described in section 4) and a C++ API (described in section 5) is provided, to allow users to easily “read” the content of this BrokerInfo file.

## 4. THE COMMAND LINE INTERFACE

The edg-brokerinfo is the command line tool to “read” the BrokerInfo file.

The edg-brokerinfo command is installed on UI and WN machines. It is normally located in:  
`/opt/edg/bin/edg-brokerinfo`

If this command is called from the shell prompt without specifying any arguments, a message is returned:

```
[user@UI] # /opt/edg/bin/edg-brokerinfo

/opt/edg/bin/edg-brokerinfo: please specify a function with parameters
use /opt/edg/bin/edg-brokerinfo --help for functions and parameters list
```

With the `--help` option, users can get the list of the available flags, functions and arguments:

```
[user@UI] # edg-brokerinfo --help

Use: # edg-brokerinfo [-v] [-f filename] function [parameter] [parameter] ...
-v : for verbose, formatted output
-f : for specify a brokerinfo file

supported funtions are:
./edg-brokerinfo getCE
./edg-brokerinfo getDataAccessProtocol
./edg-brokerinfo getInputPFNs
./edg-brokerinfo getSEs
./edg-brokerinfo getCloseSEs
./edg-brokerinfo getSEMouintPoint <SE>
./edg-brokerinfo getLFN2PFN <PFN>
./edg-brokerinfo getSEProto <SE>
./edg-brokerinfo getPFNs
./edg-brokerinfo getSEPort <SE> <Proto>
./edg-brokerinfo getPhysicalFileNames <LFN>
./edg-brokerinfo getTransportFileName <PFN> <Proto>
./edg-brokerinfo getPosixFileName <TFN>
./edg-brokerinfo getSelectedFile <LFN> <Proto>
./edg-brokerinfo getBestPhisicalFileName <PFN1> <PFN2> .. ! <Protol> <Proto2> ..
```

Therefore two possible flags (`-v` and `-f`) can be used:

`-v` : for a verbose, formatted output. The interface gives various information like: the API function name and namespace, the API returned value (namespace\_SUCCESS or namespace\_FAILED) and, of



course, the desired output value. For example, compare the two following outputs (with and without –v flag):

```
[user@UI] # edg-brokerinfo -v getCE
BrokerInfo::getBIFileName(): .BrokerInfo
BrokerInfo::getCE():
-> lxde01.pd.infn.it:2119/jobmanager-lsf-grid01
-> BI_SUCCESS
```

```
[user@UI] # edg-brokerinfo getCE
lxde01.pd.infn.it:2119/jobmanager-lsf-grid01
```

The default is without verbose output.

**-f**: on UI let users specify the pathname of the BrokerInfo file, e.g.:

```
[user@UI] # edg-brokerinfo -v -f bi_bile_1 getCE
lxde01.pd.infn.it:2119/jobmanager-lsf-grid01
```

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 (see the example in section 4.1).

The second one is from the UI, and therefore after job execution, after having retrieved the BrokerInfo file (in section 4.2 an example is provided).

To use the edg-brokerinfo command, the **EDG\_WL\_RB\_BROKERINFO** must be defined (unless the option –f is used): it should refer to the pathname of the BrokerInfo file. If is not defined the interface will give back an error message. It is not necessary to set this variable in the worker node, since it is automatically defined by the “system”.

For what concerns the various functions (getCE, getDataAccessProtocol, etc.), they are explained in details in the API sections (sections 5 and 6).

Please pay attention to the function getBestPhysicalFileName:

```
getBestPhysicalFileName <PFN1> <PFN2> .. ! <Proto1> <Proto2> ..
```

the char “!” must be used as a separator between the physical file name and protocol vectors.

#### 4.1. EXAMPLE: GETTING THE CE ID USING THE EDG-BROKERINFO COMMAND FROM THE WN



Write a brokerinfo\_test\_1.jdl file, referring to a job which issues the edg-brokerinfo command to get the CE where the job has been dispatched, e.g.:

```
Executable      = "/opt/edg/bin/edg-brokerinfo";
Arguments       = "-v getCE";
StdOutput       = "stdout.txt";
StdError        = "stderr.txt";
OutputSandbox   = {"stdout.txt", "stderr.txt"};
```

Then submit this job from an UI machine to the grid:

```
[salconi@testbed009 jdl]$ dg-job-submit -o JId brokerinfo_test_1.jdl

Connecting to host grid004f.cnaf.infn.it, port 7771
Logging to host grid004f.cnaf.infn.it, port 15830

===== dg-job-submit Success =====
The job has been successfully submitted to the Resource Broker.
Use dg-job-status command to check job current status. Your job identifier (dg_jobId) is:
https://grid004f.cnaf.infn.it:7846/131.154.99.139/124918297052489?grid004f.cnaf.infn.it:7771
The dg_jobId has been saved in the following file:
/home/salconi/jdl/JId
=====
```

Then retrieve the output sandbox files:

```
[salconi@testbed009 jdl]$ dg-job-get-output -i JId
*****
JOB GET OUTPUT OUTCOME

Output sandbox files for the job:
-
https://grid004f.cnaf.infn.it:7846/131.154.99.139/124918297052489?grid004f.cnaf.infn.it:7771
have been successfully retrieved and stored in the directory:
/tmp/124918297052489
*****
```

Then check the standard output file (i.e. the output of the edg-brokerinfo command):

```
[salconi@testbed009 jdl]$ cat /tmp/124918297052489/stdout.txt
BrokerInfo::getCE():
-> testbed008.cnaf.infn.it:2119/jobmanager-pbs-short
-> BI_SUCCESS
```



## 4.2. EXAMPLE: PARSING THE BROKERINFO FILE FROM THE UI

Write a brokerinfo\_test\_2.jdl file like this:

```
Executable      = "getBI.sh";
StdOutput       = "stdout.txt";
StdError        = "stderr.txt";
InputSandbox    = {"getBIfile.sh "};
OutputSandbox   = {"stdout.txt", "stderr.txt"};
```

Where *getBIfile.sh* is a simple bash script like this:

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

which therefore prints in the standard output the content of the BrokerInfo file.

Submit the job and retrieve the output sandbox files:

```
[salconi@testbed009 jdl]$ dg-job-submit -o JId brokerinfo_test_2.jdl

Connecting to host grid004f.cnaf.infn.it, port 7771
Logging to host grid004f.cnaf.infn.it, port 15830

===== dg-job-submit Success =====
The job has been successfully submitted to the Resource Broker.
Use dg-job-status command to check job current status. Your job identifier (dg_jobId) is:
https://grid004f.cnaf.infn.it:7846/131.154.99.139/130820297825808?grid004f.cnaf.infn.it:7771
The dg_jobId has been saved in the following file:
/home/salconi/jdl/JId
=====

[salconi@testbed009 jdl]$ dg-job-get-output -i JId

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

Output sandbox files for the job:
-
https://grid004f.cnaf.infn.it:7846/131.154.99.139/131239299403696?grid004f.cnaf.infn.it:7771
have been successfully retrieved and stored in the directory:
/tmp/131239299403696
```



\*\*\*\*\*

```
[salconi@testbed009 jdl]$ cp /tmp/131239299403696/stdout.txt file_to_parse
```

Now the BrokerInfo file is a local file (*file\_to\_parse*) and it is possible to parse it and to get from it all the needed information. For example it is possible to get the CE id:

```
[salconi@testbed009 jdl]$ edg-brokerinfo -v -f file_to_parse getCE
```

```
BrokerInfo::getCE():  
-> testbed008.cnaf.infn.it:2119/jobmanager-pbs-short  
-> BI_SUCCESS  
[salconi@testbed009 jdl]$
```

## 5. THE BROKERINFO API - C++ CLASS

This is a simple class for parsing the *BrokerInfo* file.

BrokerInfo has been developed like a singleton class, so only one object's instance can be created in a program context. There isn't a public data structure. Its private data structure contains:

- a string *BrokerInfoFile\_*, filled by the value of environment's variable EDG\_WL\_RB\_BROKERINFO;
- one *instance\_* handler, used for referencing the BrokerInfo object;
- one ClassAd\* *ad\_* object for parsing the file;

The public methods are:

- *BrokerInfo\* instance(void);*  
returns the handle of the singleton object.
- *BI\_Result getCE(string& CE);*  
returns BI\_SUCCESS if the file has a CE parameter specified, otherwise, if operation fails, a BI\_ERROR.  
The string CE's value is referenced to file's value.
- *BI\_Result getDataAccessProtocol(vector<string>& DAPs);*  
returns BI\_SUCCESS if the file has a DAPs parameter list specified, otherwise, if operation fails, a BI\_ERROR.  
The string vector DAPs is referenced to the specified data access protocol list, ordered by SE order list.
- *BI\_Result getInputPFNs(vector<string>& PFNs);*  
returns BI\_SUCCESS if the file has a PFNs parameter list specified, otherwise, if operation fails, a BI\_ERROR.  
The string vector PFNs is referenced to the specified physical file name list, ordered by SE order list.
- *BI\_Result getLFN2PFN(string LFN, vector<string>& PFNs);*  
returns BI\_SUCCESS if the file has a LFN to PFNs parameter list with correct mapping, otherwise, if operation fails, a BI\_ERROR.  
The parameter LFN contains the logical file name specified and the parameter PFNs will be referenced to the list of physical file name matched.
- *BI\_Result getSEs(vector<string>& SEs);*  
returns BI\_SUCCESS if the file has SE parameter list, otherwise, if operation fails, a BI\_ERROR.  
The parameter SEs is referenced to the list of storage element specified.
- *BI\_Result getSEproto(string SE, vector<string>& SEProtos);*  
returns BI\_SUCCESS if the file has SEProtos parameter list, otherwise, if operation fails, a BI\_ERROR.

The parameter SE contains one storage element name and parameter SEprotos is referenced to the list of protocols implemented on specified SE.

- *BI\_Result getSEPort(string SE, string SEProtocol, string& SEPort);*  
returns BI\_SUCCESS or, if operation fails, a BI\_ERROR.

The parameter SE contains one storage element name and parameter SEProtocol contains the protocol name, parameter SEPort will be referenced to the value of matching port for that protocol.

- *BI\_Result getCloseSEs(vector<string>& SEs);*  
returns BI\_SUCCESS or, if operation fails, a BI\_ERROR.  
The parameter list SEs will be referenced to the list of the computer element's nearest storage element specified in file.

- *BI\_Result getSEMOUNTPoints(string CloseSE, string& SEMOUNT);*  
returns BI\_SUCCESS or, if operation fails, a BI\_ERROR.  
The parameter CloseSE contains one near storage element name and parameter SEMOUNT will be referenced to the relative mount point.

The private methods implemented in BrokerInfo class are:

- *BrokerInfo(void);*  
is the constructor class, this method search and check the environment variable and read the brokerinfo file, create and fill the ClassAd object. If is not possible to parse the file with specified PARSE\_RULE value a fault assertion will be generated.
- *BI\_Result vSearch(char\* searchstr, vector<string>& retvect);*  
utility code - search for a given string (param searchstr) into brokerinfo file and build a vector string (reference param retvect) with objects found.
- *BI\_Result sSearch(char\* searcharg, string searchstr, int& position);*  
utility code - search for a given argument (param searcharg) in a given string (param searchstr) that contains a list of values and reference the param position to the matching position of searcharg in searchstr.
- *BI\_Result svIndexBuild(char\* sarg, string sstr, string varg, vector<string>& retvect);*  
utility code - is a string/vector index build function.
- *void vBuild(string buildstr, vector<string>& retvect);*  
utility code - explode a given string (param buildstr) that contains a list of values in a referenced (param retvect) list of separate values.

## 6. THE REPLICACATALOGB API - C++ CLASS

This is a simple class which implements the functions described in section 2.  
Its private data structure contains only:

- one BrokerInfo\* *brokerinfo\_* handler for parsing the file;

The public methods are:

- *ReplicaCatalogB(void);*  
the default constructor;
- *vector<string> getPhysicalFileName(string LFN);*
- *string getBestPhysicalFileName(vector<string> PFN, vector<string> Protocols);*
- *string getTransportFileName(string PFN, string Protocol);*
- *string getPosixFileName(string TFN);*
- *getSelectedFile(string LFN, string Protocol, string& TFN, string& FileName);*

which have been fully described in section 2.