

Subject:           **THE RESOURCE BROKER INFO FILE**

Author:            Flavia Donno, Salvo Monforte, Francesco Prelz, Massimo Sgaravatto, Livio Salconi

Partner:           **INFN**

Diffusion:        **WP MANAGERS**

Information:

**Version 1\_3**

---

## Contents:

1. THIS SOLUTION IS BASED ON THE NEGOTIATED WP4 AND WP5 MDS INFO.....	2
2. IMPACT ON TESTBED INSTALLATIONS (WP6).....	2
3. BRIEF DESCRIPTION FOR THE NEED OF AN INFO FILE.....	2
4. THE BROKER INFO FILE .....	5
5. HOW WILL WP1 AND WP2 CO-OPERATE TO PROVIDE THIS SOLUTION ?.....	7
6. THE BROKERINFO C++ CLASS.....	7
7. THE REPLICACATALOGB C++ CLASS.....	9

<b>WARNING: What follows is assumed to happen by M9</b>
---

**WARNING: What follows is assumed to happen by M9**

## 1. THIS SOLUTION IS BASED ON THE NEGOTIATED WP4 AND WP5 MDS INFO

We describe the technical details of the communication of information that is available to the PM9 WP1 Resource Broker to a running application (via the Replica Catalog Access library, provided by WP2). The following solution is based on the format of the MDS information that we already negotiated with WP4 (CE information) and WP5 (SE information). In particular, CE information must contain the list of SEs (identified by a hostname for PM9) that are “close”(LAN-wise) to the CE, and the SE mount path, *in case the SE is accessible from the CE worker nodes via Posix file calls* (e.g., the SE storage area is mounted to all the worker nodes via NFS). SE information must contain the list of “close” CEs, and the list of protocols that the SE is able to speak (an appropriate server must run on the SE node for each listed protocol; typical protocols may be GridFTP, RFIO, etc.). SE information must also contain the port number where each of these servers is listening.

## 2. IMPACT ON TESTBED INSTALLATIONS (WP6)

In order for this transitional way to construct TFNs (Transport File Names, please read below if you would like to know more) to work, we make two assumptions on the configuration of CEs and SEs in the PM9 testbed. These constraints will be relaxed for Release 2, when we will hopefully be able to refine and complete the definition of the information schema. We feel we cannot afford the extra complication of dropping these assumptions for PM9:

1. **Servers for different protocols on the same SE have to be configured to serve a given file with the same pathname.** That is: if, for example, a file is accessed via RFIO with path `/complex/path/to/my/file`, and a GridFTP server is available for the same SE, it should be possible to get it from a GridFTP server using the same, absolute `/complex/path/to/my/file` pathname.
2. **When an SE is mounted locally on a CE (e.g. via NFS), it should be mounted so that the local file path can be obtained from the SE path (which is unique for the above assumption) by just prepending a local mount path.** This can require that some machinery be used for the mount procedure. For instance, if the SE above exports `/complex/path`, and a CE mounts this area under `/my/local`, appropriate symbolic links should be added in the SE storage area so that the file can be accessed locally as `/my/local/complex/path/to/my/file`. The CE will then advertise `/my/local` as the mount point for the specified SE.

## 3. BRIEF DESCRIPTION FOR THE NEED OF AN INFO 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.

We assume that WP5 will publish this information in the MDS among the other attributes describing 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 WP2 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 WP2 API's aware of the Resource Broker choice is needed.

## 4. 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 a proposal for the format (based on Condor ClassAds) and for 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
]
```

## 5. HOW WILL WP1 AND WP2 CO-OPERATE TO PROVIDE THIS SOLUTION ?

WP2 is committed to provide an implementation of `PFN[] = getPhysicalFileNames(LFN)` for M9.

This implementation directly accesses the Replica Catalog information. We call this the implementation "A" of `getPhysicalFileNames`. This is the only function that will exist in "A" form for PM9.

The other calls described in this document will exist at PM9 only in a form that operates based on the contents of the `.BrokerInfo` file *only*. We call this implementation "B". Implementation "B" of `getPhysicalFileNames` (and of the wrapper call `getSelectedFile`) will use information in the `.BrokerInfo` file whenever possible, and will fall back to implementation "A" if nothing can be found in the file. Implementation "A" and "B" of these calls will be included in the library that WP2 will provide for the applications at PM9, and will be implemented by WP2.

WP1 is responsible for the format of the `.BrokerInfo` file, and will implement a service library to access the file. The code for this library will be kept in the WP1 CVS server. For PM9, the library will be built along with the WP2 library, and distributed as part of it.

## 6. THE BROKERINFO C++ CLASS

This is a simple class for parsing the `.BrokerInfo` file, which contains technical details and information for running an application. `BrokerInfo` is 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. It's 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 it 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.

## 7. THE REPLICACATALOGB C++ CLASS

This is a simple class for give to user and developer an API interface to the BrokerInfo class so it's 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);`

that are fully described in chapter 3.