View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UNT Digital Library

LBNL-57520

Replica Registration Service Functional Interface Specification 1.0

Arie Shoshani, Alex Sim, Kurt Stockinger
Lawrence Berkeley National Laboratory

Contributors:
Jean-Philippe Baud, CERN
James Casey, CERN
Peter Kunszt, CERN
Robert Schuler, USC-ISI

Overview 2
- Referring to files 3
  - GUIDs, LFNs, and PFNs 3
  - PFNs and SURLs 3
  - LFN namespace structure 4
  - Name formats 4
  - File attributes 5
- Registering files 5
  - A uniform interface for all registration functions 5
  - Continuous registration and registration modes 6
  - Error directives 7
  - First-time and subsequent registration 8
  - Unregister 9
  - Checking the existence of file references 9
  - Synchronous vs. non-synchronous requests 10
- Coventions 10
  - Notation 10
  - Parameters 11
  - Error Reporting 11

Replica Registration Service Interface Specification 12
- Register Functions 12
  - openCollectiveRegistration 12
  - register 13
  - closeCollectiveRegistration 14
  - getRegistrationStatus 14
  - getRegistrationSummary 15
  - abortRegistration 16
  - getAbortRegistrationStatus 16
  - unregister 17
  - getUnregisterStatus 18
  - getUnregisterSummary 19

Discovery Functions 19
getFileReferences.. 19
getFileReferenceStatus 20
getSystemInfo 20
Name Space Functions.. 21
makeDirectory.. 21
removeDirectory 21
makeLink 22
listDirectory 22
getListDirectoryStatus 23
Attribute Functions 23
getFileAttributes 23
getFileAttributesStatus... 23
insertFileAttributes 24
getDirectoryAttributes 24
getDirectoryAttributesStatus.. 24
insertDirectoryAttributes 25

Overview

The goal of the Replica Registration Service (RRS) is to provide a uniform interface to various file catalogs, replica catalogs, and metadata catalogs. It can be thought of as an abstraction of the concepts used in such systems to register files and their replicas. Some experiments may prefer to support their own file catalogs (which may have their own specialized structures, semantics, and implementations) rather than use a standard replica catalog. Providing an RRS that can interact with such a catalog (for example by invoking a script) can permit that catalog to be invoked as a service in the same way that other replica catalogs do. If at a later time the experiment wishes to change to another file catalog or an RLS, it is only a matter of developing an RRS for that new catalog and replacing the existing catalog. In addition, some systems use metadata catalogs or other catalogs to manage the file name spaces. Our goal is to provide a single interface that supports the registration of files into such name spaces as well as retrieving this information.

Perhaps the most important motivation for an RRS is the need to keep track of the registration process itself. For example, if a catalog is down for a period of time, then there is a need for a service that will continue to try to register the files as soon as the catalog becomes operational. Furthermore, one may wish to find out the status of the registration process, whether it is successful, or pending, or aborted because of errors. Most catalogs are not designed to maintain such information for an entire multi-file (or entire directories) registration requests. The RRS is a service whose purpose is to perform a multi-file registration request based on a desired registration behavior (modes), and keep track of the registration process.

Referring to files

GUIDs, LFNs, and PFNs

There are several ways to refer to a file. If the location of the file is known, one can specify its Physical File Name (PFN). However, since a file may have multiple replicas, it is convenient to refer to the file by using Logical File Names (LFN). Some communities of users prefer to support a unique immutable LFN for each file, and provide a mapping between the LFN and one or more Physical File Names (PFNs). In many cases, LFNs are designed to be structured names. This is a desired property, since the file name conveys a meaning, such as the date, purpose, or conditions that were used at the time the file was generated. However, having such structured names makes it difficult to guarantee global uniqueness of the name. Furthermore, there may be a need to change file names over time, or even have multiple aliases for a file name. For this reason, some communities use a "globally unique identifier", referred to as GUID, to identify a file, in addition to an LFN. Given that a GUID is used for a file, that file can now have multiple LFNs that are treated as name-aliases for the GUID.

Since we wish to have this specification applicable to all communities, we adopt the more general case of having a GUID for a file. In addition, we permit multiple LFNs per GUID. For communities that only use a single LFN and no GUID, we consider that LFN to be equivalent to a GUID.

The one-to-many relationships between a-GUID to-PFNs and a GUID-to-LFNs are shown schematically in Figure 1.

PFNs and SURLs

The most straightforward way to specify a file is by a Physical File Name (PFN). A PFN can be specified as a physical URL made of the format: protocol://machine:port/directory-path/file-name. For example: gridftp://cs.berkeley.edu/home/temp/foo. Note that the protocol specified in this case is the transfer protocol.

Some storage systems support multiple physical devices and multiple directories, and may want to have the freedom of changing the physical location of a file without changing the reference to it by Grid clients. An example of a software layer that permits this functionality is a Storage Resource Manager (SRM). The SRM is a single endpoint for accessing a file regardless of its physical location on a particular site. The site is a virtual entity referring to the collection of resources under the administrative control of the site manager. The concept of a site permits a single filename to be assigned to a physical file regardless of it physical storage location.

The reference for a file on a site is called a "Site URL" (SURL). For example: srm://data.berkeley.edu:4004/dir/foo is the name of a file managed by an SRM residing on the machine data.berkeley.edu, on port 4004. When requesting the file from the SRM using the SURL, the SRM returns the "transfer URL" which is the actual PFN. For

example, for the file above the SRM may return the PFN on another machine, cs.berkeley.edu, by using the URL gridftp://cs.berkeley.edu/home/temp/foo. Note that a PFN is a special case of an SURL, where the transfer service specified by the protocol is the site endpoint. Therefore, we only use SURL in the remainder of this document.

LFN namespace structure

LFNs are commonly organized into directory structures, similar to any file system (such as the Unix file system). Some file systems consider directory names as LFNs as well and assign GUIDs to them (this can be automatically assigned by the catalog). The value of treating directories as LFNs is that one can refer to a directory path in a similar way as a reference to a file. In this specification we allow the creation and removal of directories and references to them, so that systems that support this feature will be accessible through the RRS. This is shown schematically in the box referring to LFNs in Figure 1 by having the directory icon in it.

A common use case for using multiple LFNs is that a file is first registered with a particular LFN, and then additional LFNs are allowed to refer to the same file. The original registration is sometimes referred to as the "primary LFN" directory structure, and subsequent references to it are referred to as "secondary LFNs" or as "LFN-aliases". We do not find this distinction generalizable, useful, or necessary. Therefore we refer to all LFNs in the same way regardless of when they were defined. Thus, the RRS interface does not permit references to "primary LFNs" – only to LFNs.

Finally, the power of symbolic links as provided with common file systems is also considered useful. We make symbolic links visible through the RRS interface, by allowing definitions of links as well as including them in LFNs path names. Note that symbolic links are *not* assigned GUIDs since they point to other GUID objects. This is shown schematically in figure 1 by an arrow pointing from the LFN box to itself. Symbolic links are simply directed pointers from an existing LFN to another.

Name formats

In this specification we assume that there is a way to distinguish between GUIDs, LFNs, and SURLs from their format structure. The easiest way to do that is to use a URI structure and the corresponding protocol labels. Thus, for GUIDs and LFNs the URI format will have a protocol label "guid" and "lfn" respectively. For example:

lfn://star/run12/file17.raw (this is an LFN of a file)
lfn://star/run12 (this is an LFN for a directory)
guid://123456 (the GUID value can be any string)

All other protocol labels will be considered SURLs. Examples of such protocols are: gridftp, ftp, http, srm (for files managed by SRMs). For example:

gridftp://cs.bnl.gov:4004/tmp/run12/file17.raw (this is an SURL for a file)
srm://cs.bnl.gov:4004/tmp/run12/file17.raw (this is also an SURL for a file)

srm://cs.bnl.gov:4004/tmp/run12 (this is an SURL for a directory)

File attributes

File attributes are associated with GUIDs as shown in Figure 1. These attributes represent only global properties that do not depend on where the file resides (the SURL site) and how it is named (its LFNs). Some attributes are considered essential to verifying the correctness of file transfers – these are: fileSize, checkSumType, and checkSumValue. We refer to these attributes as "core" attributes. In addition, there may be other attributes that the underlying catalogs may store. We permit the entry and retrieval of all such attributes through the RRS interface. When requesting these attributes one can refer to only "core" attributes or to "all" attributes. The RRS will return an array of triples: fileAttributeName, fileAttributeType, and fileAttributeValue. Note that all values are passed through the interface as strings. The fileAttributeType refers to the type of the attribute, as it is stored in the underlying catalog.

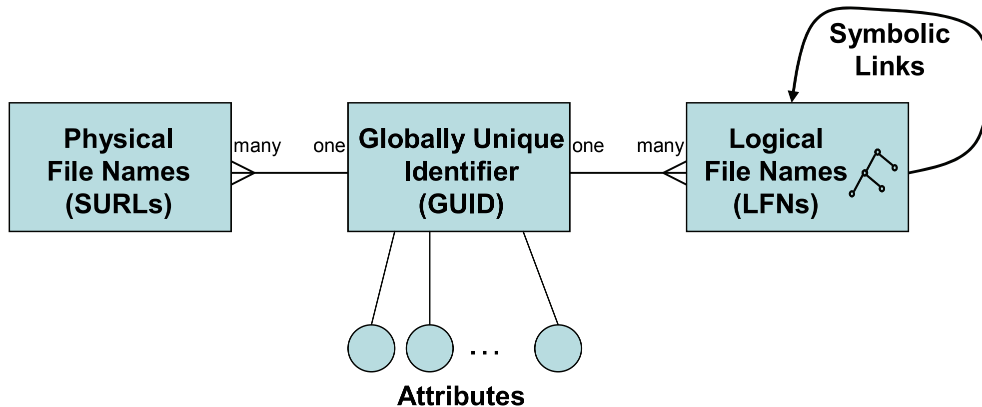

Figure 1: The relationships between GUIDs, LFNs, and SURLs.

Registering files

A uniform interface for all registration functions

The mapping between a GUID and its SURLs is usually provided by some catalog, referred to as a "file catalog", or by a service referred to as the "Replica Location Service" (RLS). In addition, some organizations support LFNs and a service that provides the mapping between a GUID and its LFNs. This capability can be provided by a combination of separate catalogs that support GUID-to-LFN and GUID-to-SURL mappings, or a single catalog that supports both, as shown schematically in Figure 2. An example of a service that supports the GUID-to-LFN mappings is the EDG-RMC (Replica Metadata Catalog), and an example of a service that supports both GUID-to-SURL and GUID-to-LFN mappings is SRB's MCAT. The RRS is designed to provide a single uniform interface for all the registration functions that refer to GUIDs, LFNs, and SURLs.

All the registration functions are for files only. Directory operations for the LFN name space are provided through separate functions, such as makeDirectory, removeDirectory, makeLink, and listDirectory. Files can be referred to by their GUID, LFNs, or SURLs. We use the term "*file references*" to refer to any of these names. All registration requests are made of pairs of file references, such as (LFN, SURL). The first item of this pair is referred to as "given" and the second item as "toBeRegistered". For example, register (LFN, SURL) is interpreted as "for the given LFN, register the SURL". Similarly, register (SURL, SURL) is interpreted as "for the given (first) SURL, register the (second) SURL". In such cases, the RRS may need to get first the GUID for the existing file reference (if it is not a GUID), and then register the second file reference using the GUID. In some cases, we allow a file reference to be null, such as the first-time registration of an LFN without providing a GUID, denoted as (--, LFN). In this case, it is expected that the underlying catalog will generate the GUID. This is explained in more details in the section on first-time and subsequent registration.

Note: From the discussion above, it is obvious that all registration actions are for file references. However, in the remainder of the document we often use the term "register a file" as a short form for "register a file reference".

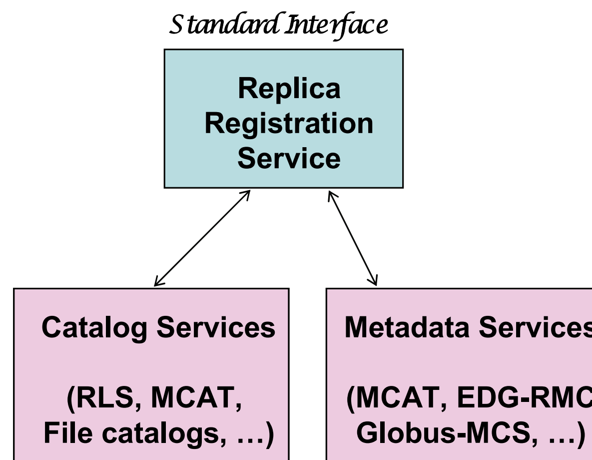

Figure 2: RRS provides a uniform interface for GUID-to-SURL and GUID-to-LFN functions.

Continuous registration and registration modes

The RRS is designed to allow the coordination between copying and registration of files. Because copying a large number of files can be a slow process, it is necessary to allow the registration process to be a long-lasting activity. Therefore, it is necessary to have a way of specifying the beginning and the end of multiple registrations. This is achieved by treating a registration as a long transaction, starting with an

openCollectiveRegistration function, followed by one or more *register* functions, and ending with a *closeCollectiveRegistration* function.

As one or more files are registered, the RRS can use different *modes of registration* as requested by the user. A client may prefer that files be registered one-at-a-time as soon as each is copied successfully, or may prefer to register all the files only after the entire request of copying multiple files (or entire directories) is successful. We refer to the desired behavior as the *registrationMode.* Accordingly, the two registration modes supported are: "continuous" and "atEnd". "continuous" means: register as soon as possible, and "atEnd" means: register all the files after the closeCollectiveRegistration function is issued. Another advantage of this choice is that it allows the burden of accumulating the deferred registration of multiple files (till all the copying is finished) to be placed on the RRS; that is, the RRS has to accept and manage delayed registration requests. Thus, the client or the component calling the RRS does not have to accumulate the deferred registration requests. Instead, it can pass them on to the RRS.

The implementation of the registration modes behavior depends on the target catalog. Some catalogs permit bulk registration of files, a feature that the RRS can choose to take advantage of. Some may prefer a limited number of files in the bulk registration (such as 100 at a time), and some may allow only a single file registration at-a-time. The RRS has to try and perform the registration as close as possible to the requested mode.

Error directives

Registration to the underlying catalog(s) may result in unrecoverable errors. Typical errors are that a GUID, LFN or SURL is not found. For example, registering an SURL for an given LFN may result in an error that the LFN was not found, or that the SURL already exists.

Under failure conditions, clients may prefer different behaviors. We refer to this as *errorDirectives*. These can be specified at the time the collective registration request is initiated with the *openCollectiveRegistration* function. Three error directives are supported:

- stop: register files until a non-transient error occurs and stop.
- stopAndUndo: register files until a non-transient error occurs, stop, and unregister all the files submitted for registration so far (undo).
- continue: record the error and continue registering files.

If a registration request involves thousands of files, it may not be wise to stop or undo the entire request because of a single error. It may be better to permit a few errors before the error directive gets triggered. We allow for such a parameter, called the *errorDirectiveTrigger,* to be set as an integer. Regardless of whether the trigger occurs, the RRS records all such errors.

First-time and subsequent registration

The registration of files into a catalog requires the distinction between a first-time registration, and subsequent registrations. During a first-time registration, a unique GUID needs to be provided by the requester, or is automatically generated by the underlying catalog GUID generation service. Some simple file catalogs use the source physical file name, that was first registered, as the GUID. In other catalogs the GUID is generated by a its own "GUID generator" that guarantees a globally unique identifier. Our goal is to have a single Replica Registration Service (RRS) that can accommodate special purpose file catalogs (such as a file catalog of a HEP experiment), catalog services (such as the RLS), or other more general catalogs.

As mentioned above, all file registrations have pairs of file references, such as (LFN, SURL). For subsequent registrations the first item of this pair is referred to as "given" and the second item as "toBeRegistered". Thus, the "given" item has to be found in the underlying catalog, while the second item should not exist. In contrast, for first-time registration, such as (GUID, LFN), both items need to be registered, and therefore both should not exist in the underlying catalog.

To distinguish between a first-time registration and subsequent registrations the *register* function has a Boolean flag called "firstTimeRegistration". When requesting a first time registration, the flag must be "true". In first-time registration, the combination of the (GUID, LFN), and (GUID, SURL) can be specified. Note that the first file reference can only be a GUID, and it needs to be registered as well. In addition, we permit the specification of a single file reference: register (--, LFN) and register (--, SURL), in which case the underlying catalog is expected to generate a GUID. The registration of a GUID by itself is not permitted.

The (GUID, LFN) or (--, LFN) first-time registration amounts to assigning to a newly registered file an LFN. This can be used to register the LFNs for files ahead of time, and later assign to them SURLs in a subsequent registration – a common use case. Similarly, it is possible to register a file with a (GUID, SURL) or (--, SURL), and later assign an LFN to that file in a subsequent registration.

When a subsequent registration is requested, the RRS needs to verify that the "given" file reference is already registered. For example, a registration of a (GUID, SURL) implies that a new SURL is registered for that GUID. The RRS needs to check that the GUID already exists, and also check that the SURL does not exist. To allow full flexibility, we allow the registration of an LFN or an SURL given an existing GUID, LFN, or an SURL. This can simplify the client interaction with the RRS. For example, register (LFN, SURL) may require to get the GUID for the LFN first from a metadata catalog, and then register the (GUID, SURL) to a replica catalog. The RRS is designed to save the client from having to do this extra step.

To summarize, all registration requests are made pairs of file references. For first-time registration, the registration of (GUID, LFN) or a (GUID, SURL) implies that the first file reference, the GUID, needs to be registered as well, and therefore should not exist.

In subsequent registration, the first file reference *must exist* and the second file reference *should not exist* in the underlying catalogs. The RRS relies on the underlying catalogs to verify correctness. Therefore, good catalogs should provide verification of the existence of file references.

Unregister

The *unregister* function can be used to refer to a registration previously made by a collective registration request by using the request-token. The most general case is when only a request-token is provided without any specific file references. This is interpreted as "unregister all the file reference registrations in that request". However, since this is a global operation, and can cause serious difficulties if a mistake is made, we added a flag called "unregisterCollectiveRequest" that also has to be set. Note that we do not consider "unregisterCollectiveRequest" meaningful before the closeCollectiveRegistration function is issued, and therefore will return an error in that case, saying "cannot unregister entire request before closeCollectiveRegistration is issued".

All the other cases to consider are requests to unregister specific file references. The specification of what to unregister can be done using pairs of file references. Similar to the subsequent "register" case, for the "unregister" function the first file reference in the pair is interpreted as "given" and the second as "toBeUnregistered". For example, unregister (LFN, SURL) is interpreted as "for the given LFN, unregister the SURL". We also allow the unregistration based on a single file reference, including "unregister (GUID, --)", "unregister (--, LFN)", and "unregister (--, SURL)", along with a flag "*unregisterAllReferences*". If the flag is set to "true", all the file references will be unregistered (i.e. the GUID, all the LFNs, and all the SURLs). This flag is required to be set for "unregister (GUID)". If the flag is set to "false" and the "unregister (--, LFN)" or "unregister (--, SURL)" are issued, then only the specified LFN or SURL are unregistered.

We note that if an unregister function with a request-token is issued, that collective registration request can be in-progress (i.e. not closed yet) and therefore some of the files that need to be unregistered may still be in the RRS queue. In such a case the RRS needs to remove these requests from the queue, and unregister files that were already registered. Specifically, the RRS should suspend the collective registration process, perform the unregistration as requested, and then continue with the registration. The same is the case after the request is closed, but the actual registration is still in-progress. The RRS needs to check the status of the request, and act accordingly.

Checking the existence of file references

One of the reason to have a GUID for a file, is to guarantee uniqueness of the file reference. This implies that it is possible to have LFNs and SURLs that are not unique. That is, the same LFN or the same SURL can be assigned to two or more GUIDs. Although this is not a good practice, and is usually avoided, most catalogs permit this usage, and do not check for conflicts.

We could have chosen all register and unregister functions to always require a GUID to avoid this issue, but this amounts to undue burden on the client to always get a GUID when only the LFN or the SURL is known, and also increases the communication to the RRS. Thus, we permitted the more general case where the LFN (or the SURL) is given, and expect the RRS to verify that the LFN (or the SURL) has a single GUID associated with it. Therefore, it is necessary that the RRS always checks not only that the file reference exists, but also that it corresponds to a single GUID only, otherwise an error should be returned. In such cases, the client still has the possibility to register using the GUID, but it is the responsibility of the client to choose the correct GUID.

We believe that all future file catalogs will enforce uniqueness of LFNs and SURLs within a virtual organization. However, for catalogs that do not enforce this, the RRS can prevent errors by checking that a unique GUID exists for a given LFN or SURL.

Synchronous vs. non-synchronous requests

Some requests may require a long time to perform. For example, a register or unregister request may take a long time depending on how many files are involved, and how busy the underlying catalog is. In such cases, the requests cannot be synchronous – that is, blocked till a response is provided. To support non-synchronous (non-blocking) requests, we use the technique of returning a request-token, and a "status" function corresponding to that request, such as "getUnregisterStatus", or a summary function. The status function can be used to get the details of each file in the request, and the summary function to provide general information including the number of registered and pending requests.

In the case of file registrations, we allow multiple registration calls to be issued under the same collective registration request. For this reason, we have an openCollectiveRegistration function where a request-token is returned. This same request-token is used for the status of all the registrations issued till the closeCollectiveRegistration function is issued.

Coventions

Notation

The following notation is used to denote the function parameters.
The symbol => is used below to express "means"

item() => list of items (ordered collection)
item[] => set of items (unordered collection, array)
{ item, item, … } => tuple of items
{ item | item | … } => choice operator
element => a named variable that has a type: string, integer, boolean, or enumerated,
e.g. "string LFN"

item => a single element, a tuple, a list, or a set, e.g. LFN, LFN[]
{item} => a way of enclosing a complex item for the list or set notation. e.g. { LFN() }[] is used to denote a set of LFN lists
_ => underlining an item makes it mandatory (as opposed to optional)
::= => assignment operator
// => comment
"" => literal, e.g. "stop"

Note: The item definition is recursive allowing composite usage of this notation. For example, { LFN | SURL }[] denotes a set of elements that are either LFNs or SURLs. Similarly, { GUID, LFN }[] denotes a set of GUID-LFN tuples, and the expression { { GUID, LFN }[] }[] denotes "a set of such sets". Usually, it is sufficient to use only one or two levels or recursion for denoting the function parameters. This is the case in this document.

Parameters

The most common parameters are defined as follows. For certain functions, additional parameters are defined in the respective specification.

- userID ::= user identifier. In a Proxy-System, the userID is retrieved from the proxy.
- requestID ::= request identifier. It is used for retrieving the request status. The duration of how long the requestID is kept persistent (lifeTimeOfRequestID), is system and can be retrieved with the function getSystemInfo.
- entryOffset ::= from 0 to total number of entries represented by requestID.
- numberOfEntries ::= number of entries for which status information is requested.
- statusCode::= "in progress" | "done" | "suspended" | "failed".

Notes:

- The entryOffset and numberOfEntries can be used to restrict large outputs. For example, if the number of entries in response to listDirectory is expected to have 10,000 entries, one can specify entryOffset=5000, numberOfEntries=100 to get only 100 entries starting at entry 5000.
- The catalogEndpoint for the RRS has to be specified in a configuration file before deploying the RRS. The client can retrieve this information with the function getSystemInfo.

Error Reporting

The most common errors are defined as follows. For certain functions, additional errors are defined in the respective specification.

The following are considered non-transient errors:

- GUID not found
- LFN not found

- SURL not found
- GUID already exists
- LFN already exists
- SURL already exists

The following is considered a transient failure

- Catalog down (temporarily not accessible)

Replica Registration Service Interface Specification

Register Functions

openCollectiveRegistration

This function has to be called before calling the function “register”. This function is necessary in order to allow for multiple registration calls over a period of time to be considered as a single collective registration request. During this call the registration mode, the error directives, and the errorDirectiveTrigger can be set, and apply to the entire collective registration process. For details of the meaning and usage of these features, see the introductory remarks.

IN: String userID,
String registrationMode, // default: “atEnd”
String registrationErrorDirectives, // default: “stop”
Int errorDirectiveTrigger // default: 1

OUT: String requestID,
String statusCode,
String statusExplanation

Parameters

- registrationMode ::= “atEnd” | “continuous”.
  - atEnd: register all files at the end of the request (default).
  - continuous: register all files as soon as possible (depends on the catalog setup).
- registrationErrorDirective ::= “stop” | “stopAndUndo” | “continue”.
  - Stop (default): register files until non-transient error(s) occur and stop.
  - stopAndUndo: register files until a non-transient error(s) occurs and unregister (undo).
  - continue: continue registering files even if there are non-transient errors, and record errors.
- errorDirectiveTrigger ::= number of errors before error directives “stop” or “stopAndUndo” are triggered.

Note:

- The registration request can specify only a single catalog. Registration to multiple catalogs can be issued as two individual requests to register, without any synchronization between the catalogs.

register

Register one or more file references. For first time registration, the Boolean flag " firstTimeRegistration" must be set to "true"; otherwise, it is considered a subsequent registration. For the semantics and behavior of first-time and subsequent registration, see introductory section.

Notes:

- This function can be called multiple times for a given requestID. The file references will be registered according to the mode and errorDirectives specified in "openCollectiveRegistration". For explanation of the modes, see introductory section.
- Multiple "register" calls can be made until the function "closeCollectiveRegistration" is called (see below).

```
IN:   String userID,
      String requestID,
      Boolean firstTimeRegistration,                  // default: false
      { { String GUID | String LFN | String SURL },   // first file reference (given)
        { String LFN | String SURL } }                // second file reference
                                                      // (toBeRegistered)
      }[]

OUT:  String statusCode,
      String statusExplanation
```

Permitted Use Cases: First Time Registration

- { GUID, SURL }
  For a given GUID, a SURL is registered. The GUID must not exist in the catalog for successful registration.
- { GUID, LFN }
  For a given GUID, an LFN is registered. The GUID must not exist in the catalog for successful registration.
- { --, SURL }
  The file SURL is registered in the catalog and a GUID is generated automatically.
- { --, LFN }
  The LFN is registered and a GUID is generated automatically.

Notes:

- A GUID cannot be registered by itself.
- All file references for first time registration must not exist in the catalog

Permitted Use Cases: Subsequent Registration
All specifications have a pair of file references that must exist – no null allowed. In the first position GUID, LFN and SURL can be specified. In the second position, only LFN and SURL can be specified. The six allowed combinations are:

- { GUID, SURL }
  Register a new SURL for the given GUID.
- { GUID, LFN }
  Register a new LFN for the given GUID.
- { LFN, SURL }
  Register a new SURL for the given LFN.
- { LFN, LFN }
  Register a new LFN for the given LFN.
- { SURL, LFN }
  Register a new LFN for the given SURL.
- { SURL, SURL }
  Register a new SURL for the given SURL.

Note:

- The first item in each pair must exist in the catalog, and the second item must *not* exist.

closeCollectiveRegistration

This function has to be called after the function "register". Since "register" can be called multiple times for a given requestID, the RRS needs to be notified about the last registration call.

IN: String userID,
String requestID

OUT: String statusCode,
String statusExplanation

getRegistrationStatus

Provides status information about the file reference registrations.

IN: String userID,
String requestID,
Int entryOffset, // default: 0
Int numberOfEntries // default: total number of entries in the request

OUT: { { String GUID | String LFN | String SURL }, // first file reference
String actionCode,
String errorCode,

{ String LFN | String SURL }, // second file reference
String actionCode,
String errorCode,
String fileStatusCode,
String statusExplanation }[]

Parametes

- actionCode ::= "given" | "toBeRegistered" | "generated".
- fileStatusCode ::= "done" | "inProgress" | "pending" | "error".
- errorCode ::= "null" | "alreadyExists" | "notFound".
  (The errorCodes "alreadyExists" and "notFound" are only given if fileStatusCode returns "error"; otherwise the errorCode is "null")
- FileStatusCode::= "in progress" | "done" | "catalogDown" | "failed".

Note:
The action codes are provided in order to return the requested action on the file reference as originally submitted. The errorCode is therefore associated with each file reference. For example, suppose that a first time registration of a {GUID, LFN} is: {guid://34567, lfn://tmp/foo}, and the LFN exists - an error. The returned status is {guid://34567, toBeRegistered, --, lfn://tmp/foo, toBeRegistered, alreadyExists, error).
A second example. Consider the first time registrationof the form { --, LFN}. the GUID is not provided, and the system generated it. Suppose that the registration request is { --, lfn://tmp/foo}, and the GUID guid://54321 was generated. The returned status is {guid://54321, generated, --, lfn://tmp/foo, toBeRegistered, --, done).
A third example. Suppose that a subsequent registration of a {LFN, SURL} that was not registered yet is: {lfn://tmp/foo, gridftp://cs.lbl.gov/home/foobar}. The return status is: {lfn://tmp/foo, given, --, gridftp://cs.lbl.gov/home/foobar, toBeRegistered, --, pending}.

getRegistrationSummary

Provides only summary status information about a particular registration request. Detailed file status information can be retrieved with getRegistrationStatus described above.

IN: String userID,
String requestID

OUT: Int numOfRegisteredFiles,
Int numOfFailedRegisteredFiles,
Int numOfPendingFiles,
String statusCode,
String statusExplanation

Parameters

- numOfRegisteredFiles ::= number of file references that were registered in the catalog for a given requestID.

- numOfFailedRegisteredFiles ::= number of files that failed to get registered in the catalog for a given requestID.
- numOfPendingFiles ::= number of file references that are still to be registered in the catalog for a given requestID.
- statusCode ::= "completed" | "inProgress" | "aborted" | "partiallyUnregistered" | "unregistered" | "catalogDown".

Note: The "partiallyUnregistered" code is used to specify that some of the files were unregistered for this request. The "unregistered" code is used to specify that the entire request was unregistered after the registration was closed.

abortRegistration

This function can be called anytime after the openCollectiveRegistration, and possibly after one or more "register" calls were made. The registration request represented by requestID is stopped and all registered file references are unregistered from the catalog, and the pending file reference registrations are removed.

IN: String userID,
String requestIDIn

OUT: String requestIDOut

Parameters

- requestIDIn ::= requestID of the previously called register function
- String requestIDOut ::= output requestID. This requestID is used for retrieving the status with getAbortRegistrationStatus.

getAbortRegistrationStatus

IN: String userID,
String requestID

OUT: Int numOfUnregisteredFiles,
Int numOfPendingFiles,
String statusCode,
String statusExplanation

Parameters

- numOfUnregisteredFiles ::= number of files that were unregistered from the catalog or removed from the RRS queue for a given requestID.
- numOfPendingFiles ::= number of files that are still to be unregistered from the catalog for a given requestID.

unregister

Unregister one or more files from the catalog. If only a requestID is specified then all registrations that were previously made by the registration function with this requestID get unregistered. Otherwise, specific pairs of file references can be specified to be unregistered. For more details on the behavior of the unregister function, see introductory section.

```
IN:     String userID,
        String requestIDIn,
        Boolean unregisterCollectiveRequest // unregister entire request, default: false
        { { String GUID | String LFN | String  SURL }, // first file reference (given)
          { String LFN | String SURL },                // second file reference
                                                       // (toBeUnregistered)
          Boolean unregisterAllReferences              // default: false
        }[]

OUT:    String requestIDOut,
        String statusCode,
        String statusExplanation
```

Parameters

- requestIDIn ::= requestID of the previously-called register function
- String requestIDOut ::= output requestID. This requestID is used for retrieving the status with getUnregisterStatus.
- unregisterAllReferences ::= if only a GUID is provided as a file reference (i.e. {GUID, --}) then this flag has to be set. Can also be set if only an LFN or SURL are provided (i.e. {--, LFN} or {--, SURL}), in order to completely remove the file from the catalog and all its references.
- unregisterCollectiveRequest ::= for a request specified only with a requestIDIn, all files get unregistered.

Use Cases

- { requestIDIn }
  The flag unregisterCollectiveRequest must be set or an error is returned. All files, that were previously registered with the specified requestID, get unregistered.

The following uses cases can be specified with or without a requestID. In case no requestID is specified, files get unregistered without taking into account previous requests.

Pairs of file reference cases:

- { GUID, LFN }
  For a given GUID, the LFN gets unregistered.
- { GUID, SURL }
  For a given GUID, the SURL gets unregistered.
- { LFN, SURL }

For a given LFN, an SURL gets unregistered.

- { LFN, LFN }
  For a given LFN, an LFN gets unregistered.
- { SURL, LFN}
  For a given SURL, an LFN gets unregistered.
- { SURL, SURL }
  For a given SURL, an SURL gets unregistered.

Single file reference cases:

- { GUID, -- }
  The flag unregisterAllReferences must be set or an error is returned. By unregistering the GUID, all SURLs and LFNs are also unregistered. This function can be called after register to remove the file registration and all its associated references.
- { --, LFN }
  Unregister a specific LFN. If the flag unregisterAllReferences is set, the file is completely removed from the catalog.
- { --, SURL }
  Unregister a specific SURL. If the flag unregisterAllReferences is set, the file is completely removed from the catalog.
- { LFN, -- } and { SURL, --) are not allowed.

getUnregisterStatus

The structure of the parameters of this function is similar to the structure of the getRegisterStatus function. The only differences is that the actionCode has the value "toBeUnregistered", and the errorCode can have only the value "notFound".

```
IN:     String userID,
        String requestID,
        Int entryOffset, // default: 0
        Int numberOfEntries // default: total number of entries represented by requestID

OUT:    { { String GUID | String LFN | String  SURL }, // given file references
          String actionCode,
          String errorCode,
          { String LFN | String SURL },  // file references to be unregistered
          String actionCode,
          String errorCode,
          String fileStatusCode,
          String statusExplanation
        }[]
```

Parameters

- actionCode ::= "given" | "toBeUnregistered".

- fileStatusCode ::= “done” | “inProgress” | “pending” | “error”.
- errorCode ::= “null” | “notFound”.
  (The errorCode “notFound” is only given if fileStatusCode returns “error”; otherwise the errorCode is “null”)
- FileStatusCode::= “in progress” | “done” | “catalogDown” | “failed”.

Note:
An example: Suppose that an unregistration of a {LFN, SURL} that completed successfully has the file references: {lfn://tmp/foo, gridftp://cs.lbl.gov/home/foobar}. The return status is: {lfn://tmp/foo, given, --, gridftp://cs.lbl.gov/home/foobar, toBeUnregistered, --, done}.

getUnregisterSummary

IN: String userID,
String requestID

OUT: Int numOfUnregisteredFiles,
Int numOfFailedUnregisteredFiles,
Int numOfPendingFiles,
String statusCode,
String statusExplanation

Parameters
- numOfUnregisteredFiles ::= number of files that are unregistered from the catalog for a given requestID.
- numOfFailedUnregisteredFiles ::= number of files that failed to get unregistered from the catalog for a given requestID.
- numOfPendingFiles ::= number of files that are still to be unregistered from the catalog for a given requestID.

Discovery Functions

getFileReferences
Given a particular file reference, this function specifies the file references that should be returned. For example getFileReferences specified as a (GUID, LFN), will return all the LFNs for that GUID.

IN: String userID,
{ { String GUID | String LFN | String SURL }, // given file reference
{ String GUID | String LFN | String SURL } // requested file references
}[]

OUT: String requestID

Additional examples. getFileReference for:

- (GUID, SURL) should return all SURLs for the given GUID.
- (LFN, LFN) should return all LFNs for the given LFN (including itself).
- (LFN, SURL) should return all SURLs for the given LFN.

getFileReferenceStatus

Returns for a given file reference, all the requested file references

IN: String userID,
String requestID,
Int entryOffset, // default: 0
Int numberOfEntries // default: total number of entries represented by requestID

OUT: { { String GUID | String LFN | String SURL },
{ String GUID | String LFN | String SURL }[],
String statusCode,
String statusExplanation
}[]

getSystemInfo

Retrieve information about the setup of the RSS. These parameters are specified in a configuration file at deployment time of the RRS.

IN: String userID

OUT: String catalogEndpoint,
Int lifeTimeOfRequestID,
Int lifeTimeOnCatalogRetry

Parameters

- catalogEndpoint ::= endpoint of the catalog.
- lifeTimeOfRequestID ::= minimal amount of time a requestID is kept by the RRS. Note that if the life time of a requestID expires, it cannot be used for subsequent status requests.
- lifeTimeOnCatalogRetry ::= length of time the RRS retries to execute a request if a catalog is down.

Name Space Functions

makeDirectory

Make a directory and register in the catalog.

IN: String userID,
String LFN

OUT: String statusCode,
String statusExplanation

Parameters

- LFN ::= directory path.

Use Case

Assume that a client wants to create the following hierarchical directory structure of LFNs:

lfn://cms
lfn://cms/higgsCandidate/
lfn://cms/higgsCandidate/run1
lfn://cms/higgsCandidate/run1/file1
lfn://cms/higgsCandidate/run1/file2
lfn://cms/higgsCandidate/run2
lfn://cms/higgsCandidate/run2/file1
lfn://cms/higgsCandidate/run2/file2

In this case the LFNs at the lowest level of this hierarchy (leaf nodes), namely "lfn://cms/higgsCandidate/run1/file1", "lfn://cms/higgsCandidate/run1/file2", etc. are logical references for files. The others are logical references for directories.

removeDirectory

Remove a directory from the catalog.

IN: String userID,
String LFN

OUT: String statusCode,
String statusExplanation

Parameters

- LFN ::= directory path

makeLink

Make a symbolic link from an LFN to another LFN. This function is similar to the Unix command ln –s.

IN: String userID,
{ String LFN | String SURL }, // target LFN for symbolic link
String LFN // name of the symbolic link

OUT: String statusCode,
String statusExplanation

Use Case

Assume that client A has created the following directory structure of LFNs:

lfn://cms
lfn://cms/higgsCandidate/
lfn://cms/higgsCandidate/run1
lfn://cms/higgsCandidate/run1/file1
lfn://cms/higgsCandidate/run1/file2
lfn://cms/higgsCandidate/run2
lfn://cms/higgsCandidate/run2/file1
lfn://cms/higgsCandidate/run2/file2

Now assume that client B wants the create a symbolic link from his directory structure to the directory structure of client A:

lfn://atlas
lfn://atlas/higgs
lfn://atlas/higgs/cmsResults

The function for creating the symbolic link "lfn://atlas/higgs/cmsResults" that refers to "lfn://cms/higgsCandidate" is as follows:

makeLink(userID="JohnDoe",lfn://cms/higgsCandidate, lfn://atlas/higgs/cmsResults)

listDirectory

List the directory information for a given LFN.

IN: String userID,
String LFN,
Boolean recursive // default: false

OUT: String requestID

getListDirectoryStatus

Note that the returned list can be long, and therefore the entryOffset and the numberOfEntries can be specified.

IN: String userID,
String requestID,
Int entryOffset, // default: 0
Int numberOfEntries // default: total number of entries represented by requestID

OUT: String LFN,
{ String LFN | String SURL }[],
String statusCode,
String statusExplanation

Attribute Functions

getFileAttributes

IN: String userID,
{ String GUID | String LFN | String SURL }[],
String attributeCategory

OUT: String requestID

Parameters

- attributeCategory ::= "core" | "all".

Note

- Currently the core attributes are defined as "checkSumType", "checkSumValue" and "fileSize". In future RRS versions these attributes could be extended.

getFileAttributesStatus

This function is provided in order to get the file attributes necessary for verifying the correctness of a file. It can also be used to get all the attributes in a (attributeName, attributeType, attributeValue) format. For more details, see introductory section.

IN: String userID,
String requestID,

Int entryOffset, // default: 0
Int numberOfEntries // default: total number of entries represented by requestID

OUT: { { String GUID | String LFN | String SURL },
String attributeName,
String attributeType,
String attributeValue }[],
String statusCode,
String statusExplanation

Parameters

- attributeName ::= "checkSumType" | "checkSumValue" | "fileSize".
- attributeType ::= "Integer" | "Real" | "String".

Note

- Whether a GUID, LFN or SURL is returned, depends on what was defined in the respective getFileAttributes request. For instance, if the respective request was getFileAttributes(userID, LFN[], attributeCatagory="all"), then for each LFN, all attributes are returned

insertFileAttributes

IN: String userID,
{ { String GUID | String LFN | String SURL },
String attributeName,
String attributeType,
String attributeValue }[]

OUT: String statusCode,
String statusExplanation

getDirectoryAttributes

IN: String userID,
String LFN

OUT: String requestID

getDirectoryAttributesStatus

IN: String userID,
String requestID,
Int entryOffset, // default: 0
Int numberOfEntries // default: total number of entries represented by requestID

OUT: { String LFN,
String attributeName,
String attributeType,
String attributeValue }[],
String statusCode,
String statusExplanation

insertDirectoryAttributes

IN: String userID,
{ { String GUID | String LFN},
String attributeName,
String attributeType,
String attributeValue }[]

OUT: String statusCode,
String statusExplanation