POOL FileCatalog Documentation

Authors: Zhen Xie, Maria Girone
Date: 10 September 2003
Version: 0.8
Template Version: 1.5

Document Status Sheet

Title:	Pool File Catalog Component Description	
ID:		
Version	Date	Reason for change
0.1	22 Aug 02	New document
0.2	10 Nov 02	MySQL and XML implementations descriptions
0.3	18 Dec 02	EDG implementation and high level test description
0.4	27 Feb 03	Interface and command-line tool changes. Add more use-cases.
0.5	9 May 03	POOL-1-0-0 release
0.6	4 June 03	Implement consistent transaction protocol for all backends
0.7	24 June 03	POOL-1-1-0 release
0.8	10 Sept 03	Add Python interface to the component

1 Description of the component

1.1 Purpose of the component

The file catalog component in POOL is responsible for maintaining consistent list of accessible files together with their unique and immutable file Ids. Its main user is the storage components who consult the file catalog when a new file is to be accessed.

The file catalog is also used to store some file related metadata as for example the logical filename which in contrast to the file ID may contain a memorisable text string.

The relationship between the file Ids and file names is the following:

When a file is created for the first time, it is assigned to a unique and immutable file Id and it is also assigned to a physical file name which identifies its physical location. Later on, different copies of the file may be created. Each replica of the file has its own distinct physical file name but the same file Id as its master copy. In another word, FileID is the logical identifier of a file and all its replicas. Due to the generation mechanism, the format of the FileID is not easy for user to read and to remember, as a consequence, human readable and memorisable alias may be provided to a FileID. These alias are called logical file names. This view is consistent with that of the replica management service within the context of the EU Data Grid (EDG) Project[1].

The file catalog component provides an interface to two types of users. First of all, it provides an interface for the storage components in POOL to register and lookup a file inside the application process. Command line tools are provided to handle catalog operations outside the application process, as assigning Logical File Names, registering files in the catalog, appending one catalog to the other, etc.

Three different implementations are provided:

- An XML-catalog can be used and/or produced by a single user inside one job. It is useful when user wants to run the application disconnected from the network. The content of a XML catalog or a part of it can be published to the other two types of catalogs. A part of the other two types of catalogs can be extracted into a XML catalog.

- A native MySQL catalog can be used in a production farm. It can handle multiple users and multiple jobs. However, it is not on the Grid. The content of the MySQL catalog or a part of it can be published to the Grid-aware catalog. A part of the other two types of catalogs can be extracted into a MySQL catalog.

- EDG-RLS based catalog is used by the entire Virtual Organization(VO). The EDG Project will provide the Replica Management Service, which controls files that belong to a VO. In particular, the Replica Location Service (RLS) component [2] maintains information about the physical locations of files, while the Replica Metadata Catalog (RMC) component [3] provides the information on the logical file names and metadata. The pool file catalog component provides an interface to the EDG-RLS and EDG-RMC for the Grid-aware applications.

1.2   Known problems and restrictions

The details of the interaction between the file catalog and grid components still need to be defined in more detail. In the version 1.0 release of the component we still make a few simplifying assumptions to get started. Any or all of those may not hold in the longer term

1) POOL will not directly create file replicas and assumes to use the first replica of each file.

2) POOL will assume that in case of several existing replicas for a given GUID the first one (master copy) can be used for writing/appending

3) We do not yet check authorization on the catalog level. (EDG RLS does not provide this functionality yet).

4) The implementation of the EDG container is still under development due to some missing feature of the API interface in use in this version. As a result, the cache size of the container will be the limit on the number of entries in the container.

Limits of MySQL database: limit on the length of the data type varchar is 255. This restricts the length of physical and logical filenames in the current implementation.

1.3   Repository of the component

/pool/FileCatalog   repository of the catalog interface and common utility classes code
/pool/MySQLCatalog   repository of MySQL catalog code
/pool/XMLCatalog   repository of XML catalog code
/pool/EDGCatalog   repository of EDG catalog code
/pool/Utilities/FileCatalog   repository of catalog command-line tools source code
/pool/Scripts/FileCatalog   repository of scripts used by file catalog
/pool/PyFileCatalog   repository of the Python interface of the component

2   User guide

2.1   How to construct the catalog contact string

To obtain the connection to the catalog, a contact string of the format:

[prefix_][protocol]://[username]:[password]@[host]:[port]/[path]

or

[prefix_]file:path

The **[prefix_]** field is used to distinguish different catalog implementations. In case of absence, a local XMLCatalog will be used.

The supported prefix are: **xmlcatalog_** , **mysqlcatalog_, edgcatalog_**

The supported protocols are: **mysql** for MySQL catalog; **http** for XML and EDG catalog; **ftp** for XML catalog; **file** for XML catalog.

Some examples of the contact strings for different catalogs are shown as follows:

- **MySQL**:

    mysqlcatalog_mysql://@lxshare070d.cern.ch:3306/testFCdb

For the MySQL catalog, the **[path]** field represents the database name. The **[username]** field should be the username of the database. In case of absence, the login name of the user will be taken. The default value for the **[port]** field is 3306.

- **XML**:

    xmlcatalog_file:/tmp/FileCatalog.xml

    file:/tmp/FileCatalog.xml

    xmlcatalog_http://pc01.cern.ch/file001, if the catalog is at remote site and read only

- **EDG:**

    edgcatalog_http://rlstest.cern.ch:7777/edg-replica-location/services/edg-local-replica-catalog

2.2   How to construct the query string

The component supports query on the file metadata. In POOL_1_0_0 release, the query is a plain string consists of the attribute, "=" or "like" predicates and the desired value of the attribute. The wildcard "%" on the attribute value is allowed. Due to the string implementation of the XML and EDG catalogs, numerical queries are not supported in this release. All the string values must be quoted within a pair of single quotes. Example of some query strings: "jobid='sim101'", "owner like '%me%'"

The query strings can be passed to the command-line tools using the **–q** option or passed to the catalog API as argument of the lookup methods.

The query attribute can be either the metadata or 'pfname', 'lfname' and 'guid'.

FileID(GUID) is and should not be explicitly defined as an attribute because it is implicitly defined when the metadata schema is created. It is invisible to the user.

In this release only 'AND' logic is supported by all implementations, e.g. "jobid='sim101' AND owner line '%me%'".

2.3 How to use command-line tools of the component

The command-line tools provided by the File Catalog are in the */pool/Utilities/FileCatalog* repository.

General options:

-h print help message

-u the catalog contact string. If absent, the contact string is picked up from the environment variable **POOL_CATALOG**. The contact string specified by –*u* option overrides that taken from the environment variable.

-l LFN

-p PFN

-m customized cache size when using the catalog container, if this option is not given, the default cache size 1000 is assumed.

1. Register PFN

FCregisterPFN –p pfname [-F -u uri -h]

By default, this command register a PFN without assign a unique FileID to it. This is useful when one wants to pre-allocate a PFN to a not-yet-existent physical file. Later on, one can register the pre-allocated PFN from inside the job.

–F option : force the real file registration from the command-line: a FileID is generated and registered in the catalog with the given PFN.

2. Register LFN

FCregisterLFN –p pfname –l lfname [-u uri -h]

Register the LFN (specified by **–l** option) to the PFN (specified by **–p** option).

3. Register a replica file name

FCaddReplica –p pfname –r replica [-u uri -h]

Register a replica file name (specified by **–r** option) to the PFN (specified by **–p** option).

4. Lookup PFNs

FClistPFN [-l lfname –q query –m cachesize –u uri -h]

-l option: list all PFNs with given LFN

-q option: list all PFNs satisfy the query on file metadata

If no option is given, all PFNs are displayed.

5. Lookup LFNs

FClistLFN [-p pfname –q query –m cachesize –u uri -h]

-p option: list all LFNs with given PFN

-q option: list all LFNs satisfy the query on file metadata

If no option is given, all LFNs are displayed.

6. Lookup Meta Data

FClistMetaData [-l lfname –p pfname –q query –u uri –m maxcache –h]

-l option: list metadata associated with the file with given PFN.

-p option: list metadata associated with the file with given LFN.

If no option is given, all metadata entries are displayed.

7. Describe the file meta data definition

FCdescribeMetaData [-u uri -h]

Describe meta data in the catalog.

Format of the output:

((attribute1_name, attribute1_type), (attribute2_name, attribute2_type))

8. Define the meta data specification

FCdefineMetaData [-d metadatadefinition –u uri –h]

Create meta data specification specified by the **–d** option.

Format of the input:

"(*(attribute1_name, attribute1_type), (attribute2_name, attribute2_type))"*

9. Insert meta data

FCaddMetaData   [-p pfname –l lfname -m cachesize –u uri -h]

Insert file meta data associated with file with given PFN (specified by **–p** option) or with given LFN(specified by **–l** option).

The format of the input:

"((attribute1_name, attribute1_value), (attribute1_name, attribute2_value))"

10. Delete a PFN or LFN entry

FCdeleteEntry [–p pfname –l lfname -u uri -h]

Delete the PFN specified by **–p** option; delete the LFN specified by **–l** option.

If the PFN is the last one associated with the File, all the LFNs associated with the file are deleted as well

If the LFN is the last one associated with the file, the operation will not affect the associated PFN and replica information.

11. Clean up the catalog deleting file registrations by unsuccessful jobs

FCclearUnsuccessful [-u uri –h]

12. Extract a fragment from the source catalog and attach it to the destination catalog

FCpublish -d destinationcatalog [-p pfname -l lfname –q query –m cachesize -u sourcecatalog -h]

The destination catalog is specified by the **–d** option.

-u option: the source catalog contact string. If not specified, POOL_CATALOG value will be taken.

-l option: extract/publish catalog fragment associated with given LFN .

-p option: extract/publish catalog fragment associated with given PFN.

-q option: extract/publish catalog fragment selected by given query on file metadata.

If no query is specified, the entire source catalog will be appended to the destination catalog. The operation is atomic.

13. Rename PFN

FCrenamePFN –p pfname –n newpfname [-u sourcecatalog -h]

Rename the PFN (specified by the –p option) to the new one (specified by the –n option).

2.4   C++ API of the component

Class **IFileCatalog** is the interface of the component. It provides functions of the following types:

- Connection and transaction control functions
- Catalog insertion and update functions
- Catalog lookup functions
- Cross catalog operations
- Catalog entries deletion functions

The catalog has two transaction states: in transaction and between transactions. Transaction starts with start() and ends with commit() or rollback(). Methods start() and commit() or rollback() should always be called in pairs. Exceptions will be thrown if these methods are not in pairs. Commit() methods takes IFileCatalog::CommitMode as arugment. REFRESH

mode indicates the XML parser (for the XML catalog) will be reinitialised at the next start() method while ONHOLD mode indicates that parser states will not be changed at the next start() method. The default value of the argument is the REFRESH mode.

Between connect() and start(), start() and disconnect() are the between transaction states. Exceptions will be thrown when catalog operations are called at the between transaction states. Methods connect() and disconnect() also should be called in pairs. Exceptions will be thrown if connect() or disconnect() method is called twice in a row.

User can register PFN and LFN of a file or add a replica file name to a registered file. There are two states of PFN registration: fully-registered and pre-registered. Fully-registered state indicates that the physical file actually exists; pre-registered state indicates that the file doesnot exist physically, but PFN is registered as a place holder. Adding a replica file name to a PFN in pre-registered state is not allowed. The pre-registered state can be updated to fully-registered state. Bulk insert of PFNs and LFNs are also supported.

User can lookup PFN(s) by given FileID through methods lookupBestPFN(), lookupPFN(); by given LFN through method lookupPFNByLFN() ; or LFN or by a query on the metadata through method lookupPFNByQuery(). If the query is an empty string all PFNs in the catalog will be returned. In the current release, the lookupBestPFN method returns the first PFN found. Similar functions are provided for LFN lookups. One can also lookup FileID by PFN or LFN.

The component supports associating metadata with the file. The purpose of the metadata is to ease the file lookup and the catalog fragment selection. However, the assumption is that the meta data schema can be defined once with one catalog. If the metadata schema is updated, the old schema and the old metadata will be lost. Only catalogs with the same metadata definition can cross populate each other. The metadata can be defined through method createMetaDataSpec(). Metadata insertion and lookup methods are also provided.

User can import a fragment of another catalog into the current catalog through importCatalog() method. The two catalogs may have different backends. The selection of the catalog fragment is by querying on the file metadata. If an empty string is passed, the entire catalog will be appended to the current catalog.

The interface provides method to delete LFN and PFN entries in the catalog. However, one should use these methods with caution, especially when the catalog is shared by more than one user.

Class **IFCContainer** provides an interface to iterate on catalog entries. It has the combined functionality of a container and an iterator. Only sequential iteration is support through the method hasNext() and Next(). For scalability reason two modes of iterating are supported: retrieve results into a cache in memory or retrieve item one by one directly from the catalog

backend. The cache size is defined by the user and the default value is 1000 entries. The cache is used repeatedly until all results are retrieved, new batch of entries will overwrite old entries in the cache. When the cache size is set to 0, the one-by-one mode is switched on. The cache size and working mode of the iterator can be changed through the reset() method.

Each container is bound to a given filecatalog. Containers are created through the catalog interface. Note, user is responsible for deleting the containers created by the catalog getContainer() methods.

Class **FCSystemTools** is a helper class collecting common utilities functions. In particular, the createCatalog() function encapsulates the usage of the seal::PluginManager to load concrete file catalogs. Different catalogs are loaded according to the prefix of the catalog contact string. If an empty string is passed to the method, a default XMLCatalog is created. Another helper class **URIParser** can be used to parse the catalog contact string into two parts: prefix which can be used to create the catalog and url which is can be passed to the catalog connection method. If a contact string is not passed to the URIParser constructor, the parser tries to pick it from the environment variable POOL_CATALOG. The purpose of these helper classes are to ease the catalog creation. They do not provide essential catalog functionalities.

An example of application code is shown below:

```
URIParser p("xmlcatalog_file:catalog.xml");

p.parse();

IfileCatalog* mycatalog=FCSystemTools::createCatalog(p.prefix());

mycatalog->connect(p.url());

IfileCatalog::FileID fid;

mycatalog->start();

mycatalog->registerFile("aPFN","fileformat",fid);

mycatalog->registerFilename("aPFN","lfn:aPFN");

mycatalog->commit(IFileCatalog::ONHOLD);

std::string bestpfn, filetype;

mycatalog->start();

mycatalog->lookupBestPFN(fid, IFileCatalog::READ, IFileCatalog::SEQUENTIAL,
bestpfn, filetype);

mycatalog->commit(IFileCatalog::ONHOLD);

mycatalog->start();

IPFNContainer* mypfs=mycatalog->getPFNContainer(100);

mycatalog->lookupPFNByQuery("",*mypfs);

while(mypfs->hasNext()){

    std::cout<<mypfs->Next()<<std::endl;
```

```
    }

    delete mypfs;

    mycatalog->commit();

    mycatalog->disconnect();

    delete mycatalog;
```

2.5   Python interface of the component

The component provides a Python interface which allows the catalog operations to be called from a Python script instead of a compiled C++ application. The name of the Python extension module is **FileCatalog** which contains two Python classes: *FileCatalog* and *Container*. The module can throw its own exception: *PyFileCatalogError*.

The module can be imported if the $POOLProject/$arch/lib is added in the PYTHONPATH. This can be set with `eval `scram runtime –csh`` in a SCRAM configured environment.

Following the common style of Python extension modules, this module is self-documenting. From the Python interpreter one can use `dir(FileCatalog.FileCatalog)`, `dir(FileCatalog.Container)` to see the available methods and `help(FileCatalog.FileCatalog.method)` to see the usage of the method. All the methods provided by the C++ API of IFileCatalog class and IFCContainer class are available in Python with the same name.

The module defines the following constants in the module scope which behave as enum type in C++ to be used as function arguments:

REFRESH	(argument of the commit() method)
ONHOLD	(argument of the commit() method)
PREREGISTER	(argument of the registerFile() method)
REGISTER	(argument of the registerFile() method)
LFN	(argument of the Container() constructor)
PFN	(argument of the Container() constructor)
META	(argument of the Container() constructor)

Due to the language difference the Python module has different way of creating a catalog container with respect to the C++ API. E.g.

```
pfns=Container(a_catalog_instance,  PFN, cache_size)
```

Besides, Python methods support default arguments and keyword arguments: e.g. the following calls are also legal

```
pfns=Container(a_catalog_instance,  PFN)    (default cache size=1000)

pfns=Container(a_catalog_instance,  "cache size"=120, "container type"=PFN)
```

Note: due to some problems with the seal plug-in manager, one has to reset the dlopen flag before importing the module:

```
import sys, DLFCN

sys.setdlopenflags( DLFCN.RTLD_NOW | DLFCN.GLOBAL )

from FileCatalog import *
```

More examples can be found in the tests area of the component
/pool/PyFileCatalog/tests

3 Detailed C++ API of the component

FileCatalog component depends on the following POOL components

/pool/POOLCore

/pool/AttributeList

All classes are defined in the *pool* namespace.

3.1 Public interfaces

Interfaces

IFileCatalog Class

connect()	**This method establishes the connection to the catalog backend using a url.** **Exception is thrown in case of problems.**
	Syntax: void connect(const std::string& url)
disconnect()	**This method disconnect from the catalog backend.** **Exception is thrown in case of problems.**
	Syntax: void disconnect() const
start()	**This method starts the catalog transaction. Exception is thrown**
	Syntax: void start() const
commit()	**This method commits the catalog transaction. Exception is thrown if the operation cannot be committed.** **CommitMode can be IFileCatalog::REFRESH or IFileCatalog::ONHOLD. The default value is REFRESH.**
	Syntax: void commit(const CommitMode) const
rollback()	**This method rollbacks the catalog transaction. Exception is thrown if the opertation cannot be rolled back.**
	Syntax: void rollback() const
registerFile()	**This method register a file with given PFN and return the corresponding FileID and type of the file from the argument list. The default mode IFileCatalog::REGISTER is to fully register the file, meaning the physical file with given PFN really exists. Alternatively, in IFileCatalog::PREREGISTER mode, the physical file doesnot exist, the registration in the catalog is only a placeholder. The file can be fully registered later from the application process. Exception is thrown if PFN is already registered.**
	Syntax: void registerFile(const std::string& pfn, const std::string& filetype, FileID& fid, const FileStatus fsta=IFileCatalog::REGISTER) const

registerFilename()	**This method register the LFN of a file with given PFN.**
	Syntax: void registerFilename(const std::string&pfn, const std::string&lfn) const
lookupBestPFN()	**This method returns a PFN associated with the given FileID. The first PFN found is returned. The file type is also returned. FileOpenMode and FileAccessPattern are passed as a hint to the Grid components for file transfer.**
	Syntax: void lookupBestPFN(const FileID& fid, const FileOpenMode& omode, const FileAccessPattern& amode, std::string& pf, std::string& filetype) const
lookupPFN()	**This method returns PFNs associated with given FileID.**
	Syntax: void lookupPFN(const FileID& fid, IPFNContainer& pfs) const
lookupLFN()	**This method returns LFNs associated with given FileID.**
	Syntax: void lookupLFN(const FileID& fid, ILFNContainer& lfs) const
lookupPFNByQuery()	**This method returns PFNs satisfy the query on the metadata.**
	Syntax: void lookupPFNByQuery(const std::string& query, IPFNContainer& pfs) const
lookupLFNByQuery()	**This method returns LFNs satisfy the query on the metadata.**
	Syntax: void lookupLFNByQuery(const std::string& query, ILFNContainer& lfs) const
lookupMetaDataByQuery()	**This method returns meta data selected by the query.**
	Syntax: void lookupMetaDataByQuery(const std::string& query, ImetaDataContainer& metas) const
lookupPFNByLFN()	**This method returns PFNs associated with given LFN**
	Syntax: void lookupPFNByLFN(const std::string& lfn, IPFNContainer& pfs) const
lookupLFNByPFN()	**This method returns LFNs associated with given PFN**
	Syntax: void lookupLFNByPFN(const std::string& pfn, ILFNContainer& lfs) const
lookupFileByPFN()	**This method returns the FileID and file type with given PFN**
	Syntax: void lookupFileByPFN(const std::string& pfn, FileID& fid, std::string& filetype) const
lookupFileByLFN()	**This method returns the FileID with given LFN**
	Syntax: void lookupFileByLFN(const std::string& lfn, FileID& fid) const
addReplicaFilename()	**This method adds a replica file name to given PFN.**
	Syntax: void addReplicaFilename(const std::string& pfn, const std::string& replicapfn) const
renamePFN()	**This method replaces a given PFN with a new PFN.**
	Syntax: void renamePFN(const std::string& pfn, const std::string& newpfn) const

insertPFNs()	**This method inserts a group of PFNs into the catalog.**
	Syntax: void insertPFNs(IPFNContainer& pfs) const
insertLFNs()	**This method inserts a group of LFNs into the catalog.**
	Syntax: void insertLFNs(ILFNContainer& lfs) const
importCatalog()	**This method appends a fragment of the given source catalog to the current catalog. The fragment is selected by query on the source catalog metadata. If the query string is empty, entire source catalog is appended to the current catalog. One can specify the default cache size for this operation. The default value is 1000.**
	Syntax: void importCatalog(IfileCatalog* fc, const std::string& query, unsigned int cachesze=FCDEFAULT_CACHE_SIZE) const
deleteLogicalFilename()	**This method deletes the specified LFN.**
	Syntax: void deleteLogicalFilename(const std::string& lfn) const
deletePhysicalFilename()	**This method deletes the specified PFN. If the PFN is the last one associated with a file, all associated LFN and metadata are deleted as well.**
	Syntax: void deletePhysicalFilename(const std::string& pfn) const
clearUnSuccessful()	**This method deletes catalog entries generated by unsuccessful jobs.**
	Syntax: void clearUnSuccessful() const
isReadOnly()	**This method checks if the catalog is read-only to the user.**
	Syntax: bool clearUnSuccessful() const
createMetaDataSpec()	**This method create the metadata definition of the catalog.**
	Syntax: void createMetaDataSpec(AttributeListSpecification& spec) const
insertAttributes()	**This method inserts metadata associated with given FileID .**
	Syntax: void insertAttributes(const FileID& fid, AttributeList& attrs) const
getMetaDataSpec()	**This method retrieves the meta data definition of the catalog.**
	Syntax: void getMetaDataSpec(AttributeListSpecification& spec) const
getPFNContainer()	**This method creates a PFNContainer with given cache size. The default cache size is 1000 PFN entries. User is responsible for deleting the created container.**
	Syntax: IPFNContainer* getPFNContainer(unsigned int cachesize=FCDEFAULT_CACHE_SIZE) const
getLFNContainer()	**This method creates a LFNContainer with given cache size. The default cache size is 1000 LFN entries. User is responsible for deleting the created container.**
	Syntax: ILFNContainer* getLFNContainer(unsigned int cachesize=FCDEFAULT_CACHE_SIZE) const
getMetaDataContainer()	**This method creates a MetaDataContainer with given cache size. The default cache size is 1000 meta data entries. User is responsible for deleting the created container. Null pointer is returned if the catalog has no defined meta data.**

	Syntax: IMetaDataContainer* getMetaDataContainer(unsigned int cachesize=FCDEFAULT_CACHE_SIZE) const
sameMetaDataSchema()	**This method checks if two catalogs have the same schema.**
	Syntax: bool sameMetaDataSchema(IfileCatalog* fc) const

- **template class< typename Item> IFCContainer Class**

IFCContainer()	**The constructor creates an instance of the container bounded to a given catalog and initialised with given cachesize.**
	Syntax: IFCContainer(unsigned int cachesize, IFileCatalog* catalog)
reset()	**This method resets the cache size and the iterator to the initial value.**
	Syntax: void reset(unsigned int cachesize)
hasNext()	**This method tells if there is next entry in the container.**
	Syntax: bool hasNext()
Next()	**This method retrieves the next item from the container. If the cache size is set to 0, it retrieves one result directly from the catalog backend.**
	Syntax: Item& Next()
size()	**This method tells the current size of the container. Warning: if the cache size is set to 0, this method always returns 0. It cannot tell the total number of items have been returned.**
	Syntax: unsigned int size() const
max_size()	**This method tells cache capacity of the container**
	Syntax: unsigned int max_size() const

- **FCSystemTools Class**

createCatalog()	**This method creates a catalog instance from the prefix of the catalog contact string. If empty prefix is specified, a XML catalog instance is created. User is responsible to delete the instance.**
	Syntax: static IFileCatalog* createCatalog(const std::string& prefix)
GetEnv()	**This method gets the specified environment variable**
	Syntax: static const char* GetEnv(const char* key)
FileExists()	**This method if the specified file exists**
	Syntax: static bool FileExists(const char* filename)

3.2 Exceptions generated

All exceptions thrown from the File Catalog components are either seal::Exception or derived from it. Explicit catalog exceptions are used for uniform exception message printing purpose throughout catalog subcomponents. User should catch seal::Exception

The explicit file catalog exceptions are:

FCduplicateLFNException	thrown when attempting to register a LFN which already exists in the catalog
FCduplicatePFNException	thrown when attempting to register a PFN which already registered in the catalog
FCnonexistentFileException	thrown when attempting to register the replica of a nonexistent physical file
FCbackendException	this exception propagates exceptions thrown by the backend libraries
FCillegalContactStringException	thrown in case of illegal contact string is used to load or connect to the catalog
FCTransactionException	thrown when attempting to do an transactional operation outside the transaction state or attempting to update a read-only catalog. The transactional operations are file registrations, catalog lookups and catalog updates
FCconnectionException	thrown in case of failure of connection or disconnection
FCinconsistentSchemaException	thrown in case the schemas of the two catalogs are different in the cross catalog operation
FCstringLimitException	thrown in case the string length exceeds the limit of the backend, i.e. 250 char.

Operations

4 Analysis of the component

4.1 Glossary

Catalog Contact String: The string to identify a file catalog.

The format compliant with **[4]:**
[prefix_][protocol]://[username]:[password]@[host]:[port]/[path]

or

[prefix_]file:path

The latter format is accepted by the XML catalog.

File Identifier (FileID): Globally unique identifier associated with a file. It may be generated by the UUID mechanism in the format of binary or a long string

Physical File Name (PFN): The name refers to the physical location of a file.

Alias : Different logical names assigned to the same file or a set of files sharing the same

FileID.

Logical File Name (LFN): The same as alias.

Catalog fragment: A set of entries in the file catalog grouped by the user using certain criteria. For instance, all the files produced by a job, all the files contain data from a single run, etc.

Extract a catalog fragment: Select a subset of entries in a central catalog and put them in a local catalog. This operation should be atomic, i.e. selected entries are extracted in one transaction.

Publish a catalog fragment: Make a fragment of the file catalog available on the larger central catalog. The fragment of the catalog should be published atomically, i.e. one transaction publishes the entire fragment.

Register a file: Assign a FileID to a file and insert the PFN-FileID mapping in the catalog.

Replica: An exact copy of a file that is linked to the original file through some well-defined mechanism.

XML catalog: Catalog based on XML. It is human-readable and editable. This catalog or its fragment will eventually be published to larger catalogs.

Native MySQL catalog: Catalog based on MySQL database. It contains larger datasets

than the XML catalog, all the files produced by a farm, for instance.

It is not Grid-aware.

RLS based catalog: Grid-aware file catalog built on the EDG-RLS component.

Replica Location Service (RLS): A Grid service, which maintains and provides

access to information about physical location of copies of files.

4.2 Requirements

1. The file catalog contains the PFN-FileID and LFN-FileID mapping.

It supports FileID->PFN, PFN->FileID look-ups.

The FileID is globally unique and immutable.

2. A file can be registered without specifying anything for its LFN. LFNs can be assigned to files and registered in the catalog after the job is finished.

3. It should be possible to register files before the job runs, from the job or after the job is finished.

4. It should be possible to register and populate a file in one job and append to the same file later from a second job.

5. The catalog should be available both on and off the Grid. A job can be run locally independent of any catalogs on the Grid.

6. If the job crashes, the Grid catalog should be able to clean up the registration of the files that will be produced by this job and related jobs in the catalog. However, it's not the responsibility of the catalog to find and delete the partially produced file by a crashed job.

7. The catalog should support catalog browsing.

4.2.1 Implemented requirements

4.2.2 Future improvements

Lookup file can accept user-defined function object;

More strict and consistent transaction state control by all subcomponents;

More consistent error handling and error messages by all subcomponents

4.3 Use cases reports

The first two use cases are typical interactions of the storage manager with the file catalog. They happen in a single process.

The other use cases are typical operation mode of a group of jobs.

ID	*1*	*Register a file*
Description		*This use case happens when the storage manager creates a new file. The PFN may or may not already be registered by user in the catalog before the job runs. The file catalog component generates the FileID, adds it to the catalog and returns FileID to the storage manager. In the case that PFN is pre-registered in the catalog, the FileID found in the catalog is returned.*

Flow of events	Basic flow	*T=t0: the storage manager passes the PFN to be registered in the catalog.*
		T=t1: the file catalog component checks if the PFN is pre-registered
		T=t2: if the PFN is not pre-registered, the file catalog component generates a unique FileID.
		T=t3: the file catalog component inserts into the PFN-FileID pair in the file catalog.
		T=t4: the file catalog returns the FileID to the storage manager.
	Alternative flow	*T=t2: if the PFN is pre-registered, the file catalog component returns the FileID to the storage manager.*
	Alternative flow	*List of actions*
Special requirements	In the case of registering files before job runs: the pre-registration is handled by the command-line tools.	
Pre conditions	It is the responsibility of the user to guarantee the uniqueness of the PFN.	
Post conditions		
Extension points		

ID	2	*Lookup a file*
Description	*This use case happens when the storage manager dereferences a smart pointer to open a file for reading or update.*	
Flow of events	Basic flow	*T=t0: The storage manager specifies a unique FileID.*
		T=t1: the file catalog component looks up the PFN corresponding to the FileID in the catalog,
		T=t2: the file catalog component returns the PFN to the storage manager. In the current release, the first PFN found is returned.
		If PFN not found or the file cannot be accessed as indicated then throw exception.
	Alternative flow	*List of actions*
	Alternative flow	*List of actions*
Special requirements	The file may be opened for update. In this case, we assume the file to be opened is the master copy and is still available for writing	
Pre conditions		
Post conditions		

Extension points	If the file is not local, it's the responsibility of the Grid to decide how to ship or replicate the file. We pass to the Grid software the FileAccessPattern indicated by the user as a hint.
	We pass to the Grid software also the FileOpenMode(read,write or update) indicated by the storage manager. The Grid security should decide if the user has the permission to do indicated operation on the file.

ID	3		*Run a write-only production job*
Description			*In this scenario a large amount of new data are produced by the job and to be published to the collaboration. No input file is required by this job.*
Flow of events		**Basic flow**	*T=t0: before the production starts, the production manager registers all the output file PFNs and corresponding jobIDs into the catalog.*
			The job status is set to 0.
			This step is optional, PFNs can be registered also during the job.
			T=t1: the production starts and registered files are populated. When each job finishes successfully, the job status is set to 1.
			T=t2: the production ends. The production manager clean up the catalog entries where job status is 0 which means the job didn't end successfully.
			T=t3: The selected catalog fragment is published to the central catalog.
		Alternative flow	*Depends on the setup of the production, the catalog can be Grid-based catalog if the production runs in a Grid-aware environment; a MySQL catalog if the production runs in an isolated computer farm; or a XML catalog if the production runs in an isolated environment.*
			XML catalog fragment can be published to MySQL and Grid-based catalog; MySQL catalog fragment can be published to the Grid-based catalog.
		Alternative flow	*List of actions*
Special requirements			We assume that the Grid knows how to invalidate catalog entries produced by other jobs related to a crashed job.
Pre conditions			
Post conditions			
Extension points			

ID	4	Run a read-write job in an isolated environment
Description		*The job runs in an isolated environment. The job needs input files and it creates output files as well. The user decides whether or not to publish the output data later.*
Flow of events	**Basic flow**	*T=t0: User extracts the catalog fragment needed by the job from a central catalog into a local XML catalog when the network is available;*
		T=t1: user runs the job disconnected from the network. During the job run, the local input XML catalog is used for reading; the output files are registered into another local XML catalog.
		T=t2: n jobs run. n output XML catalogs are produced.
		T=t3: User selects and publishes the local catalog fragments to the central catalog.
	Alternative flow	
	Alternative flow	*List of actions*
Special requirements		
Pre conditions		*Resources needed by the job for reading are available.*
Post conditions		
Extension points		

ID	5	Browsing of the catalog
Description		*User needs to have available in memory a part of the catalog*
Flow of events	**Basic flow**	*T=t0: User decides which catalog start browsing*
		T=t1: The browsing starts returning all LFN-FileID pairs and/or all PFN-FileID pairs.
		T=t2: Navigation from LFN to PFNs via common FileID can be performed on shell or scripting. A new file catalog may be created by command lines with part of the extracted information.
	Alternative flow	
	Alternative flow	*List of actions*
Special requirements		

10/09/03

Pre conditions	
Post conditions	
Extension points	

4.4 Sequence diagrams

4.4.1 Sequence diagram for use case:

4.4.2 Sequence diagram for use case:

5   Technology surveys

This section describes briefly all the products, tools or components that have been evaluated because of any interest for the component

Name	MySQL	Vendor	MYSQL AB
Version	4.0.4-beta	License	GPL
Description	Database backend of native MySQl catalog and RLS based catalog		
Component related comments	InnoDB table type is used because it supports transaction.		
Conclusions			
What can be useful	InnoDB database engine, MySQL++1.7.9		

Name	Xerces-C++	Vendor	Apache.org
Version	2.1.0	License	
Description	Validating the XML-parser for C++.		
Component related comments			
Conclusions			
What can be useful			

6 Component design

6.1 Main Strategy

Complete separation between interface and implementations.

Multiple language support: compiled C++ and scripting Python.

6.2 Schemas

6.2.1 MySQL

Table: t_lfn(

        lfname varbinary(250) primary key,

        guid varbinary(40),

        INDEX idxt_lfn(guid)

) type=innodb

Table: t_pfn(

        pfname varbinary(250) primary key,

        guid varbinary(40) ,

        filetype varbinary(250),

        jobstatus int,

        filestatus int,

        INDEX idxt_pfn(guid)

) type=innodb

Optionally, file metadata can be associated to the fileID. Schema describes the metadata is defined by user.

A SQL script for creating the MySQL catalog schema can be found in
pool/Scripts/mysqlcatalog_schema_POOL1.sql

6.2.2 XML DTD

<!ELEMENT POOLFILECATALOG (META*,File*)>\

<!ELEMENT META EMPTY>\

<!ELEMENT File (physical,logical,metadata*)>\

<!ATTLIST META name CDATA #REQUIRED>\

<!ATTLIST META type CDATA #REQUIRED>\

<!ELEMENT physical (pfn)+>\

```
<!ELEMENT logical (lfn)*>\
<!ELEMENT metadata EMPTY>\
<!ELEMENT lfn EMPTY>\
<!ELEMENT pfn EMPTY>\
<!ATTLIST File ID ID  #REQUIRED>\
<!ATTLIST pfn name ID #REQUIRED>\
<!ATTLIST pfn job_status CDATA #IMPLIED>\
<!ATTLIST pfn filetype CDATA #IMPLIED>\
<!ATTLIST pfn file_status CDATA #IMPLIED>\
<!ATTLIST lfn name ID #REQUIRED>\
<!ATTLIST metadata att_name  CDATA #REQUIRED>\
<!ATTLIST metadata att_value CDATA #REQUIRED>\
";
```
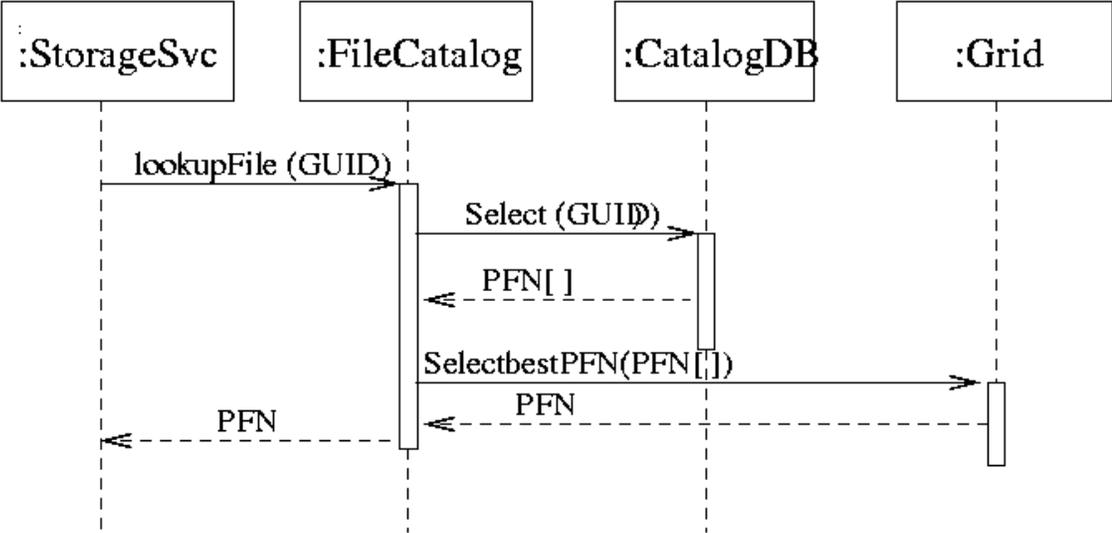
6.3   UML diagrams

1. Sequence diagram for use case: register file.

 :CatalogDB is the backend database of the catalog.
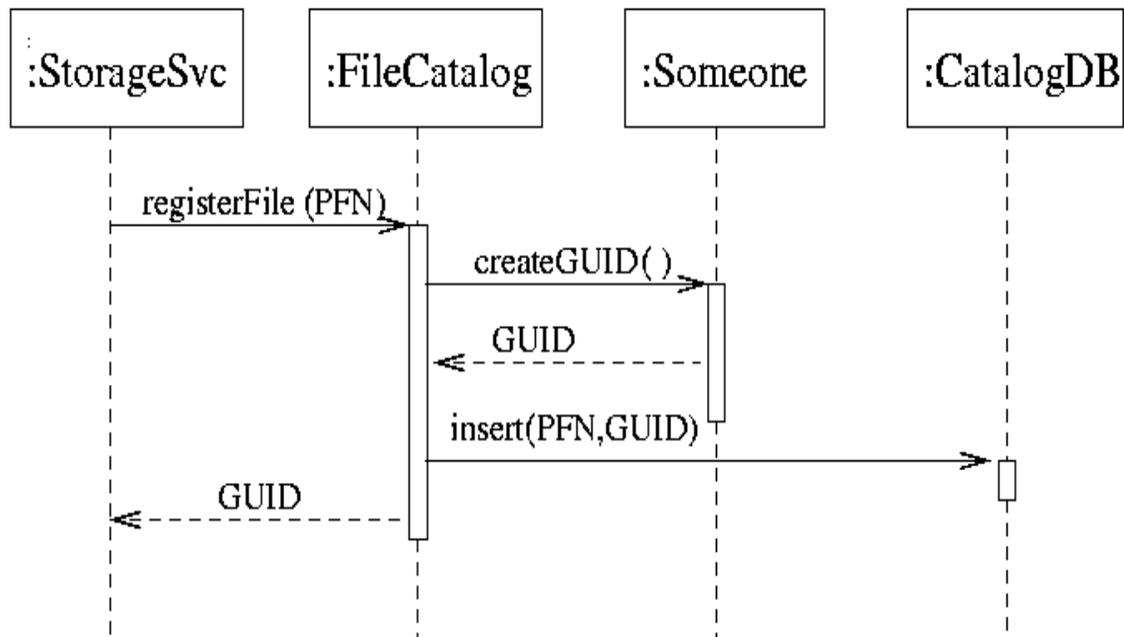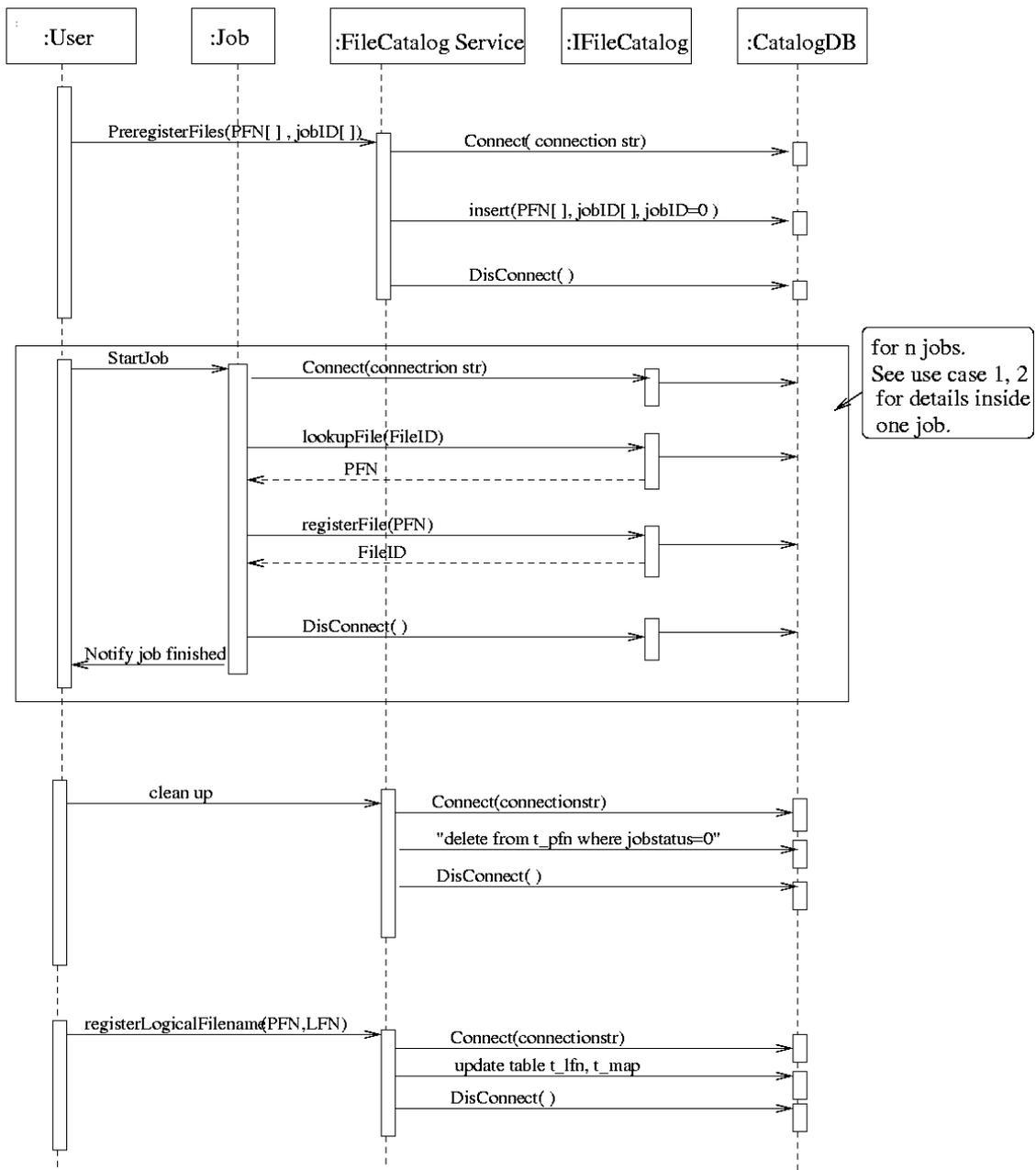

2. Sequence diagram for use case: lookup file.

For non Grid-aware catalog, the PFN[0] is returned.

3.sequence diagram for use case production.

:FileCatalog Service is the catalog command line tool while :IfileCatalog is the catalog interface to the storage manager.

.

6.4   Patterns used

List the patterns that have been used in the design.

6.5   Exceptions used

Explain the exceptions structure inside the component, identifying internal and external ones. Cut and paste the UML diagram for exceptions.

6.6   Additional comments, restriction and known problems

6.6.1   Example runtime setup script for SCRAM users.

The runtime environment variable can be set from a XML-style file:

eval `scram runtime –csh pool.env` (from csh)

eval `scram runtime –sh pool.env` (from sh)

An example pool.env is attached below:

```
<Ignore>

<Runtime name=POOL_CATALOG
value="mysqlcatalog_mysql://@lxshare070d.cern.ch:3306/testFCdb">

the MySQL db contact string</Runtime>

</Ignore>

<Runtime name=POOL_CATALOG value="xmlcatalog_file:FileCatalog.xml">

the XML db contact string </Runtime>

<Ignore>

<Runtime name=POOL_CATALOG value=" edgcatalog_http://rlstest.cern.ch:7777/edg-
replica-location/services/edg-local-replica-catalog">

the EDG RLS contact string </Runtime>

</Ignore>

<Runtime name=POOL_OUTMSG_LEVEL value="7">output message threshold</Runtime>
```

<Ignore> tags starts and ends the comment lines.

7   Implementation

7.1   MySQL

MySQLFileCatalog class implements the abstract interface. It contains an opaque pointer to class MySQLImpl which hides the private data members and functions used by the MySQL catalog and implements calls to the MySQL++ library.

MySQLPFNContainer, MySQLLFNContainer, MySQLMetaDataContainer implement the iterators for the MySQLCatalog.

The component throws FCException for logical errors and rethrows MySQL++ exceptions as FCException .

7.2   XML

The classes XMLFileCatalog is the concrete implementations of the interface IFileCatalog, handling file catalogs written as XML files. The DTD is held in memory and realizes the many PFNs to unique FileID to many LFNs association as described below. The catalog is parsed by using the Xerces-C libraries that provides the DOM interface with the functionalities to navigate through the trees. The catalog contact string is specified via the environment variable POOL_CATALOG. It mainly expects local files, though to browse the catalog content the http protocol is supported within the XercesC library. Other protocols might be implemented in a consistent way. In case the specified file catalog does not exist a new one is created from scratch.

Logging, error and exception handling are also ensured. Error and exception handlers are used by the component by the mechanism provided by the XercesC for the internal consistency of the XML file with respect to the DTD, and by the one defined in the IFileCatalog abstract component.

The class XMLFileCatalog, which implements the abstract interface, uses the class PoolXMLFileCatalog as the layer that provides the explicit functionalities to handle the XML specific catalog. PoolXMLFileCatalog encapsulates all the calls to the Xerces-C library. The consistency of the XML file catalog with respect to the DTD is ensured via the DOMError class that provides exceptions that are propagated up to the used code. The XercesDOMParser class provides the writing of the catalog as a DOM tree in the file specified by the contact string.

XMLPFNContainer, XMLLFNContainer, XMLMetaDataContainer implement the iterators for the XMLCatalog. The XMLQueryHandle class provides the functionality for the query handling.

The component throws FCException for logical errors and rethrows Xerces-C exceptions as FCException .

7.3 EDG

The EDGCatalog component implements the abstract user interface IFileCatalog by encapsulating methods and functions of the edg-rlc [2] and edg-rmc [3] c++ client libraries.

EDGPFNContainer, EDGLFNContainer, EDGMetaDataContainer implement the iterators for the EDGCatalog. The EDGQueryFiller and EDQQueryHandle provide the functionality for the query handling.

The component throws FCException for logical errors and rethrows edg-rls-client libraries exceptions as FCException .

8 To do

TO DO

9 References

[1] http://grid-data-management.web.cern.ch/grid-data-management/docs/ReplicaManager/White-Paper.pdf

[2] http://proj-grid-data-build.web.cern.ch/proj-grid-data-build/edg-rls-server/user-guide/edg-rls-userguide.pdf

[3] http://proj-grid-data-build.web.cern.ch/proj-grid-data-build/edg-metadata-catalog/user-guide/edg-rmc-userguide.pdf

[4] http://www.ietf.org/rfc/rfc2396.txt