# A Proposal for a Metadata Interface

*The ARDA Project*

*DRAFT (Jan 19th 2005)*

**Editors: B. Koblitz, N. Santos**

## *Introduction*

In the following we present a proposal for an interface design to access metadata catalogues on a grid-enabled infrastructure which is based on web-services. The interface was designed to be easy to use, portable over different SOAP toolkits, while allowing for performance through bulk operations. To prove the validity of the design, a prototype implementation was written and evaluated.

## *Design Goals*

The proposed metadata interface was designed as a general way to access databases and catalogues in a grid environment, whether these databases already existed outside of the grid scope or whether their schemas were conceived taking grid related issues like fine grained permissions into account. Implementations of the interface can either act as services with SQL or XML databases serving as back ends, or file-based caches or database proxies. The interface was inspired by the POSIX interface to extended file attributes but adapted to work over a network connection. The goal is to allow a fast but still simple access for the user clients or other grid services without requiring complex implementations.

The requirements for the design were to allow the user not only to read and write metadata from and into the storage back end, but also to discover the underlying schema of the metadata and even change it, in case the back end and the access policy allow this. The user can thus be provided with a means to set up his own metadata storage on the grid, including the possibility for schema evolution, without the need for an administrator to do this work manually on the storage back end. This is in contrast to the current gLite design, which only foresees fixed database schemata and expects applications to know this schema a priori [1]. We consider the proposed design closer to the application requirements. However, the discussions with the gLite team is ongoing.

The requirements for the interface were also taken and extrapolated from existing implementations of metadata services implemented by the HEP experiments at the LHC, which were studied extensively by the ARDA group [2, 3, 4]. Several design issues related to the actual implementation were also taken from these studies. The capabilities of the interface are also a superset of the requirements of the metadata working group [5].

Wherever prudent, bulk operations have been made possible in the interface design. Bulk operations are important for performance, since they can reduce remote calls over the network which are very time consuming. On the other hand, bulk operations are difficult to implement both on the server as well as on the client. On the server bulk operations need to be implemented as a single transaction: either it is fully executed or not. On the client preparing a bulk request can be complicated since the user needs to set up more complex data structures. Finally, error handling by the user can be very demanding. Therefore, for every interface call all these issues where weighed against each other to balance performance against ease of implementation and usability. It was for example found that a good alternative to lists of entries was the use of pattern matching which is much simpler to use as well as for the implementation to provide. Patterns were

therefore used wherever possible.

The current design should be understood as a starting point, where extensions are possible and feedback is necessary to improve its usability.

## *Terminology*

**Metadata**: Metadata are key value pairs which can be assigned to entries in a catalogue. Metadata may be stored in different back ends from SQL databases over XML databases to files.

**Entry**: Entries are the entities to which metadata can be attached. They can be added or removed from a database. Entries take the form of collection names separated by slashes followed by an entry name, similar to a path in a file-system. This allows hierarchies of entries. It is also possible to use the wild-cards “*” and “?” in entries, however not in the collection part.

**Collection**: A collection is a set of entries. Collections are also entries, that is, a collection can again contain a collection. Collections are a means to structure metadata hierarchically and restrict searches to subsets of the data. An implementation may choose not to have collections at all and thus work with per entry schemas (e.g. an extended attribute enabled file system).

**Schema**: A schema is the set of metadata attributes and their storage types of an entry. Implementations may limit schemas to collections, that means that all entries in a collection share the same schema.

**Attribute**: The name or key of a part of the metadata. Attributes must be alpha-numerical ASCII strings starting with a letter. They are case-sensitive.

**Value**: An application can assigned a values to the attributes of an entry which are transferred to the back end as a transfer value and stored there according to their storage type. Applications can discover these storage types and need to interpret a transferred value accordingly.

**Transfer value**: Values are transferred as strings of printable ASCII characters. This is due to the restrictions of the transport protocol and to allow interoperability on the network. However, the underlying back end may choose to have a different storage type. To send arbitrary binary data over the network, they can for example be UU-Encoded. In this case the applications themselves must take care of interoperability.

**Storage type:** Storage types are the data types used by the storage back ends to store the metadata values. They are implementation dependant, however all implementations must support the value strings of variable length used for the transmission over the network.

## *Interface Definition*

The following is the list of web-service interface methods provided through the SOAP protocol, a WSDL file can be found at [6]. First the methods to manipulate entries:

**`int addEntry(string entry, string type)`**

> This method adds an **`entry`** into the catalogue, where **`type`** is the the type of the entry: either “Collection” or “Entry”. An implementation could provide additional types, for example collections which inherit schemas from parent collections, views or indices.
>
> Return values: **`MD_SUCCESS`**, **`MD_ERR_NOENT`** (parent collection does not exist), **`MD_ERR_PERM`**, **`MD_ERR_INT`**, **`MD_ERR_TYPE`**, **`MD_ERR_EXIST`**

**int addEntries(list<string> entries, list<string> types)**

This method is a bulk version of the above method. It adds a list of **entries** into the catalogue, where **types** is the the type of the entry: either "Collection" or "Entry".

**Return values: MD_SUCCESS, MD_ERR_NOENT** (parent collection does not exist)**, MD_ERR_ILCMD** (size mismatch or illegal type)**, MD_ERR_PERM, MD_ERR_INT, MD_ERR_EXIST**

**int removeEntries(string pattern)**

Removes all entries matching **pattern** (including a collection) from the catalogue. Collections can only be removed if they are empty.

**Return values: MD_SUCCESS, MD_ERR_NOENT, MD_ERR_PERM, MD_ERR_INT, MD_ERR_NEMPT**

**int listEntries(string pattern, Handler &handler)**

Lists all entries matching **pattern** (including collections) and their types. The function fills a handler which can be used to iteratively retrieve the result and which is passed as a reference. The handler is a structure of type **Handler,** which gives access to the following fields:

```
struct Handler {
      handle_t handle;
      DataChunk chunk;
}
struct DataChunk {
      list<string> values;
      bool last;
}
```

The **values** list in the **DataChunk** member contains alternating the entry and its type of as many entries as the implementation chooses to provide in one chunk. While the last property is not set and no **error** is returned, the getNext() function described below can be called using **handle** to retrieve more results.

Return values: **MD_SUCCESS, MD_ERR_NOENT, MD_ERR_PERM, MD_ERR_INT**

**int getNext(handle_t handle, DataChunk &chunk)**

This method is used to retrieve results from after calling methods which return an iterator like **getAttr(), listAttr()** or **find()**.

Return values: **MD_SUCCESS, MD_ERR_ILCMD** (the session handle was bad or the last entry was already read.), **MD_ERR_ABORT**

**int abort(handle_t handle)**

This method is used to abort requests which return an iterator.

Return values: **MD_SUCCESS, MD_ERR_ILCMD, MD_ERR_ABORT**

Another way of listing entries fulfilling certain properties is the **find()** call described later on. The following are methods to manipulate the attributes of entries:

**int addAttr(string entry, list<string> keys, list<string> types)**

This method is used to add a list of attributes (**keys**) to an entry (including a

collection) which have the storage type **types.** The command is as a schema changing command expected not to be called frequently, also because it may result in a performance costly operation in the back end. It therefore lacks the capability to apply schema changes to several entries as bulk operations. This is further alleviated if an implementation chooses to have only per collection schemas, in this case a schema change acts on all entries within a collection.

Return values: **MD_SUCCESS, MD_ERR_NOENT, MD_ERR_PERM, MD_ERR_INT, MD_ERR_TYPE, MD_ERR_ILKEY, MD_ERR_ILCMD, MD_ERR_EXIST**

**int getAttr(string pattern, list<string> keys, Handler &handler)**

Reads the values of a list of attributes (**keys**) for a **pattern**, returning a handler.

Using the **handler**, the client repeatedly calls **getNext()** which fills up the handler structure with the name of the next **entry** and the associated **values** of the attributes until an error is returned or the **last** filed is true. The implementation can return the several entries at once with the only limitation that full entries and their attributes will be returned, an entry is never split.

Return values: **MD_SUCCESS, MD_ERR_NOENT, MD_ERR_PERM, MD_ERR_INT, MD_ERR_ILKEY**

**int listAttr(string entry, Handler &handler)**

Returns the names and types of all attributes via the **values** field of the **Handler** object. The string in values has the format of a declaration of the respective column in SQL. Entry may currently not be a pattern, so all the data is returned in one go. However, it may be possible in the future to extend the semantics.

Return values: **MD_SUCCESS, MD_ERR_NOENT, MD_ERR_PERM, MD_ERR_INT**

**int clearAttr(string pattern, string key)**

Clears the value of the attribute named **key** of all entries matching **pattern**. Getting the value of this attribute will henceforth return an empty string. Note that there is no way to distinguish a cleared attribute from an attribute which has not yet been set for an entry.

Return values: **MD_SUCCESS, MD_ERR_NOENT, MD_ERR_PERM, MD_ERR_INT, MD_ERR_NOKEY**

**int setAttr(string pattern, list<string> keys, list<string> values)**

Sets the list of **keys** to the list of **values** for all entries matching **pattern**. This bulk operation is very powerful and needs some setting up of data structures. On the other hand it can be easily implemented transaction safe in the back end and also error checking should be easy on the client side.

Note that in the case where pattern does not contain wild cards, the implementation may choose to insert the new entry into the catalogue if it did not formerly exist. This is a way of populating the catalogue halving the number of remote calls (**addEntry()** spared).

Return values: **MD_SUCCESS, MD_ERR_NOENT, MD_ERR_PERM, MD_ERR_INT, MD_ERR_NOKEY, MD_ERR_EXIST**

Finally a method exists to look up entries, including simply listing them:

**int find(string pattern, string query, Handler &handler)**

Retrieves a list of entries which match the **pattern**, fulfilling the query con-

dition **query**. The entries are stored in chunks in the value fields of the **handler.** The size of this list can be chosen freely by the implementation.

The query string has the following format in BNF:

```
<query> ::= '(' <expression> ')'
<expression> ::= <predicate>
     | <predicate> <logical>  <predicate>
<predicate> ::= <function> <comparator> <value>
     | 'not' <predicate>
<logical> ::= 'and' | 'or'
<function> ::= <key> | <func> '(' <key> ')'
<comparator> ::= '=' | '<=' | '>=' | '>' | '<' | '!='
<func> ::= 'abs' | 'sin' ...
```

which allows for example query strings like

```
(tracks > 10 and sin(angle) <0.5)
```

Return values: **MD_SUCCESS, MD_ERR_NOENT, MD_ERR_PERM, MD_ERR_INT, MD_ERR_ILKEY, MD_ERR_NOKEY, MD_ERR_QUERY**

The this is a list of errors that can can occur in the remote calls:

**MD_SUCCESS** (=0): No error, everything went fine.

**MD_ERR_NOENT:** No such file or directory.

**MD_ERR_ILCMD:** Illegal command, e.g. size mismatch of arguments.

**MD_ERR_PERM:** Permission denied.

**MD_ERR_INT:** Internal error, e.g. connection failed.

**MD_ERR_NEMPT:** Collection not empty (when removing it).

**MD_ERR_TYPE:** Illegal storage type

**MD_ERR_ILKEY:** Illegal key, the key is malformed

**MD_ERR_NOKEY:** No such key exists for the entry

**MD_ERR_ABORT:** Request aborted

**MD_ERR_EXIST**: The entry exists

**MD_ERR_QUERY:** The query is malformed (aka syntax error)

In general, the methods return 0 (**MD_SUCCESS**) on return, if no error occurred. In a programming language that supports exceptions, instead of returning error numbers, it could be possible to use the exception mechanism instead. Instead of passing references in the few cases where this is used, values would then be returned.

## *Related Interfaces*

As can be seen from the list of interface calls, the interface design is complementary to the interface of a file catalogue. A file catalogue is expected to have functionality to attach metadata to its entries using the above interface, which allows the only possible way to implement a high-performance selection of files based on their metadata as needed for example by user analysis jobs or the workload management system. Such an implementation will need to share the same table-space in the database back end to prevent cross-database join operations.

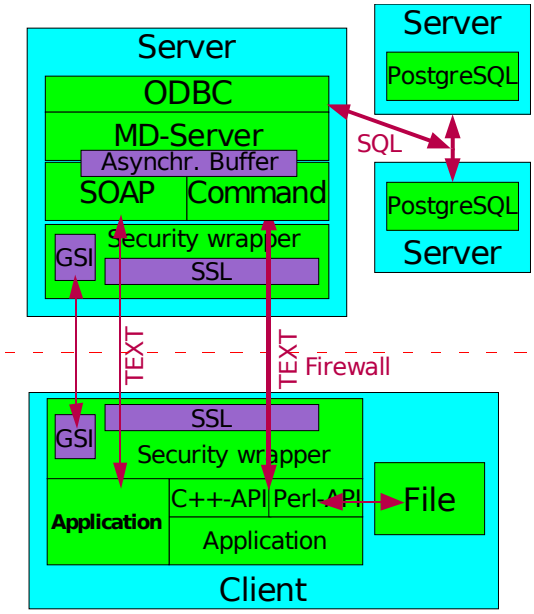


*Figure 1: Design of the metadata server prototype: Two interfaces for clients have been provided, a SOAP interface and a command interface with TCP streaming.*

Another issue is access control mechanisms. In case fine-grained access restrictions are necessary, they should be implemented using the interface of the file catalogue which already features such an interface, in order not to replicate interface designs. The above mentioned metadata catalogue implementation by the ARDA team has such an access restriction scheme based on per collection ACLs implemented. Studies with the implementation showed that it is possible to implement ACLs up to the collection level without noticeable performance penalties, similar to what AFS provides. However, if ACLs are needed on the entry level, severe performance problems are expected.

Finally, quotas may be a problem if users are allowed to add metadata freely into the catalogue. Currently no such interface is defined.

## *Reality Check*

In order to test whether the proposed interface design is a good solution for the user, can be fully implemented including transaction safety in the bulk operations and no performance bottlenecks exist, a prototype implementation was written. This prototype was intensively tested and also made available to external users.

The prototype design is shown in Figure 1. Applications can either directly talk to the metadata server through the SOAP interface, or via a streaming interface, which sends commands to the server as a single text-string which contains the call's parameters and the server answers with a stream of text as a response. Both interfaces can be encrypted using SSL and authentication can be done using grid proxy x509 certifictes. However, currently only the streaming implementation has these security features enabled. To investigate access controls, the implementation also provides ACL based access control on a collection level, where users can maintain their own access groups.

The actual work is being performed by the MD-Server class which queries an SQL database (in our prototype PostgreSQL) via an ODBC back end. We have extensively studied the streaming interface and have found that, using different cli-

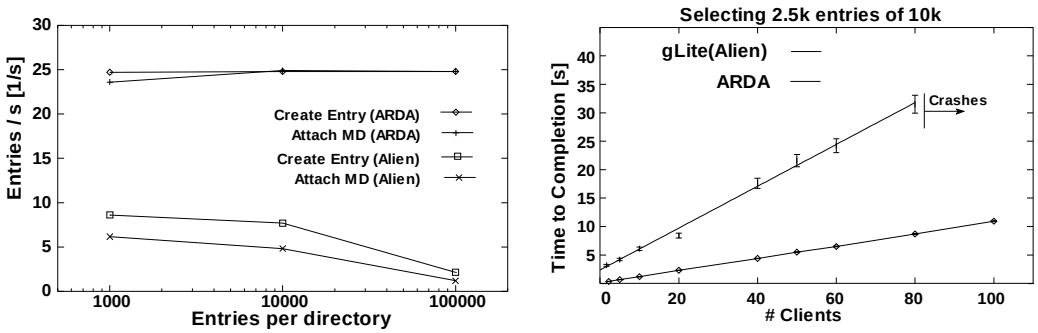


*Figure 2: To the left the insertion speed into the catalogue is shown for the ARDA metadata prototype and the Alien metadata catalogue when creating 1000, 10000 or 100000 entries in a collection (directory). Also shown is the speed with which 5 metadata attributes are added. To the right the speed with which entries are selected is shown for a growing number of clients which concurrently access the catalogue. The interface-implementation scales well up to 100 clients (the maximum number of connections allowed to the back end database), while the Alien catalogue suffers from various different server crashes resulting from time-outs. The tests were performed with the streaming front end, comparisons showed that the performance of the SOAP interface and streaming interface at least on a local network are compatible.*

ents, we were able to explore and test the following issues:

- Usability of the API: The API seems to be minimal, rather complete and fast (see figure 2 and [7]).
- The memory footprint is small (and independent of the query-size!): 128KB per connection for the server process plus a database process per connection (ODBC allows connection pooling which would make it possible to serve all server threads with only one database instance) for the streaming interface. The SOAP interface runs several threads managed by a master thread.
- The test implementation is stable and scales well (over a 100 concurrent connections were reached transferring a total of 1.2GB of data without a noticeable performance penalty on a desktop computer).
- ACLs for metadata can be implemented on a collection level without noticeable performance loss. In case metadata and file-catalogue live in one table-space on a relational database back end, restrictive metadata access permissions could be implemented without losing performance.

At this point we are confident that the proposed interface is a reasonable starting point for a generic metadata catalogue which can be used as a foundation to implement the file-metadata or other metadata catalogues for the HEP experiments.

## Acknowledgements

The ARDA team would like to thank the gLite team for their fruitful collaboration, especially R. Rocha, with whom this interface design was discussed extensively.

## References

[1] https://edms.cern.ch/file/487871/1.0/EGEE-DJRA1.2-487871-v1.0.pdf

[2] http://lcg.web.cern.ch/lcg/PEB/arda/public_docs/CaseStudies/ami_new.pdf

[3] http://lcg.web.cern.ch/lcg/PEB/arda/public_docs/CaseStudies/refdb_draft_v0.2.pdf

[4] LHCb metadata catalogue study in preparation.

[5] http://ppewww.ph.gla.ac.uk/~shanlon/Metadata/CoreUseCases_v5.pdf

[6] http://project-arda-dev.web.cern.ch/project-arda-dev/metadata/downloads/Metadata.wsdl

[7] http://lcg.web.cern.ch/lcg/PEB/arda/public_docs/CaseStudies/SOAPTest.pdf