

Requirements and Implementation Ideas of a Metadata Catalogue

v2.0

The ARDA Project

25/10/04

Editor: B. Koblitz

Abstract: We present a design proposal for a generic metadata catalogue which has an interface inspired by the POSIX standard. Experience with a small prototype implementation has been performed to validate this design. Streaming of results allows the server to operate with a small memory footprint and eases problems with timeouts.

Definitions

In the following we will define metadata as attributes of a file and take the view point is file-related metadata. For general metadata the reader should simply read collection for directory, and entry for file, since the semantics of general metadata are a subset of the semantics of file-metadata which knows of hierarchies in addition:

Metadata is key-value pairs associated to a file where the key is a 0-terminated string and the value can be any binary data of a given size. This corresponds to the POSIX definition of file metadata, also termed extended file attributes, and allows to transparently store files with attached metadata into metadata aware file systems which are currently coming widely into use. We will use the words key and attributed interchangeably.

Metadata can be attached to directories, in this case files inherit the metadata as defaults. Here, directories are any collection of files defined in the file-catalogue. A directory does not need to have the semantics of a directory in a UNIX file system.

Copying a file copies also the metadata associated to the file.

In the case of the Grid, metadata will be associated with an LFN.

Rationale for this Design:

Using the POSIX definition of metadata for grid-metadata allows to copy files and their metadata to local file systems preserving the semantics of this metadata. Using this design allows also to use the well though through POSIX interface to this metadata.

Attaching metadata to the LFN distinguishes the metadata catalogue from the file catalogue where metadata-like access controls for files are associated with GUIDs. Associating the metadata with LFNs allows the user to make very flexible user of metadata: Every user can have his own metadata associated with a file or have different metadata associated with a file depending on the context (directory it is stored in), e.g. one directory may contain MC-files with their creation date and software versions as metadata in the context of the MC production system while a physicist analysing MC-files may have only parameters of the employed generators as metadata.

Attaching metadata to the LFN allows to group metadata by directory and thus provides a hierarchical structure allowing to distribute metadata over sites. Using

directories to group files with similar metadata also allows to efficiently search files matching certain criteria in individual directories.

However, if a file catalogue is chosen without providing a directory structure (or in the case of files without LFN where the GUID can be used as index), it would still need to provide the functionality to group files in order to allow efficient searching.

A Possible Implementation:

Every directory containing files with metadata is represented by an SQL table with all keys as columns and all files in the directory as rows with the respective values. This allows fast searches for files in a directory. It also allows users to have different metadata schemas. Adding or removing metadata from a file may alter the columns of the table. Setting metadata for a directory changes the defaults for the column values of the table (a variant may be a row with defaults for the directory).

Experience with existing metadata prototype implementations [AMI, REFDB] suggests to keep the protocol overhead due to a server in front of the database backed as small as possible by offering a minimum functionality and not encapsulating the responses of the database directly (e.g. by SOAP). To keep the memory footprint of the server small even for very large responses, it is necessary to stream the response to the client. This is a key feature which most of the systems analysed by ARDA miss so far.

A possible implementation could be to use clients which send text commands to talk to a multi-threaded server in front of a database which would stream back the response. On top of this very simple interface which allows very quick implementation of clients in different programming languages, authentication, authorization and encryption could be added by wrapping the connection with SSL (such a solution has been developed by ARDA for the encapsulation of gLite commands). The server should be implemented as a back end talking to the RDBMS and a front end receiving the commands and dispatching the responses. A web-services interface is currently designed together with the gLite development team. It is using iterators for handling large amounts of data. Also SOAP implementations for streaming back data are currently evolving and could be used in a later version of the web-services interface.

Proposal for a command interface:

The design of the command interface of the server is inspired by the POSIX API. On the other hand it needs to take care of performance issues related to the fact that all commands need to be sent over a TCP connection by the client. Therefore the interface allows for bulk transfer of data to reduce the number of remote calls.

The following interface is designed in such a way as that it is complementary to the interface of a file-catalogue. This means that the management of entries (files) and collections (directories) is handled by the file-catalogue interface. A stand-alone metadata server, which is not integrated into the file catalogue, would need to implement some parts of the file catalogue interface, too. The full interface design of a stand-alone metadata server can be found at [PROTOCOL].

The following commands can be sent as text to the database front end. The response will be a line with a number as OK/Error code and then a list of strings, each string on a single line, this list of strings is terminated by the EOT character:

```
addattr directory key [type]
```

Adds a new key to the list of keys of a directory. Type is the name of an SQL datatype which will be translated if necessary into a data type understood by the back end.

The type is only used as a hint for the back end to store the data efficiently and allow efficient queries. The type may be ignored by the implementation (e.g. if the back end is a filesystem). If type is omitted a default type is used. In a filesystem the types and defined keys could be stored as attributes of directories. The list of types supported by the metadata server needs to be defined.

`removeattr dir key`: Removes an attribute corresponding to key.

`setattr file [key value]...`

Sets a list of keys to given values for a file. The keys must exist.

`getattr path key1 key2 ...`

Returns for all files matching the path first the file name is returned then the values of the requested keys as a text string. All strings are returned on a single line.

`listattr file`: Returns all keys of a file as text-strings and their type.

`clearattr path key`: Unsets an attribute corresponding to key for all files matching path.

`find pattern query`: Returns a list of file names matching pattern and fulfilling the SQL query on their attributes. The list is terminated by an empty string. The SQL query needs to be parsed by the implementation to prevent exploits and to translate the query to the query language of the back end.

The commands can contain quoted strings e.g. the SQL query could be 'tracks > 10'. The responses must be plain ASCII. In any case the end of line and EOT characters need to be escaped. If it is necessary to store binary data, this can be done by storing UU-Encoded strings but handling this is the task of the client application.

To handle collections (directories) and entries(files) at least calls to list files, delete them, move them, copy them and do the same with directories need to be provided but may be implemented by the file-catalogue part of the service.

The key feature of this protocol is the possibility to stream it, allowing for a small memory footprint on the server. It can be in addition easily authenticated/encrypted using GSI.

Proposal for a C++ API:

In the following a C++ client-side interface is described as an example of a programming API. Not all calls are documented, instead only some calls are described to illustrate how the data stream can be handled by the user's application through iterators. A C++ example of the API was chosen because C++ allows easy handling of bulk data operations with its container classes. The API should look similar in other programming languages. The interface is strongly influenced by the POSIX API for metadata which is implemented in C. More information on this API can be found in the man-pages located at <http://acl.bestbits.at/man/man.shtml>.

A good example of a bulk transfer handled using iterators on the client side is

```
int getAttr(const string &pattern, const list<string > &keys,
           AttributeList &attributes)
```

which returns the values of all keys for all files matching pattern, via the `AttributeList` class which allows iterative access via

```
int AttributeList::getRow(string &file, vector <string > &attributes)
```

```
bool AttributeList::lastRow()
```

to the data which is streamed back by the server after the execution of the `getattr` command. The `getRow()` call returns the name of the current file and its attributes via the function's arguments. With `lastRow()` it is possible to check whether there are still unread entries. Note that `AttributeList` actually keeps a buffer of the data already streamed to the client whose content is made available entry by entry through the `getRow()` call. This interface is able to retain the speed of a streamed connection as well as keeping the memory usage on the server as well as the client side small. The full documentation of the interface is located at [ARDAMD].

Note that the design of this interface is similar to the design of the ODBC or JDBC interfaces to access databases and its performance is thus well-proven.

A Prototype Implementation

In order to study the usability of the interface design and the APIs, as well as their performance impact, a prototype implementation was developed. It is written in C++ and acts as a multi-threaded server in front of a PostgreSQL database back end using the ODBC abstraction layer. A C++ client library talks to the Server using the described text protocol via a TCP/IP connection. In addition, a Perl API was created as well as a Python API. As expected for such a simple text-protocol, writing the client APIs in the different programming languages turned out to be very simple.

Using different clients, we were able to explore and test the following issues:

- Usability of the API: The API seems to be minimal, rather complete (if many updates are expected a bulk-update command could be added) and performant.
- The memory footprint is small (and independent of the query-size!): 128KB per connection for the server thread plus a database instance per connection (ODBC allows connection pooling which would make it possible to serve all server threads with only one database instance).
- The test implementation is stable and scales well (60 concurrent connections were reached transferring a total of 1.2GB of data without a noticeable performance penalty on a desktop computer).
- Having metadata- and file-catalogues implemented in one database allows an effective system for access control based on ACLs for metadata without noticeable performance loss.

Closing remarks

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

Whether there is a size limit on the metadata, the size of this may be an implementation detail like in file systems, database systems typically have size limitations for their data types. Implementation of the DB schema itself is also an implementation detail. It would be possible to have schema evolution or support for fast addition or removal of columns.

Links:

[AMI] http://lcg.web.cern.ch/LCG/peb/arda/public_docs/CaseStudies/ami_new.pdf

[REFDB]

http://lcg.web.cern.ch/LCG/peb/arda/public_docs/CaseStudies/refdb_draft_v0.2.pdf

[ARDAMD] <http://project-arda-dev.web.cern.ch/project-arda-dev/metadata/>

[PROTOCOL] <http://project-arda-dev.web.cern.ch/project-arda-dev/metadata/protocol.html>