An Interactive Shell and a Python Module for DMLite

G. Jahn

CERN openlab[2]

01.09.2012

Contents

1 Introduction **1**
1.1 DPM and DMLite . . . 1

2 PyDMLite **2**
2.1 Implementation . . . 2
2.2 Installation . . . 2
2.3 Usage . . . 3

3 DMLite Shell **4**
3.1 Implementation . . . 4
3.2 Available Commands . . . 5
3.3 Command Line Options . . . 5
3.4 Adding new Commands . . . 5

4 Prospect **7**

A PyDMLite Example **8**
A.1 Special Methods of the `stat` class . . . 8

B Sources **9**

C References **9**

Abstract

DMLite is a grid-aware storage solution used by the WLCG that is light-weight, simple and open to many standards. To make the API even more versatile, Python bindings for the DMLite library were created so that DMLite commands are also available for scripting purposes. The result is the *PyDMLite* module that is described in chapter 2.

Furthermore, a Bash-like shell for DMLite commands was developed that allows a quick access to many of the features of DMLite via the command line for easy testing and scripting. The focus of the shell was put on good usability and extendability so that new commands can be added easily.

1 Introduction

The *Large Hadron Collider* (LHC) at CERN is a synchrotron near Geneva which provides higher collision energies and luminosities than any other particle accelerator in the world. Nearly two decades before the LHC was placed into operation in 2010, its parameters and its four major experiments ALICE, ATLAS, CMS and LHCb were designed. Because of the unprecedented collision rates and detector dimensions, the experiments estimated the requirement of enormous amounts of computing resources: Alone the ATLAS Experiment estimated a required computing power of more than 10^{12} instructions per second and a raw data output of about 10^6 GB [6] per year which seemed nearly infeasible at that time. Until today, these requirements have even risen, the ATLAS Experiment produces about 10-20 PB [7] of data per year.
This data is distributed in a network of computer clusters, the *Worldwide LHC Computing Grid* [4] (WLCG), where it is stored and processed. The WLCG is a Grid infrastructure of more than 140 sites in 35 countries established particularly for the LHC to make the data and computing power available to scientists around the world. The entities of the WLCG are hierarchically structured in Tiers, there is one Tier-0 center directly at CERN which distributes the data to 11 Tier-1 centers in different countries. The lowest level is formed by more than hundred Tier-2 sites which are usually smaller university clusters.

1.1 DPM and DMLite

The grid storage solutions used in the WLCG must provide high-performant read and write access to the large amounts of data that are stored distributedly. Different interfaces for the transfer of data between sites and services must be provided. A light-weight storage solution mainly used at Tier-2 sites is *Disk Pool Manager* (DPM) [3]. The focus of DPM is to simplify the grid-aware management of large data storages.

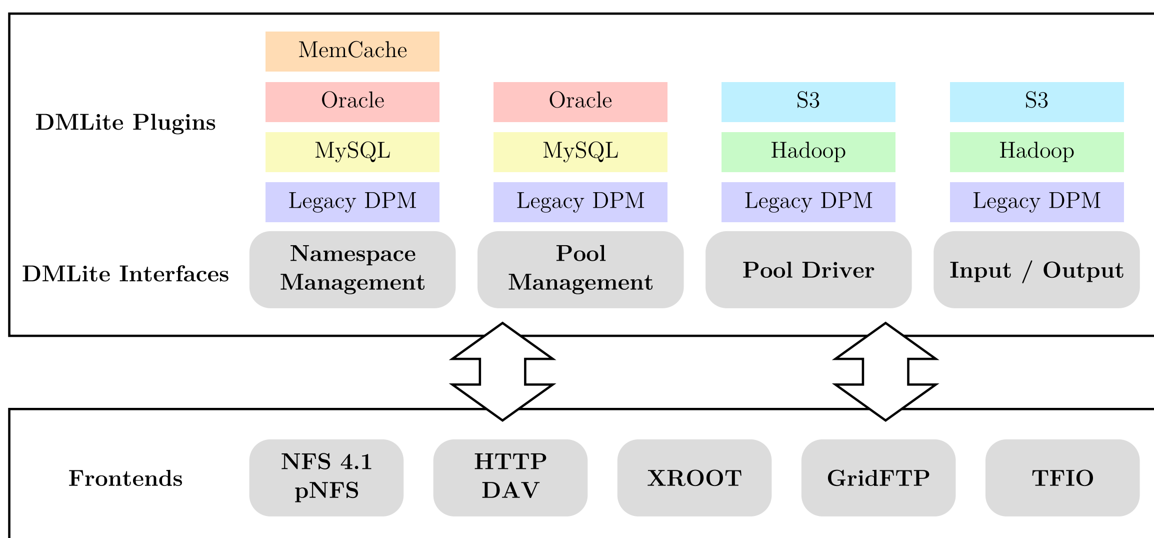

Figure 1: Schema of the DMLite structure: The frontends access the abstracted DMLite interfaces which forward the requests to enabled plugins.

As DPM is getting more and more widespread, a flexible integration into different systems and the usage of standard protocols becomes more important. This and performance issues lead to the development of *DMLite*[5], a plugin-based library that allows a standard protocol compliant access to DPM as well as other storage solutions as HDFS or S3. Different storage backends can be added to DMLite as plugins, many of these plugins can be stacked as shown in a sample in Figure 1: Here, the legacy DPM Plugin is always used as a last fall-back plugin for functionality that is not implemented by the other plugins. The performance of the pure DPM system is enhanced with caching and database plugins and new pool types are added via the S3 and Hadoop plugins.

2 PyDMLite

As all frontends will be accessing DMLite, it will gradually become the new core of the DPM system and thus the most important library. The library is a C++ library with the additional possibility to access it from plain C. To the make it even more versatile, the first goal of my project was to create Python bindings for this library so that all DMLite functionality is easily accessible via a Python module. The outcome of this effort is called *PyDMLite* and is available in the DMLite-SVN in the folder `python`. It is compliant with the current DMLite API version 20120828 and will have to be updated once bigger changes in the DMLite API occur.

2.1 Implementation

PyDMLite is a Python module implemented with *Boost.Python*[1]. The basic structure of the DMLite library as well as the naming scheme is preserved and nearly all functionality of the contained classes is available in Python. The structure of the implementation adheres to the structure of the DMLite header files: For each header file, e. g. `base.h`, there is a corresponding file, e. g. `base.cpp` which exports the contained classes, structures, and constants to Python. If some classes are purely virtual and thus wrapper classes are required or further definitions are used to make data types accessible to Python, an additional `basewrapper.cpp` file is used.
This structure makes the process of keeping the Python bindings consistent with the DMLite library relatively easy since it is clear which files need to be changed. The compilation of these files will be done separately and they will be linked together by a CMake file. This proved to be faster than the previous approach and gives also the advantage that only the changed files have to be recompiled.

2.2 Installation

If the source is downloaded from the SVN, the PyDMLite module can be installed via `make install`. This will copy the file `pydmlite.so` to the Python module folder, so that it is available in every python script via

```
1 import pydmlite
```

The python module will search for the DMLite library `libdmlite.so.0` in the default path `/usr/lib64/`, if it is not installed there, the installation location has to be specified with the `export` command. The correct installation of the module can be checked in the python interpreter:

```
% python
Python 2.4.3 (#1, Jun 19 2012, 13:51:22)
>>> import pydmlite
>>> pydmlite.API_VERSION
20120828
>>> dir(pydmlite)
['API_VERSION', 'Acl', 'AclEntry', 'Authn', 'AuthnFactory',
   'BaseFactory', 'BaseInterface', 'Catalog', 'CatalogFactory', ...
```

2.3 Usage

As stated before, the general structure of the original DMLite library is preserved and the naming scheme of the library is largely untouched. There are only some small exceptions: For STL-vectors the Pythonic equivalent methods are used, e. g. `append` instead of `push_back`. For the `Any`- and `Extensible`-objects `set***`-functions have been introduced to support the important primitive types from C++. The following listing compares two pieces of code that illustrate the changes:

```
// C++ code
dmlite::SecurityContext root;
root.user["uid"] = 0u;
root.groups.push_back(group);

# Python code
root = pydmlite.SecurityContext()
root.user.setUnsigned("uid", 0)
root.groups.append(group)
```

The `Any`-type is also enhanced with an `extract` method which takes advantage of Pythons dynamic typing and returns value in the corresponding Python type. This can be used as an alternative to the `get***`-methods of `Extensible`-objects:

```
# two equivalent methods to get the user id of root
ext = authn.getUser("root")

# method a)
userid = ext.getUnsigned("uid")
# method b)
userid = ext.getElement("uid").extract()
```

Note that method b) would also work if the field `uid` would be of any other primitive data type whereas method a) always tries to interpret the field as an `unsigned int`. A more comprehensive example for the usage of the PyDMLite module can be found in the appendix.

3 DMLite Shell

For an easy access to the DMLite functionality, especially in terms of testing and debugging of the individual modules but also of the DMLite library itself, the Python bindings can be used. The Python interactive shell allows for a quick and interactive access to the internal objects and structure. Still, it can be a tedious task to create the same objects over and over again in the Python shell, this is why a Bash-like interactive shell for more abstract DMLite commands was introduced to have a more compact syntax at hand. The key features of the shell are

- intuitive Bash-like commands,
- quick navigation (via auto-completion),
- help system for easy syntax lookup,
- easy extensibility,
- command line options for usage in scripting.

3.1 Implementation

The DMLite Shell was entirely written in Python; it accesses the DMLite library via the PyDMLite module. It does not support all DMLite functionality yet, but new commands can be added easily. The DMLite Shell can be found in the folder `python/shell` in the SVN.

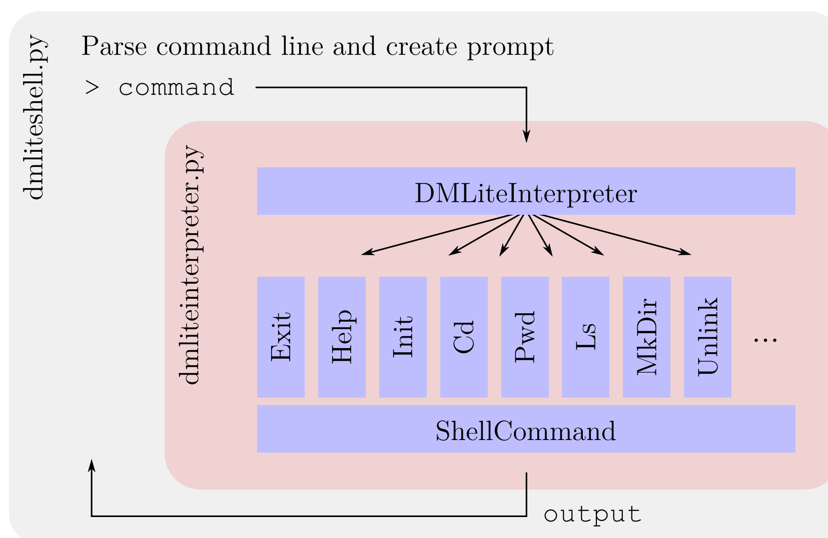

Figure 2: Schema of the DMLite Shell, blue rectangles are classes: The commands are passed to the class `DMLiteInterpreter` which selects the correct command class that is derived from the `ShellCommand` class which provides basic functionality.

The executable file is `dmliteshell.py` which parses the command line options and creates a prompt if necessary; commands can also be passed directly in the command line options or via a file. These commands are then passed to `dmliteinterpreter.py` which interprets the

input and is also able to do the auto-completion. It has the whole catalog of all commands which are implemented as classes derived from `ShellCommand`. This allows for a very dense and convenient way to define commands.

3.2 Available Commands

The commands implemented so far give access to the following aspects of DMLite:

- Basic folder navigation (`ls`, `cd`, `pwd`, `mkdir`, `rmdir`, ...)
- Basic file manipulation (`create`, `unlink`, `mv`, `chmod`, ...)
- Extend File Attributes (`info`, `acl`, `comment`, `checksum`, ...)
- Replica manipulation (`addreplica`, `rmreplica`, `chreplica`, ...)
- User/Group management (`userinfo`, `groupinfo`, `chown`, ...)

A list of all commands is available via the `help` command.

3.3 Command Line Options

The DMLite Shell will open the default configuration `/etc/dmlite.conf` if no other configuration is specified via the `-c` option. It will always access the plugins using a root security context; currently, there is no option to change that.
A useful option for scripting is `-e`, which executes a single given command and exits. For example, the contents of a directory can be listed as in the following:

```
% ./dmliteshell.py -e "ls /dpm"
1              1.7GB
cern.ch        (dir)
nfs            (dir)
test-folder-a  (dir)
testa          0B
```

To execute several commands one after another, the `-s` option can be used to pass a file with commands seperated by newlines or the commands can be piped to the DMLite Shell.

3.4 Adding new Commands

As mentioned before, adding new commands to the DMLite Shell is particularly easy because the commands are organized as individual classes. These classes are in the Python file `dmliteinterpreter.py` and their name has to be `<name>Command` where `<name>` is automatically used as the command name. The first line of the class documentation will be shown as the short description of the command, whole documentation is shown when the user enters `help <name>`. By default, the command takes no parameters, if it should take parameters, they have to be specified in the `parameters` array in the `_init` method. To define the behaviour of the command, the `_execute` method has to be overwritten. To give a quick example, the following shows the code of the `mkdir` command:

```
class MkDirCommand(ShellCommand):
    """Creates a new directory."""
    def _init(self):
        self.parameters = ['ddirectory']

    def _execute(self, given):
        try:
            self.interpreter.catalog.makeDir(given[0], 0777)
        except Exception, e:
            return self.error(e.__str__() + given[0])
        return self.ok('Created directory successfully.')
```

The expected parameters are specified as a list of strings, each string specifies one parameter. These specifiers may start with a `*` to indicate that the parameter is optional. After an optional parameter only more optional parameters are allowed to keep the command syntax simple. The first character after that denotes the type of the parameter, Table 1 lists implemented parameter types.

Type	Description
`d`, `D`	file or folder in DMLite
`f`, `F`	local file or folder
`g`, `G`	DMLite group
`u`, `U`	DMLite user
`T`	date/time including “now”
`o`, `O`	option from a predefined list
`c`, `C`	DMLite Shell command name
`?`	undefined type (no auto-completion)

Table 1: Parameter types available for the implementation of new commands. For lowercase letters, auto-completion for the parameters is available, if an uppercase letter is used an additional existence check is done.

Note that using a predefined parameter type instead of the `?` type for undefined parameters increases the usability of the command because Bash-like auto-completion with relevant items is activated. If an uppercase letter is used as the parameter type, it will be additionally checked that the parameter given by the user is a valid and existing item; if this is not the case, an error message is shown and the `_execute` function will not be called. After the parameter type, the parameter name follows. It is suggested to use only alpha-numeric parameter names that may also include a dash “-”. The parameter name is shown for example by the `help` command and when syntax errors occur. If the parameter type `o` or `O` was used, after the parameter name there will be a colon-separated list of available options.

The example above thus specifies only one parameter (that is not optional) with the name `directory` and that has auto-completion from the list of all files and folders in DMLite. Two more examples of parameter specifiers are `'*?checksum'` for an optional parameter that has neither auto-completion nor any consistency checks and `'Otype:volatile:permanent'` that is a parameter where either `volatile` or `permanent` can be chosen.

For output, the method `self.ok` may be called; if an error occurs, `self.error` should be used to display the error message. This will also result in an failed command execution which may

for example stop the execution of further DMLite Shell commands. The `_execute` method should return either the result of `self.ok` or `self.error`. The parameters given by the user are available in the `given`-list as strings or timestamps if the parameter type `T` was used.

4 Prospect

The described functionality has passed some basic testing and is thus available in the main branch in the SVN. Both PyDMLite and the DMLite Shell are working fine and are ready to use. Nevertheless, many parts of the PyDMLite module are not well-tested yet, e. g. the I/O-interface has not been used by any application. It is also important to keep in mind that the PyDMLite library needs to be updated constantly if the DMLite API changes.

The development of both projects did not only lead to new tools to access the DMLite functionality, but also revealed several small problems and bugs in the DMLite library as well as missing required functionality. Moreover, albeit the DMLite Shell and the PyDMLite module do not include all potential functionality yet, they are versatile tools that were designed to be extendable easily.

If any problems or further questions about the architecture of the system should arise, I can be contacted at `gjahn@cern.ch`. My thanks go to CERN, especially to CERN openlab and to my supervisors Ricardo and Oliver for an intensive and memorable summer.

A PyDMLite Example

The following listing illustrates what the important steps in the usage of the DMLite library via PyDMLite are: First, a `PluginManager` has to be created and a configuration file has to be loaded. Then, a valid `SecurityContext` has to be created with a user name and a corresponding group; in the example a root-user is created. Then, a `StackInstance` can be created using the `PluginManager` and the `SecurityContext` can be selected. From this `StackInstance`, all of the DMLite interfaces are available, for example `getCatalog` returns a `Catalog`-Object, with which the file catalog can be accessed.

```
Python 2.4.3 (#1, Jun 19 2012, 13:51:22)
>>> import pydmlite
>>> pluginManager = pydmlite.PluginManager()
>>> pluginManager.loadConfiguration("/etc/dmlite.conf")
>>> root = pydmlite.SecurityContext() # the following lines build a
   valid Security Context
>>> group = pydmlite.GroupInfo()
>>> group.name = "root"
>>> group.setUnsigned("gid", 0)
>>> root.user.setUnsigned("uid", 0)
>>> root.groups.append(group)
>>> stackInstance = pydmlite.StackInstance(pluginManager)
>>> stackInstance.setSecurityContext(root)
>>> catalog = stackInstance.getCatalog() # get access to the file
   catalog
>>> catalog.changeDir("/dpm")
>>> catalog.getWorkingDir()
'/dpm'
>>> catalog.create("test-file-a", 0755) # create a test file
>>> catalog.rename("test-file-a", "test-file-b")
>>> stat = catalog.extendedStat("test-file-b", False)
>>> stat.name
'test-file-b'
>>> stat.parent
0
>>> stat.guid
''
>>> stat.csumvalue
''
>>> stat.status
pydmlite.FileStatus.kOnline
>>> catalog.unlink("test-file-b")
>>>
```

A.1 Special Methods of the `stat` class

When using the `stat`-class that is available as a subobject of any `ExtendedStat`-object, the time fields are not directly accesssible but only via the methods `getATime`, `getMTime`, and `getCTime` respectively. Additionally, the methods `isDir`, `isLnk` and `isReg` have been introduced to test for the file type equivalently to the corresponding C macros.

B Sources

Figure 1: Figure has been adapted from the original which can be found in [4].

C References

[1] *Boost.Python 1.41.* http://www.boost.org/doc/libs/1_41_0/libs/python/doc/index.html

[2] *CERN openlab.* http://openlab.web.cern.ch/

[3] *Disk Pool Manager DPM.* https://www.gridpp.ac.uk/wiki/Disk_Pool_Manager

[4] *WLCG – Grid computing.* http://lcg.web.cern.ch/lcg/public/grid.htm

[5] Alvarez, Alejandro ; Beche, Alexandre ; Furano, Fabrizio ; Hellmich, Martin ; Keeble, Oliver ; Rocha, Ricardo: DPM: Future Proof Storage. (2012), Jun, Nr. CERN-IT-Note-2012-021

[6] Armstrong, WW u. a.: ATLAS: technical proposanal for a general-purpose pp experiment at the large hadron collider at CERN. In: *CERN/LHCC* (1994), S. 171–173

[7] Brumfiel, Geoff: Down the petabyte highway. In: *Nature* 469 (2011), S. 282–283