Abstract. The Disk Pool Manager (DPM) is a lightweight solution for grid enabled disk storage management. Operated at more than 240 sites it has the widest distribution of all grid storage solutions in the WLCG infrastructure. It provides an easy way to manage and configure disk pools, and exposes multiple interfaces for data access (rfio, xroot, nfs, gridftp and http/dav) and control (srm). During the last year we have been working on providing stable, high performant data access to our storage system using standard protocols, while extending the storage management functionality and adapting both configuration and deployment procedures to reuse commonly used building blocks. In this contribution we cover in detail the extensive evaluation we have performed of our new HTTP/WebDAV and NFS 4.1 frontends, in terms of functionality and performance. We summarize the issues we faced and the solutions we developed to turn them into valid alternatives to the existing grid protocols - namely the additional work required to provide multi-stream transfers for high performance wide area access, support for third party copies, credential delegation or the required changes in the experiment and fabric management frameworks and tools. We describe new functionality that has been added to ease system administration, such as different filesystem weights and a faster disk drain, and new configuration and monitoring solutions based on the industry standards Puppet and Nagios. Finally, we explain some of the internal changes we had to do in the DPM architecture to better handle the additional load from the analysis use cases.

1. Introduction
The Disk Pool Manager (DPM) is the most widely used grid data storage solution. Its software has been evolving over the years, but its goals have remained the same from the start of the project:

- Provide a lightweight, grid-aware storage solution
- Simplify life of users
- Simplify life of administrators
- Keep improving the feature set and performance

In more recent times, and as interest in big data started spreading to other areas outside the scientific communities, a new generation of tools and protocols started offering viable solutions which could potentially reduce our dependencies in custom built software. An interesting option in terms of performance and maintainability, which needed further validation.

With this in mind we have done during the last year a thorough evaluation of the available options and of our own system, both in terms of functionality and performance, adding new major goals to the project (while keeping the ones above):
• Use standard protocols
• Use standard building blocks
• Allow easy integration with new tools or systems

In this paper we report our findings and the solutions we have developed and have started releasing to users. We start by a report on the evaluation mentioned above. We include a summary of the existing architecture and the bottlenecks identified which prevented an easy path to the goals listed above, in terms of performance and flexibility. We also summarize the choices we have made to offer standards based data access, for both the download/upload and remote i/o use cases.

We then describe in detail the main result of this evaluation, namely a significant code refactoring effort that resulted in a common and extensible library called dmlite. We describe the internal interfaces we have defined for namespace interaction, pool management and i/o. We introduce all the currently available plugins that have been written for it, and give hints on how they will benefit existing setups, including relevant comparisons between the legacy and the refactored nameserver and the different pool management backends. We put special emphasis on plugins reusing standard building blocks, like memcache or hadoop/hdfs.

In the third part we describe recent improvements in the data access frontends, namely nfs4.1/pNFS, http/webdav and a new xrootd plugin. These will soon become the recommended protocols, even if support for rpio and gridftp access will keep being supported.

Before presenting the conclusions and future work, we describe the additional packages we added to the system offering configuration, management and monitoring of a DPM system using widely used tools. In this case, we have chosen Puppet[13] for configuration and Nagios[12] for monitoring.

Validation of these new efforts has been ongoing following the release early, release often principle. Components in beta state are available in a dedicated repository[4] and follow a monthly release cycle, with several sites participating in the testing effort. At the same time, a big re-packaging effort was undertaken, which allowed us to also start distributing our software using popular linux distributions like Fedora/EPEL[7]. We also organize frequent webinars[6] where new functionality and common practices are presented to the community, either by developers or by community members themselves. Finally, we have started organizing periodic DPM workshops[3], where developers, users and site administrators gather for one day to exchange ideas and present recent work and findings.

2. Evaluation
The architecture of DPM is based on a set of daemons running on two types of nodes: the Head Node includes all daemons handling file, replica, pool and filesystem metadata; the Disk Node includes all daemons handling physical data access. Large deployments split the daemons in multiple machines, or even deploy multiple instances of the same daemon, but conceptually they are grouped into these two types, as summarized in Figure 1.

![Figure 1. DPM Legacy Architecture](image)

Some of the issues identified below come from an analysis of the architecture itself. In other cases, they’re the result of targeted performance tests. For the latter, as well as the performance
results presented in the end of this paper, we have developed a performance suite tool called PerfSuite[5].

2.1. Dependency on the NS and DPM daemons
The current setup of DPM mandates a dependency on the dpns (the nameserver) and the dpm (pool and balancing) daemons for all other components. This means that all additional daemons have the additional overhead of accessing metadata information via the network, instead of directly contacting the source of the information (in this case a SQL database). It also implies that the scalability of the whole system can be no better than the one of these two daemons.

As we moved on to standard protocols and building blocks, with well tested and highly optimized frontends such as Apache2, we considered a refactoring of the functionality offered by these two daemons, consolidating it into a library called dmlite. The result of this effort is reported in Section 3.

2.2. Limitations on the legacy daemons

![Figure 2. TCP Listening Queue and Threads Performance](image)

Even if we are moving to standard building blocks, we still focus on doing valid improvements to the current daemons. To evaluate their performance we have done a set of stress tests to measure latency and maximum number of requests with a variable number of concurrent clients.

In the left plot of figure 2 we can see that there was a significant overhead on the client due to a bad setup of the tcp listening queue. The queue length was kept explicitly low to reduce the number of connection reset by peer client errors on server failure. Recent tcp implementations are very fast and this effect is less significant, allowing us to increase the default queue size and to keep a longer client request backlog. This had a big impact on the experienced waiting times for clients for any kind of request to the nameserver.

An additional change was to add the ability to specify a variable number of listening threads on the main daemons, which was previously statically set to a rather low value. This is relevant as given the change above we now have the potential to have more simultaneous client requests. The plot on the right side shows that with a higher number of threads (70) the waiting times are significantly reduced (second bar group), as is the overall request time seen by the client - the stat calls shown are done in parallel by the client with the put request, and we can see that with the new setup the first call already finds the new file.

The implementation is not nearly as flexible as the Apache Multi-Processing Modules[1], but we do rely on the latter on some of the frontends described in the rest of this paper.
2.3. Asynchronous GET

Asynchronous VS Synchronous get
(20 Server threads)

Impact of database connection pool
on DPNs performance

Figure 3. Sync GET and Connection Pool Performance

Continuing on the topic of client latency, one additional issue discovered during our evaluation (and previously reported by some of our users) was the need to always do GET requests asynchronously, with clients polling until their request was ready. This model is valid when the backend requires some preparation of the file replicas for later access, but as DPM is a disk only storage system, with replicas being always online, this was a useless overhead for clients.

We have changed the system enabling synchronous requests for data access, with significant improvements as shown in Figure 3.

2.4. Database access
We have identified two main issues regarding database access in DPM.

(i) Lack of bind variable usage
(ii) Lack of database connection pooling

The first part has been fixed in our new dmlite based implementation, with the refactored nameserver and pool management functionality. Queries have been fully reviewed, and other improvements other than usage of bind variables everywhere are also part of the new releases.

Regarding the database connection pool, DPM daemons had a one-to-one mapping between the number of threads serving client requests and the number of connections to the database. In practice, every single thread had its own database connection, which it would hold for all its lifetime. As request processing involves more than simply database access, resource usage was under-optimized.

We have fixed this in two ways: the new dmlite implementations make of course usage of a proper database pool; and we have added pool support to the legacy daemons, as an optional feature to be enabled by administrators to ensure system stability. Figure 3 shows the benefits of using a database pool for the nameserver, results being as expected.

2.5. Dependency on SRM
The usage of the Storage Manager Interface (SRM)[14] imposes a significant overhead on client data access, especially for the use case of sequentially accessing multiple small files. Clients first need to access the SRM frontend using the Storage URL (SURL) to get the Transfer URL
(TURL) which is then used to access the physical data using one of the supported transfer protocols.

While still supporting the access model described above, we have chosen standard protocols which can be used both for metadata and physical data access, with the client being automatically redirected - HTTP and NFS4.1/pNFS. In fact, other protocols like GridFTP and RFIO can also achieve this functionality, and we will start promoting their usage for metadata access within the HEP community, our larger user base.

3. Future-proof with dmlite
As described in Section 2, an analysis of the current DPM implementation showed the potential for significant improvement by refactoring its core functionality into an independent library. The initial goal was to remove the dependency on the two main legacy daemons for the new components in development at the time, and to make the core code more manageable. But quickly we understood the potential to extend this effort so that we could open DPM to adopt standard building blocks, and to make it more flexible in terms of the storage backends it supports.

Our new library is called dmlite, and it will become the core of the DPM system in upcoming releases. In this section we describe its architecture, the different interfaces available and the multiple plugins implemented until now for each of these areas. We also present preliminary performance numbers showing this effort will have a big impact not only on the flexibility offered to system administrators, but also on the performance seen by the users of the system.

3.1. Architecture
The dmlite library is now the core of the DPM system. It is either already used or will be very soon by every single frontend, providing optimized and consistent access to both the metadata and the data of the system.

The first step in the design of the library was to abstract the functionality offered by our storage system, breaking it into multiple interfaces that can then be implemented by the different plugins. Figure 4 summarizes the frontends, the available interfaces and the existing plugins.

![Figure 4. dmlite interfaces and plugins](image)

3.2. Interfaces and Plugins
Namespace Management handles all the directory, file and replica metadata. It offers an interface close to POSIX, with functions like `stat()`, `open/read/closedir()`, etc. Both basic permissions and
ACLS are also exposed and can be managed using this interface, following the implementation of the legacy dpnsdaemon.

Pool Management refers to the access and management of the storage pools. Offered functionality includes listing of pools and pool metadata; adding, removing and updating pools. It maps the functionality previously offered by the legacy dpm daemon, but separates the pool from the filesystem definition, which were previously kept tightly together. This opens the way to alternative pool backends such as the ones offered by the HDFS and S3 plugins described below.

Pool Driver is the interface exposing the management of a specific pool backend type. It includes functionality for listing pool information such as total and available space, and management of file replicas (addition, removal and listing). One important note is that there is now a separation between the file replica as defined in the namespace database and the actual physical replica. The former points simply to a pool hosting that file, and the latter is given by an additional call to the Pool Driver implementation, which will reply with either a single location or a layout of the file distribution (in case the backend is a clustered filesystem such as HDFS).

I/O is the pure POSIX based access to the file data. This is required so that we can make use of the backend client APIs for file access, useful in cases such as HDFS where the access to the file can be optimized. In cases like the Legacy DPM implementation, the plugin implementation is simply redirecting the calls to the local filesystem access equivalents.

Our implementation allows the runtime usage of more than one plugin for the same interface. Take the example of both the MySQL and Memcache plugins being configured, with the latter having priority as expected. On a stat() call, if Memcache returns the file metadata (a cache hit), then dmlite simply returns that value. On the other hand, if it would return an error indicating the file did not exist (a cache miss), then it would forward the call to the next plugin (MySQL in this case), retrieving the information to the client and populating the cache. This principle is valid for any number of plugins, which act as a sort of stack.

Listing 1 shows an example configuration of dmlite and the plugins to be loaded.

```
# First, the plugins to be loaded have to be specified. The order is relevant: the last plugin
# in the stack will be the first called by the library (LIFO). In this case, the requests will be
# processed by a profiler (prints method called and timing), then Memcache, then MySQL backend,
# and at last, the DPM adapter (calls back to the legacy DPM daemons).
LoadPlugin plugin_legacy_dpm /usr/lib64/dmlite/plugin_legacy.so
LoadPlugin plugin_mysql_dpm /usr/lib64/dmlite/plugin_mysql.so
LoadPlugin plugin_memcache_dpm /usr/lib64/dmlite/plugin_memcache.so
LoadPlugin plugin_profiler /usr/lib64/dmlite/plugin_profiler.so

# This parameter is used by both plugin_adapter and plugin_mysql, to know
# to which host they must connect.
Host localhost

# Here, the MySQL connection parameters are configured. The user must have
# access to dpm_db and cnr_db.
# MySQL user name
MySQLUsername username
# MySQL user password
MySqlPassword password
```

Listing 1. dmlite configuration sample

In the rest of this section we give some details on each available plugin. We have others in earlier phases of design or initial ideas, which should help us further improving the system’s performance and adding extra functionality.

3.2.1. Legacy DPM The first plugin we have implemented is the simplest one. It implements all defined interfaces, forwarding all the requests to the legacy namespace and pool daemons (dpnsdaemon and dpm). There is no additional logic involved, and performance values and expected scalability remains the same as the case of doing direct calls to the service daemons. In the case of the I/O interface, the plugin simply forwards the POSIX access calls to the locally mounted filesystems, as this is the way the native DPM pool management works.
This has allowed us to evolve the remaining plugins progressively, as non-implemented calls would consistently fallback to this plugin (thanks to the dmlite architecture).

3.2.2. MySQL / Oracle  
This is the most significant refactoring effort from the legacy code. Both plugins implement the Namespace Management and Pool Management interfaces, with the logic staying as close as possible to the one provided by the legacy daemons. But it is also where the first significant performance improvements were introduced, thanks to the usage of a database connection pool, bind variables, and the fact that frontends could now talk directly to the database, instead of going through the network service. This was not possible before, and it was limiting significantly the performance of our new standards based frontends.

In here we focus in the MySQL implementation as it is the most widely deployed, but similar observations could be done for the Oracle one.

![Figure 5. MySQL and DAV MySQL vs Legacy DPNS performance](image)

The left plot in figure 5 shows a comparison between doing stat, delete and add calls via the dpnsdaemon running on the same machine as the database (the way we had to do it before for the new frontends), and directly through dmlite (the new way).

The results show a dramatic improvement, for multiple reasons:

- Bind variable usage is especially relevant on repetitive queries, and the three calls above are very good examples of this - the same execution plan can be reused each time
- Accessing the data via the dpnsdaemon imposed the additional overhead of using a socket for the communication
- Accessing the data via the dpnsdaemon includes marshalling of the response / request data, while dmlite is simply going to the database. Still, the comparison is valid as these are exactly the two workflows that the new frontends we have developed have as alternatives to access the nameserver data

This simple plot validates all the refactoring effort we have performed with dmlite, even if there are many other benefits as presented in the rest of this article. The right plot in figure 5 shows a performance comparison between access to the namespace data via DAV with a dmlite based MySQL backend and the legacy dpnsdaemon. Results can be interpreted as follows:

- The performance of our new implementation is especially interesting considering the much higher number of operations per second for the stat call, the most common request
• This time the comparison includes the processing of the DAV XML request and responses, which impose a bigger overhead compared to the simpler wire protocol of the dpdnsdaemon. Still, we can easily see that the improvement is not as high as we could expect. This is due to the mod_dav Apache module implementation on which we base our frontend. We will put significant effort in further improving this component.

• The plot also shows an upper threshold in create and delete operations. This is understood, and is due to an obligatory stat call before creation/deletion in the current DAV setup, which does not exist in the legacy daemon. This will disappear in a future version and we expect the performance difference to be similar to the one for stat calls.

Even if this is the first release of the MySQL plugin after the refactored database access, there is already a significant improvement in performance. We expect even bigger improvements as we now start focusing on further optimizing the database queries.

3.2.3. Memcache
Adding a caching layer to the legacy nameserver daemon would not be a trivial task, but with the new dmilite architecture it became a matter of simply wrapping the Namespace Management calls against a memcached setup.

The left plot in Figure 6 shows the performance improvement of enabling the memcache plugin on top of a dmilite setup with a local MySQL backend. We can see an improvement of up to 7 times in the response time of a stat call with a small number of clients. The decrease in the local memcache performance with a higher number of clients is due to the clients performing the tests being setup in the same machine. The other interesting observation is the usage of a remote memcached server, which seems to reach its maximum performance with a higher number of concurrent clients than the direct database access.

The right plot shows a more detailed comparison, focusing on direct calls to the dmilite library. It shows that for read only calls like stat or replica requests, there is an overhead when there is a cache miss, and that overhead is more significant if the memcached server is accessed over the network. In the case of a cache hit, then the performance gain is between 5 and 10 times with a local memcached server, less if over the network. It also shows the overhead of update calls such as file creation or removal.

Figure 7 does the same comparison focusing on the readDir call. From the left plot we see a cache miss has an overhead as expected, but the most interesting result is the difference between
what we have defined as strict and relaxed access on a cache hit. In strict mode, the call will update the access time of every single file in the directory, meaning it has to do an additional call to the MySQL plugin for every entry in the directory. In relaxed mode, only the access time of the directory is updated, and there we can expect a performance gain of at least 3 times. Given the potential gain of the latter, we have made this configurable in the plugin, and have started working on providing such a behavior in a generic way to all plugins. The right plot shows the potential gain of using memcache with different number of files in a single directory. The performance improvement goes from 2 to 5 times, and memcache handles large directories better than the equivalent direct database access.

3.2.4. Hadoop / HDFS  Apart from the performance improvements shown in the plugins above, we also have the potential with the new architecture to improve both the feature set and the scalability of the overall system by reusing standard building blocks. The first example of this is the integration of a Hadoop/HDFS[9] pool backend via a dmlite plugin implementing the Pool Driver and I/O interfaces.

The goal is to have an additional pool flavor in an existing DPM setup having already native DPM pools, while keeping a single namespace for the whole system. The new pool can be managed independently using native Hadoop tools and libraries, and will offer its rich feature set, including periodic checksum checks, automatic hot file replication, etc. All of this while keeping the DPM extras on top, including all the grid specific requirements (authentication, authorization, grid specific frontends, standard protocol frontends) and the DPM system management tools like disk server draining.

Figure 8 shows the workflow required for accessing an existing file hosted in a HDFS pool using our DAV frontend - the workflow for a write is similar, with an extra step at the end to validate the metadata on the nameserver. Two interesting observations:

(i) There are significant differences compared to the equivalent workflow of a native DPM pool - the main one being that only one replica entry per pool is kept in the nameserver, while for native DPM we keep one entry per filesystem holding a replica of a file. We instead rely on the HDFS namenode to retrieve the actual replica information, and leave it open for it to reorganize the physical data as it finds appropriate

(ii) We currently always redirect the client to one single disk server, from which it reads the whole file using the HDFS client API. As HDFS splits the file in chunks and distributes them in the whole system, we end up reading a lot of data from the network - it will access
remote disk servers for every chunk it does not own locally. This is a significant overhead, and we’re investigating semantics on the HTTP reply to have the client reading ranges of the file (the chunks) from different disk servers.

This is still experimental work, but we expect it to become a valid option in the near future.

3.2.5. Additional Plugins

One useful plugin not shown in the initial architecture picture is the profiler. This is a simple plugin making use of the dmlite architecture to provide detailed tracing on the multiple requests being done to the library. It can be put before or after any other plugin, and can appear multiple times in the stack if desired. It has been the main source of data for the performance tests previously shown.

We have also started looking at additional plugins that could further improve the system. The most interesting option would be to add opportunistic or temporary storage to an existing system. This would significantly help small sites which suffer from short periods of much higher load than the average, as is the case when a popular conference date approaches in a given scientific community.

The most valid option to fulfill this use case is to add support for additional pool backends supported by cloud storage. As S3 is currently the most popular interface to access data on the cloud, we will be focusing on this one first, and expect the workflows to be close to the one described for HDFS. Adding support for other cloud interfaces later should not be a big task.

4. Standards based frontends

As previously mentioned, one area we have been putting significant effort on is providing standards based client access to the DPM storage system. The main reasons for this effort are (among many others):

- **Accessibility**: not having to package and distribute our own clients, and relying on the open source community to make them available in all popular architectures and platforms
- **Validation**: relying on our own custom protocols has meant a big validation effort between the different storage implementations. This effort is still there when using standard protocols, but is reduced since the test and validation suites are provided by all interested partners
- **Stability**: standard protocols are kept under extensive scrutiny
• **Ease of implementation:** we can base our systems in common libraries and services
• **No vendor lock-in**

The migration to standard protocols should help us reduce existing library dependency issues, remove the requirement for *user interface* machines where all custom clients are made available, and to have a much easier migration path to new versions of operating systems.

Our evaluation took the two ways we have to access the data:

• **POSIX or POSIX like**, such as previously offered by our *rfio* and *xrootd* frontends
• **GET/PUT style**, as offered by the existing GridFTP frontend

In the rest of this section we provide implementation and performance details of our new standard based frontends.

4.1. Access via **HTTP/DAV**

HTTP[10] is one of the most popular protocols in use for remote data access. Clients and base implementations are available for a very wide variety of systems and architectures, and in many different languages.

Mostly used to access small files, the standard has all the required features to have it serving as a file transfer and access protocol. With the addition of the WebDAV[15] set of extensions, the same system can also be used to manage file metadata, making it a full alternative to both our legacy *dpnsdaemon* and the *gridftp* daemon.

Figure 9. HTTP / DAV performance comparison

Figure 5 already showed the comparison with the legacy *dpnsdaemon* for metadata access. In figure 9 we present the comparison with GridFTP for both reading and writing files. In the first case performance is significantly better, even when compared to the usage of UberFTP which proper uses the FTP *active mode*. In the second case, the performance is similar to the one achieved by using the native GridFTP client.

Check [16] for detailed information on this implementation. It includes more results on performance, and detailed data on how we provide features like *multiple stream transfers*, *third party copies* and how we offer a grid wide *global access service* based on HTTP.
4.2. POSIX access with NFS 4.1/pNFS

NFS4.1/pNFS was an obvious match for our requirements. It provides strong authentication based on user credentials, separation between metadata and data access, a (possibly even global) hierarchical namespace, and high performance - including client side kernel caching.

As expected we did not have to write our own implementation from scratch. We analyzed the available alternatives and chose Ganesha\[8\], a NFS server running in userspace. It supports multiple NFS versions, including 4.1 which is the one we had interest on. It lacked at the time full support for pNFS, which we initially added. Later on a lot of effort from other partners - IBM, Panasas, Linuxbox - was put in improving pNFS support, and the system in general. Figure 10 shows the basic architecture for data access using NFS4.1/pNFS. The client first talks to the metadata server, which in our case corresponds to the head node, asking for file access for reading or writing. The server replies with a layout, which describes the distribution of the data in the multiple disk servers.

The protocol does support different kinds of layouts, namely block, object and file. Our implementation currently supports only the latter, as this is how data is stored in the DPM system - but the client can access different parts of the same file in parallel from replicas in different disk servers, and we will be adding similar functionality to access chunked data in clustered filesystems via dmlite.

The performance of our system is shown in figure 11. The left plot shows the results from IOZone\[11\], and it shows that we manage to saturate the network link. On the right plot we show a comparison with rfio, and again we either perform as well or better.

The NFS4.1/pNFS frontend to DPM is production ready in read-only mode, with the option of enabling writing for sites that desire it. There are two main issues we currently deal with:

- The linux client supporting pNFS is available from kernel 2.6.32 and above, which means the current systems in production in most of our sites are not good enough. RHEL 6.2 ships with this kernel version, so we can expect extensive client availability very soon
- The strong authentication support is based on the GSSAPI. Most linux distributions ship with a library called gssglue which can load multiple GSSAPI implementations, and include a KRB5 plugin. Unfortunately, support for X509 certificates is not easily available, and we will be focusing on providing one to our community
5. Monitoring and Configuration
The final area we have put significant effort in the last year and a half is related to service configuration and monitoring. As with the rest of the system, we tried to reuse existing building blocks as much as possible. After a careful evaluation of the existing options, we have chosen Nagios[12] as the base for our monitoring probes, and we have added an alternative configuration solution based on Puppet[13].

5.1. Monitoring with Nagios
The choice of Nagios was straightforward, as it was already the most commonly used solution in sites running our system. It has a very big and active community, and a lot of support in other tools - Puppet is a good example.

We have implemented a large set of probes covering each of the node types in DPM:
- **Head Node**, with probes covering metadata queries (per request type), pool information (usage, available space, ...), space token usage, etc
- **Disk Node**, with probes covering transfer throughput (per protocol, per VO, ...), local partition usage, etc
- **Nagios Host**, with probes requiring remote access, such as service availability checks (ping for head and disk node daemons)

In addition to the service availability and status monitoring, some of our probes also generate additional performance data. Figure 12 shows examples for the most common requested operations and the read throughput of the system, distinguishing between large, medium and small files.

![NFS Frontend Performance](image)

**Figure 11. NFS Frontend Performance**

![System performance with Nagios](image)

**Figure 12. System performance with Nagios**
In most cases, the probes have been written in Python, and we have a set of common modules that reduce the effort of adding new probes. In cases where there was already an existing probe in the community for a specific function, we have simply integrated the code into the DPM nagios packages. The new monitoring based on Nagios has been well received by sites, and is one of the areas where community contributions have been very significant.

5.2. Configuration with Puppet

As some of the DPM sites started having larger deployments, many in the multi-petabyte range, we looked into providing a better configuration solution than yaim. The two options were Chef[2] and Puppet[13]. We chose the latter mostly due to the very good documentation available, and this choice has proven successful as some DPM sites start moving to Puppet even for their basic fabric infrastructure.

Puppet is designed to manage system configuration declaratively. It takes as input the system resources and their state, defined in files called manifests. It then compiles them into a system catalog which is applied against the target systems, and provides the administrator with reports on these actions.

Much of the required configuration is already available in Puppet’s common modules or as community contributions, but we did have to develop a set of additional manifests covering grid and DPM specific configuration. These include modules for DPM, VOMS, gLite, EMI, and an additional MySQL module as no good candidate existed at the time (this has changed in the meantime).

Figure 13 shows an example of a cluster setup.

6. Conclusions and Future Work

The DPM system has evolved with the grid itself, staying as the most widely used grid storage system. The software has adapted overtime to new requirements from both users and storage providers, but the core of the system was recently becoming a bottleneck in face of large scale data analysis, where a larger number of files is accessed simultaneously.

We have described recent efforts to make our system future proof. We have introduced dmlite, a plugin based set of libraries which is now the new core of the system. We described the multiple plugins currently available, and presented results showing that the scalability and performance of the overall system has improved significantly. We described new plugins focusing on extended functionality, integrating our system with other popular storage systems like Hadoop.

Finally, we summarized the status of our efforts in moving towards standards based access and building blocks. We showed that our HTTP and NFS frontends are at least as performant as the legacy frontends previously available, while significantly reducing the maintenance effort on the server side, and removing the need to deploy and maintain custom clients.

Upcoming months will see a strong effort in wide deployment of these new tools and libraries. We have setup a testing infrastructure within the community where we can easily evaluate new features in a production like environment. The time for actual production deployment is to be kept as short as possible, ideally less than two weeks.
Other future work will focus both on extending functionality and improving performance. For the first part we will continue extending dmlite with new plugins suiting requests from existing grid deployments - one common request is integration with the Lustre storage system. For the second, we will concentrate on improving even more the data access performance using the standards based frontends, and making sure we have good integration with popular data analysis frameworks like ROOT. A full evaluation of data access using vector reads, asynchronous readaheads and parallelized access will be performed to fulfill this purpose.

7. Acknowledgement
This work was partially funded by the EMI project under European Commission Grant Agreement INFSO-RI-261611.

8. References