Performance Measurement and Analysis of the Grid Storage Manager DPM

Author: Martin Hellmich

CERN openlab

12/08/2011

Contents

Abstract

The grid storage manager DPM is used successfully at many Tier 2 sites in the WLCG. With Taiwan as the first Tier 1 deploying DPM, the performance requirements are increasing.

This projects extends the performance measurement suite Perfsuite [1] and implements tools for a performance analysis of DPM. The resulting understanding of DPM from this project assists the developers for further improvements of DPM and also provides ideas and tools for DPM performance measurement. All software is available through the developer repository and most tests have been included in the latest Perfsuite release.

1 Introduction

CERN [2] is certainly most-known for running Large Hadron Collider (LHC), the worlds largest particle accelerator. Through it, the most pressing questions in particle physics, such as the existence of the Higgs boson or the behaviour of anti-matter, should be answered. For this, very sensitive detectors related to the six experiments on the LHC, record the particle collisions.

These collisions, occurring at a very high rate, produce an enormous amount of data, which has to be saved and later analysed. The ATLAS experiment [3] alone has a data rate of more than 18 GByte per minute.

All data is first collected at CERN, but for the long time archiving and the analysis of the data by physicists in institutions world-wide, the World Wide LHC Computing Grid (WLCG) has been established. This two tier computing grid provides about 200,000 processors on more than 140 sites around the globe [4].

Another matter beside computing on the grid is storage. Storage solutions must be able to allow high-performance access to large amounts of data, both for sequential and random access. They also must provide various transfer methods to suit the needs of the researchers from different institutes and LHC experiments.

DPM [5] is a storage solution mainly used at Tier 2 sites as it is scalable and well performing, yet easy to administer. Also Tier 1 sites need tape management for long term archival, which DPM does not support. However, DPM can be a interesting solution if Tier 1 centres distinguish between archiving and live file access and employ two specialised systems.

This project aims at understanding the DPM architecture through analysis and performance measurement. It builds on the performance measurement suite Perfsuite [1] and uses different measurement environments. During the project, performance factors of DPM could be discovered which help evaluating future DPM developments.

2 Implementation

This chapter covers the description of the implementation of the tools necessary for the performance measurements taken during the project. It is separated in two parts: the first introduces Perfsuite, a performance measurement suite used for the development, the second the implementation of tests and other tools. The next setion then describes the setup of the testing environment and a description of how the tests were conducted.

2.1 Perfsuite

The DPM performance testing suite has been developed at the IT-GT-DMS group at CERN by Alexandre Beche and the author has contributed to its current version 0.2.0. It is available online [1] with support.

Perfsuite is a framework to run test cases, which can be written in any language and have arbitrary functionality, but have to provide their output in a specific format. The tests are then invoked through a central configuration file, which allows tests to be repeated and run with different command line arguments.

Current tests written for file transfer measurements include one part for ‘putting’ files to the DPM storage and one for ‘getting’. Most of them create random files for the tests, but the developed tests here alternatively use files specified in the Perfsuite configuration file.

2.2 Extensions to Perfsuite

Perfsuite as it is shows to be a very good tool to run multithreaded file transfer tests with throughput results from a single client. The plugin structure for tests and especially the common template makes it very easy to develop and deploy your tests. For distributed testing and a detailed analysis of the tests, changes to Perfsuite had to be made and a surrounding execution environment implemented. In the following, the author’s tests and extensions made to Perfsuite are described in more detail.

2.2.1 Test for RFCP

This test uses the command line programme `rfcp` to copy files to DPM and back to the local disk. It exists in two flavours: one takes a local file and copies it, the other, using the common template, generates files of the desired size for copying. It runs multithreaded and reads from one local file, but

test name	function
rfcp	put and get autogenerated files
rfget	get one file
rfgetmany	get different files

Table 1: The three rfcp test flavours

changes the designated file name on the server accordingly so that n files get created. For the reading part, also two version exist: either all client threads read different files, the file names coming from the files put on DPM in the first place, or they all read the same file. The latter reproduces the effect of a file being used by many analyses at the same time. Table 1 shows the test types.

To exclude the client's disk as a limiting factor here, all output is written to `/dev/null`. The disk speed can of course be a factor for the 'put' test and has to be considered.

Using the command line client RFCP instead of a self-written program gives the advantage of having the same circumstances as in a production environment. Experiments use the same tool to copy files to the computing nodes when working with RFIO. This way the test profits from the same speed ups in newer version of rfcp as real jobs do, e.g. the ability to open multiple streams when sending a file to saturate the network in RFIO version 3.

All these tests forward the throughput reports from `rfcp` to Perfsuite and comparative measurements can be done using the timestamps that Perfsuite itself records.

2.2.2 Test for Pretend RFCP

This test uses the functionality of the DPM API to simulate creating/writing or reading files with the RFIO protocol. The test requests a file on DPM either with the `dpm_get` or `dpm_put` command, polls for it with `dpm_getstatus`, opens the file with `rfio_open`, but closes it immediately and signals a successful file transfer with `dpm_putdone`.

This behaviour allows us to test the DPM performance with many concurrent clients simulating real-world file access patterns, while putting almost no load on the clients or the DPM disk nodes. Also, as no data is transferred, the network throughput from the client to the server remains limited.

The test has been fully instrumented with respect to the client - DPM interaction: A precise timestamp is taken before and after executing each of the three DPM API calls and the durations of each call are reported as well as the timestamps themselves. Together with similar information taken from the DPM log files, this gives a detailed view on the internal behaviour.

2.2.3 Test for Root over RFIO

This test represents partial read of a Root file on DPM. It uses the C program IOPerformerGrid provided to me by Wahid Bhimji which I developed further to accommodate different test situations. The program is used in a python wrapper allowing a multithreaded test just like the others.

In a test, either all events in the file are read or only a fraction of them. For specifying the read ratio, the tester gives a fraction and the stdlib.h random function is used to decide which events should be read. The resulting event reads are distributed uniformly over the file length. Additionally, one can choose between reading all branches of the events or only a subset of them.

The root version used is 5.26.00 with two different files. Both are taken from the ATLAS experiment and contain the same data but one, as indicated by the filename 'ByEntry' is ordered, the other file is not. The ordering influences the file access patterns, especially the possibility of sequential reads, which is important for the file read speed.

The output is given through the `TTreePerfStats` module of Root, which presents detailed information about the overall the read duration, disk usage and the impact of the file compression among others.

2.2.4 Test for NFS

The test designated for NFS is the simplest one developed, using `/bin/cp` to copy files. It can be run in Perfsuite to measure the throughput for a locally mounted file system like NFS.

The report values come from the timestamps recorded before and after the test. A short discussion about the accuracy of these timestamps is given in the following section about changes to Perfsuite.

2.3 DPM Logging

During the test, the DPM log files are examined to give detailed information about the internals when issuing a file request. I focussed on extracting information from the DPM daemon running on the head node and left the log files of the DPNS.

Parsing the DPNS log files is an easy task because the DPNS serves requests only synchronously. Tools parsing these logs exist at CERN to generate information used by the system monitoring tool

NAGIOS [6]. These NAGIOS plugins examine the logs immediate past and report the average request duration which triggers an alarm to the system administrator if it is above a threshold. While such tools help the administration, the DPNS log file was of little use for these detailed system analyses.

In my tests, the DPNS was only accessed indirectly through the DPM daemon. These interactions are documented in the DPM log, but only by mentioning the DPNS function used without a possibility to link this specific API call to the entry of a call in the DPNS log file. While this is a drawback in understanding the complete interaction between the DPM and the DPNS in one request, I found out that the DPM log itself holds a sufficient amount of detailed information for the analysis of the system. Therefore parsing the DPM log file is an important part of the project.

2.3.1 DPM Log Parsing

Due to the asynchronous requests the DPM uses, log file parsing is not a trivial task. I developed a script to do that with focus on ease of use, flexibility and re-use. The script contains one parsing loop, in which the contents of the log file are grouped by request so that every entry corresponds to one file interaction. I will explain the case for a 'put' request to DPM, the 'get' case is similar.

Every request consists of a `dpm_put` issued by a client and at least one `dpm_putstatus` command, with which the client polls for the physical location of the file on DPM. As one `dpm_putstatus` command is successful, the client engages in direct interaction with the corresponding disk server and only returns to the head node after all modifications have finished and transmits a `dpm_putdone`. After the `dpm_put`, which returns a 128-bit UUID to the client, the server starts a `dpm_proc_put` command to check if the file exists and for sufficient rights and retrieves the file's physical location (TURL, transfer URL). This function calls the DPNS several times to do this, since all these information are stored in the name server database. The file's UUID is used in the `dpm_putstatus` call as reference to retrieve the right file location. In the DPM log, all status requests are stored and they finish with return of QUEUED if the processing method has not started yet, ACTIVE if it is running or SUCCESS if the TURL is ready.

The challenging part for parsing is that multiple threads are involved in one request as the client opens several connections to DPM which are each processed just by the next free thread. For sure, the `dpm_proc_put` method is run in a separate thread taken from the slow threads pool. To chain them all together, the parser has to find the file's UUID in the first connection and then group according to these. Additionally, it has to be taken care that no intermediate results get deleted if a thread takes on another request, before the first file interaction has been completed.

The parser also has to be sensible to the fact that requests can overlap: a `dpm_putstatus` can occur during the internal processing of this request in a `dpm_proc_put`. The resulting script serializes those requests in a way that log messages belonging to one method are always next to each other, independent of the timestamp. This decision has the advantage that getting the duration information for one method call is easy, but estimating the time between function calls is more difficult because the entries have to be reordered. Fortunately, the overlapping only occurs for the `dpm_putstatus` call where the time relative to other functions was of lesser interest — the time between functions calls can be measured on the client more easily and the other functions reappear in the parsed log in the same order as they have been executed on the server.

It is notable that the script stores all log messages for one interaction through the assumption that all output coming one specific DPM thread must belong to the same function if the function has not ended with a return signal. This is possible, because all DPM methods log their return value in a log line containing the string "returns". Also, functions which are not used for later analysis are kept.

This is made necessary by a change to DPM to issue a log message as soon as an incoming client request is allocated to a thread. The message gives us information about the duration of the authentication process, as a thread accepts a request, then runs the authentication and then starts the desired function. By comparing the timestamp of the acceptance to the first timestamp of the function executed, an estimate of the time spent for authentication can be made.

2.3.2 The Parsing Script Usage

The script is written in Python, reads the DPM log file from standard input and prints its results to standard output as shown below.

```
cat dpm-log-file.txt | ./a_dpm_parse_log.py -r put > dpm_out.csv
```

It uses, as mentioned before, one loop in the main method to aggregate results in a dictionary. These results are then given to an analysis function, which calculates the durations of the `dpm_put`, `dpm_putdone`, `dpm_putdone` and `putstatus` functions as well as the durations between the executions of the first three. The script is able to handle multiple status requests, as it might be necessary if the processing takes too long. The output file consists of comma-separated values with the header files stored in the last row.

In future versions of DPM, the log file format might change. Even now in the to-be-released version 1.8.2 it is possible to choose between the internal DPM logging mechanism `logit` and using the syslog daemon. Different substitutes for the syslogd like rsyslogd or syslog-ng bring more format with it. During my tests, for example, I used rsyslogd to benefit from higher resolution timestamps than provided by DPM's `logit`. The parsing script is adaptable to changing formats as it relies on regular expressions to parse the lines. As long as the information needed, e.g. the DPM process and thread number, the file UUIDs, etc., is still in the logs, changing these regular expression is sufficient to adapt the script to a new format.

With multiple purposes in mind the script has been written so that the analysis method can be easily exchanged or rewritten with another focus. By changing the behaviour of the parsing algorithm so that complete server-client interactions are not stored, but immediately printed out, the script could be used for streaming parsing of DPM logs, with the analysis part moved to another stage of the pipeline. It has, however, to be shown that the amount of data from the logs is not too heavy for a streaming application.

2.3.3 DPM Logging Conclusion

As mentioned before, the DPM log file provides an investigator with all information needed for a detailed profiling of the head node itself. It records the beginning and the end of each function, as with a change to DPM, which might be incorporated in future versions, the start of the authentication as well. Therefore, the log file parsing has been the only development necessary to receive this information.

The script itself can be used on an analysis computer with a partial log file, but, as mentioned, can also be changed for use on the DPM head node to provide real-time information. The direction, which is likely to be pursued at CERN is the conversion into a NAGIOS monitoring probe. A future change in DPM might include using unique identifiers for the function calls from DPM to the name server to create a full overview over the methods and their durations or failures during one client access.

3 DPM Testing and Analysis Setup

After the last chapter introduced the software developed for DPM testing, this part reports the test setup with hard- and software used, the test execution and then discusses their results. The test discussion focuses on one DPM parameter which has been investigated thoroughly and whose test results influence the future development of DPM. For the other tests I provide data which shows their correct behaviour and the success to run Perfsuite in a distributed testing environment.

3.1 DPM Setup

The DPM head node runs the DPM and the DPNS daemon in version 1.8.2 from the development repository of DPM. The disk nodes still run the 1.8.0 versions of the file transfer protocol plugins, which are compatible with the new head node. The DPNS runs, as not to be the bottleneck of the tests, with the maximum of 99 threads, while DPM runs with the 20/20 configuration for fast and slow threads.

Both daemons are placed on the same machine and one disk server is attached to the setup using a single file system for DPM. Another disk server is still attached to the test bed, but is configured as read-only, so it does not affect further tests. DPM then effectively uses one pool with one file system on one disk node.

The computers for the head node and disk node are from the lxfsra test bed setup for DPM performance tests, which will be described shortly, as we also used these machines as clients.

3.2 Client Setup

For the test with multiple clients we had different possibilities where to take the client from with advantages and drawbacks. All configurations were expected to allow certain conclusions about the DPM server and also about preferred configurations for future tests. There were three different configurations available: the lxfsra test bed using the servers on which DPM is installed, 20 dedicated virtual machines and CERN's lxplus machines.

The lxplus environment consists of 75 machines, of which 28 systems have 8 cores and 47 systems have 16 cores. It enables us to run up to 976 instances of every test, if we follow the guideline that only one instance per core is allowed. This is especially sensitive to do as the lxplus machines are not dedicated and a high CPU load is expected on these servers during the tests, which is also why we cannot use this test configuration for client side measurements.

Instead, we use this test bed to put pressure on the DPM, as the advantage is to have a large networking infrastructure at our disposal. So this test bed is useful to run actual copy tests to see whether the dedicated disk node suffers from too many connections or can keep up the throughput.

Because the lxplus machines are shared resources, there are problems accessing them. Some machines did not run the tests at all, others failed during the tests due to other users taking up the resources. Tests run on these machines are probably not reproducible in detail, but give an indication whether the load induced by physical machines with dedicated network connections is different from the other test beds.

The dmstestvm, in contrast, consists of 20 dedicated virtual machines with one core each, running on two hypervisors. The vm hypervisors are kept private to assure that no other vm could be deployed on them during the time of testing. Each hypervisor has access to a 10 GBps network link, which allows us to compare the results with other test setups with 1 GBps each.

The third configuration runs on the same machines as the DPM head and disk node: three former disk nodes are disconnected from the pool and used as client machines. This has the advantage that the client is as powerful as the server and that they are located in the same computing cabinet, therefore having a direct network connection with little unrelated traffic.

All test bed computers are running Scientific Linux 5.5 [7] and tools are taken out of the standard repository. The programs installed, besides the DPM client libraries, are Python 2.6, tcpdump and iperf for initial network speed measurements. A more detailed description of the client hardware can be found in the Appendix 2.

3.3 Distributed Execution

Perfsuite is built to be invoked directly on the command line. When using it on multiple machines at the same time, invoking tests manually is not feasible, so I used `wassh`, an ssh client for simultaneous command execution on multiple clients and the UNIX tool `at` to time the start of a test run.

Also, Perfsuite's features of launching several tests one after another cannot be used as it can not be guaranteed that the second and third test start at the same times on all Perfsuite clients. Now, every test is started separately.

For the tests, the Perfsuite installation resides on my AFS home directory, which is accessible from every client used. This directory is copied to the working directory in `/tmp`, where the results are also written. So in the script managing a test run, I first make sure that all required directories exist, copy

Perfsuite, run it with the desired parameters and then collect the results from all clients with another script.

The program `wassh` is developed at CERN for internal use. It takes a list of host names or a file listing host names as an argument and executes the command given as second argument on all of them. With this, it functions as a couple of ssh commands executed in a for-loop with the advantage of concurrent connections. An advantage of `wassh` is that it prints the standard output of the remote command together with the host name, which makes collecting and parsing the Perfsuite results later simple. It also gives meaningful information about unresponsive hosts and executes correctly for all other machines.

The execution of Perfsuite on the client is managed through another script residing in AFS which launches Perfsuite with the desired configuration file and redirects its status output to a file in the client working directory. The execution of this script, however, is managed by `at`. This UNIX program launches its argument at a given time and by specifying a time in the near future I can be sure that all clients start the test at almost the exact same time. The clock differences on the machines used should be minimal as they all adjust their time to the same NTP server.

Using `at` introduces new problems to running Perfsuite: the command is no longer executed in a shell. Firstly, it makes it impossible to access my AFS home directory, which is the main reason to switch to a working directory in `/tmp` and secondly, shell variables and sourced files set are not valid. Especially sourcing files is problematic, as each Perfsuite test is executed in its own shell environment. Usually the tests inherit the shell environment from the shell Perfsuite is running in, but since Perfsuite is executed in a new environment through `at`, this is no longer possible.

The solution is found in wrapping every test in a shell script that sets the environment variables and then executes its parameters. This keeps changes to Perfsuite to a minimum (calling the script instead of the test directly) and also provides the possibility to print timestamps before and after the test's execution which is discussed in the following section.

Another difficulty in running Perfsuite on various machines is that they mostly do not have the required software installed. Python 2.6, a prerequisite for Perfsuite, is installed on all machines, but the DPM client and Root libraries are not. Since I do not have the rights to install software on these machines, the Root libraries are sourced from AFS.

Due to this, the tests allow to define the executable to run, making the deployment of these files together with Perfsuite easier. Also, the source includes paths that can be changed in the single wrapper script.

Retrieving the results from the Perfsuite instances is a straightforward process: `wassh` reads the result files from each test and prints it to standard output. Because it adds the host name to each output, the result file on the controlling host can easily be parsed to obtain results grouped by host name.

The following section deals with the way Perfsuite opens the tests and its implications on timestamps used for measurements. In Perfsuite, timestamps are collected in the main programme before and after test execution and give another way to, for example, calculate the throughput. Here, accuracy is of importance.

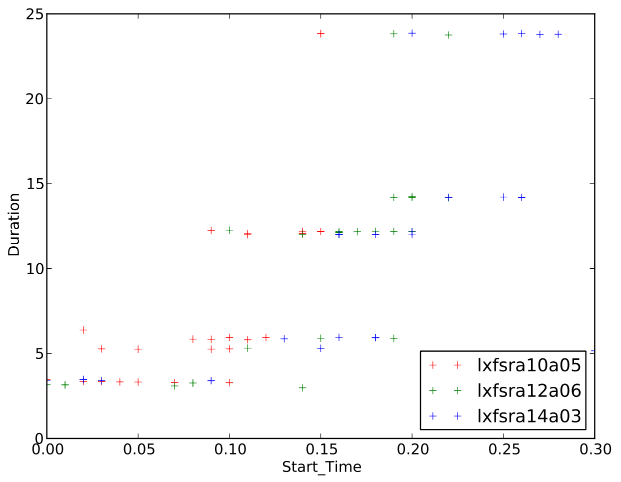


Figure 1: The duration of creating a zero-length file on DPM. The colours differentiate the clients.

4 DPM Performance Analysis

Using the lxfsra test bed and the test `Pretend RFCP`, an analysis of DPM behaviour when many clients create files on the DPM has been performed. The test setup is as follows: three lxfsra clients run Perfsuite with the test which starts 25 threads on each machine to open 25 connections to DPM simultaneously.

Our expectation, given that 75 simultaneous requests are reasonable for a file server and that the file is only opened and then immediately closed, is that the requests finish very fast. They will not complete at the same time, because the DPM daemon only uses 20 threads per type, but creating an empty file should be quick.

We can see, however, that from the client's point of view, the file creation process, from the beginning of the `dpm_put` to the end of the `dpm_putdone` takes a notable long time, up to 20 seconds, as can be seen in Figure 1.

The duration values are grouped by the client machine and we can see that none of them is penalised against the others. The Figure shows the duration the interaction takes on the y-axis corresponding to the start time on the client. In this plot we see that the test does not start at the same time for all threads, but within a reasonable time frame of 250 ms. We also see that the duration increases the later the interaction starts, and it increases in discrete steps. The increase over time suggests a buffer filling up, which is investigated further.

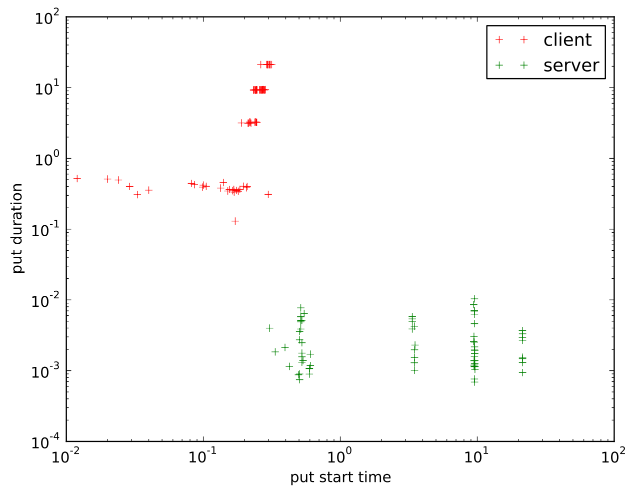
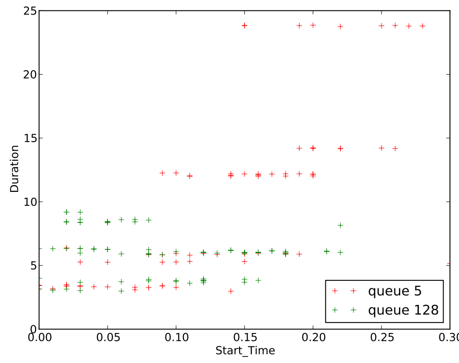


Figure 2: Measurements for the dpm_put on the client and server side. The functions start at the same time on the client, but with a delay on the server. This delay corresponds to the added duration as measured on the client side. Because of large differences in the duration as well as the starting time, both axes use a logarithmic scale.

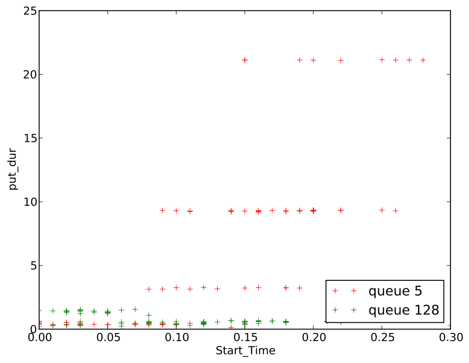
If we inspect the server logs, we can see that the functions do not take enough time to explain the delay on the client. We see that there is a significant difference between the perceived execution time of the `dpm_put` and that the server is contacted with a delay after the client starts the request. This is shown in Figure 2 with the start of the function on the server side in green and on the client side in red. The Figure displays that all requests star on the client at approx. the same time, but start later on the server. We can also see the differences in the duration on the y-axis. To better distinguish between the times measured on the server, the y-axis is shown on a logarithmic scale.

4.1 Increasing the Socket Queue Length

Recording all exchanged packets between server and client shows the connection establishment where we can see that a client needs more than one SYN request to open a connection in about half the cases. A SYN retry is necessary when the server does not respond to the client's request, i.e., the server does not accept the connection, nor denies it. This is the case when the socket queue, which holds connection attempts until they are accepted by the program listening on the server, is full: further attempts to connect to the socket are silently discarded. The results are retries from the clients which follow the regulations given by the specific operating system. Running Scientific Linux, the



(a) Durations for the small socket queue increase over time and remain steady for a large socket queue.



(b) The dpm_put method can be accounted for taking most of the processing time, as perceived by the client.

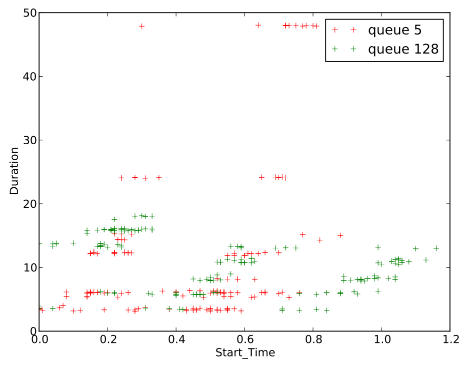
Figure 3: File access durations with a socket queue length of 5 vs a queue length of 128. (a) shows that the duration of a file create request is much shorter, largely contributed to a faster perceived executing time of the dpm_put as shown in (b).

retries for SYN packets are limited to 5 retries corresponding to a waiting time of about 180 seconds, while the waiting time increases with each retry. The value for the number of retries can be seen and adjusted in `/proc/sys/net/ipv4/tcp_syn_retries`.

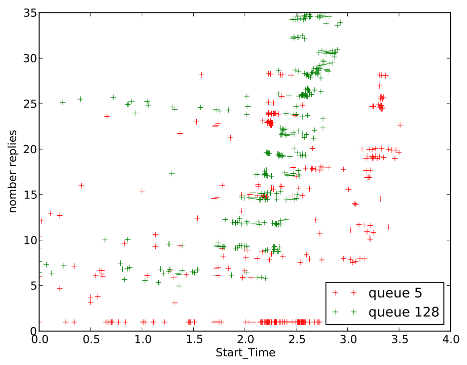
The DPM source shows that the listening socket is initialised with a queue length of five, which could explain this behaviour. As the connections cannot be accepted as fast as they appear, the queue fills up and further connection attempts are discarded. After a waiting period, the queue is free with a high probability and the connection is accepted.

In a test with a larger socket queue of 128 places, the maximum possible value for systems running Scientific Linux, we obtain the results as shown in Figure 3a. The duration of all interactions is lower with the larger socket queue and in particular, the duration of the `dpm_put` method as perceived by the client, has plummeted (see Figure 3b).

However, the DPM log files show that in the case of the large socket queue, the processing time, in the `dpm_proc_put` function has increased in average from 1.2 seconds to about 1.8 seconds, lessening the speedup gained. This can be accounted to the fact that now, as all requests can be accepted almost immediately, the server makes more use of its resources. The log files for the case with a short buffer queue show that although all threads are used, there are waiting periods on the server induced by the second-long waits on the client side. An analysis reveals an idle time during the test with a socket



(a) Comparison between socket queue lengths with 150 concurrent requests.



(b) Comparison between socket queue lengths with 300 concurrent requests.

Figure 4: These plots compare the large socket queue length to the old value for higher load, i.e., more simultaneous requests. The number of clients remains three, but the number of requests started per client increases to 50 in (a) and to 100 in (b).

queue of length five, while the larger socket queue reduces the idle time per thread, thereby reducing the overall time for all requests.

At this point we can say that the short queue length seems to act as a gatekeeper to the DPM head node. If requests cannot be handled immediately, there is almost no space to queue on the server and the client begins a minimal two seconds waiting period. On the one hand, this eases the load on the server as requests coming in bulks are stretched out and the DPM daemon is not overloaded.

On the other hand, this load limiting mechanism might be too strict on DPM. As we have seen with a larger buffer queue, DPM seems to handle many more simultaneous requests with only a slight increase in the duration of the computations than allowed in the standard configuration.

The next step in the analysis is to see whether the new solution also works faster on a higher loaded server. Figure 4 shows two plots with the test repeated with 150 and 300 concurrent requests; these are 50 and 100 requests per client respectively. Both graphs compare the durations between a queue length of 5 and 128.

The long durations for the requests to succeed indicates that the DPM server is under heavy load. While Figure 4a gives a similar picture as the test with 75 requests for the durations, Figure 4b is different: this might partly be due to the fact, that the requests are spread over several seconds, or, which we do not see in the graph, that 50 requests fail with the error message `Connection reset by peer` in

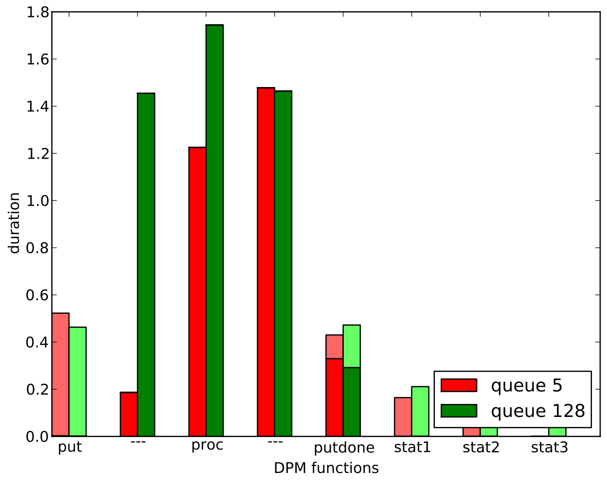


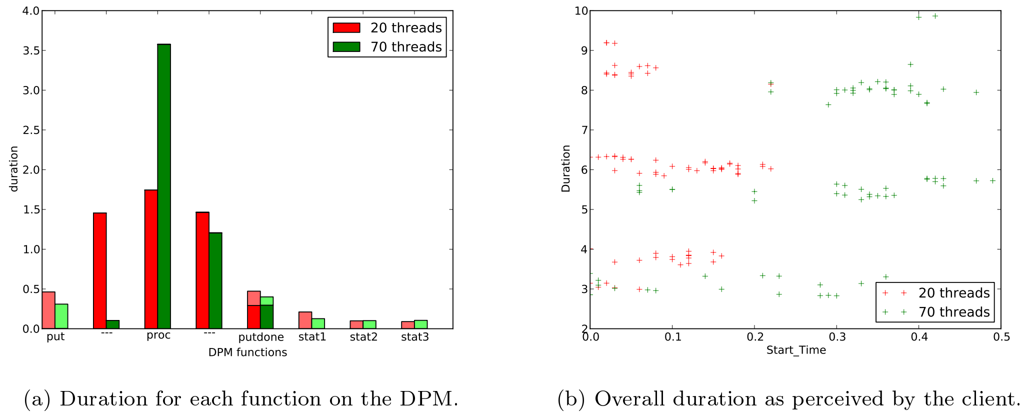
Figure 5: The durations of DPM function and the waiting times between them.

the test with the short buffer queue.

Thus we can say that the DPM configured with the larger socket queue is performing better than with a small one, either responding faster to requests or succeeding to process more during the same time frame. How bad a request failure is in a production environment is not straightforward, as application software will have their own retry mechanisms built-in, which might resubmit requests we have seen failing.

For our analysis, it is notable that the durations for a request rise as the DPM is exposed to a higher load. A profiling of the functions on DPM shows that the major hold-up for processing is a waiting period before the `dpm_proc_put` method. As seen in Figure 5, the waiting time before a request is assigned to a slow thread on DPM for processing the request, is much longer using a large socket queue. This can be explained given the information we extracted earlier that the DPM accepts more request in a time frame. As we only have 20 slow threads for 75 requests available, the requests have to queue internally for up to two durations that it takes DPM to process a request in the `dpm_proc_put` method.

Through this, we see the importance of the partitioning of fast and slow threads in combination with the short socket queue. Requests can be accepted very fast by the `dpm_put` function in a fast thread which can immediately serve the next request and thus the socket queue is growing relatively slowly. The DPM load is better represented by the number of requests queued in the DPM database waiting for a slow thread than by the socket queue size.



(a) Duration for each function on the DPM.

(b) Overall duration as perceived by the client.

Figure 6: DPM with 75 concurrent requests in two configurations: with 20 slow threads and 70. (a) shows the DPM view and (b) the client view for the duration.

In Figure 5 we also see that the `dpm_proc_put` takes longer in average for the large socket queue. This might be due to the better utilisation of the server's resources, inducing a higher computational load and more concurrent accesses to the DPNS.

4.2 Adjusting the Number of Threads

These results suggest that DPM would perform better with a higher number of slow threads, as more requests queued internally could be served at the same time. A comparison between 20 slow threads and a new configuration with 70 slow threads, almost reaching the maximum of 100 with 20+70 threads is shown in Figure 6. The Figures display the results for 75 concurrent requests to DPM. In Figure 6a we see that the waiting time for the `dpm_proc_put`, an available slow thread, has decreased. This is expected as 70 out of 75 requests are assigned a thread immediately. On the other hand, the processing time of the `dpm_proc_put` has increased and the overall speedup is minimal, as can be seen in Figure 6b which displays the client view.

The long duration for the `dpm_proc_put` methods can be explained by DPM's behaviour when creating files. The `dpm_proc_put` method contacts the DPNS to check whether the file can be created and to create the necessary entries in the database. For this, also the database entry of the parent directory must be updated as it contains a counter for the number of files in it. With a new file being created, this counter is updated. Also, during these requests, it must be assured that the directory is not deleted, so a lock is set on the directory entry.

Many concurrent file create requests in the same directory lead to the effect that the `creatx` call issued in `dpm_proc_put` to the DPNS creating the file takes several seconds to complete due to the lock on the directory. Changing this behaviour would imply changes to the DPNS and to the name server database, which why we do not follow that path any further.

Instead, inspecting the speedup of our changes to DPM in a case where this limitation does not apply, we can see that te duration of the `spm_proc_put` method is reduced from 3.5 seconds to under 1 second. A reduction in processing time is measurable both for a DPM configuration with 20 as for one with 70 slow threads. In this scenario, the increase of slow threads has very little effect on the overall duration, as the function takes much less time as if it had to wait for the directory database entry.

5 Conclusion

Overall, we can look at three possible optimisations. The first is setting the queue of the listening socket to a higher value and involves a change in the DPM source code. The second is adjusting the number of threads to a suitable value and is already common practise. The third method involves the client behaviour: if we see the DPNS part as a black box at the moment, the only way to avoid the directory lock when creating many files in one directory is to use several directories, if a large number of files is involved. We discuss all three methods shortly with some remarks about the performance gain from these measures.

In our test situation with bursty requests, increasing the queue length has proven to be beneficial to the overall duration of client-server interaction. We have seen that the short queue leads to connection retries on the client side, which render the interactions longer as necessary, given the resources of DPM. Here, extending the DPM socket queue length leads to a better utilisation of the server.

Of course it has to be noted that increasing the queue length does not help if DPM is under continuous heavy load. If that is the case, the functions processing the request itself must be faster, otherwise even the largest queue fills up at some time. Here action is undertaken by the developers creating a synchronous method for ‘getting’ and ‘putting’ files, which needs less internal communication. However, a larger queue length helps the server to cope with short bursts of requests.

The case for the socket queue is especially interesting in the context of a synchronous put method where DPM only uses one kind of threads which handle the whole request. Then, the internal queue would no longer exist and, in the case of all threads being in use, new requests would queue at the socket. A short value there could lead to more connection failures on a highly loaded DPM.

When increasing the number of threads I could see that for file put requests, the number of fast threads seems to play a minor role, while the number of slow threads should be higher. Bearing in mind the limitation of the total number of DPM threads it might make sense to use an asymmetric configuration of threads. Other workloads where the slow threads are not involved, should be tested thoroughly to see if these suffer from such a configuration. Also, the asymmetrical configuration only gives a significant speedup if the function computed in the slow threads takes sufficiently long. This is in the case for creating files in one directory, but not for different ones.

As for the third point, application developers working with DPM might wish to check if their programmes do create files this way and might be interested in testing them with multiple target directories. Relaxing the lock on the parent directory while creating a file can easily introduce inconsistencies and might involve changes to the DPNS database.

Bibliography

References

[1] “Perfsuite.” https://svn.cern.ch/reps/lcgdm/perfsuite . last visited on 19.8.2011.

[2] “CERN.” http://public.web.cern.ch/public/ . last visited on 27.2.2011.

[3] “ATLAS.” http://public.web.cern.ch/public/en/LHC/ATLAS-en.html . last visited on 27.2.2011.

[4] “WLCG Poster.” http://lcg.web.cern.ch/LCG/dissemination/flyers/Grid_english_2009.pdf , 2009. last visited on 27.2.2011.

[5] “DPM.” https://svnweb.cern.ch/trac/lcgdm/wiki/Dpm . last visited on 27.2.2011.

[6] “NAGIOS.” http://www.nagios.org/ . last visited on 27.2.2011.

[7] “Scientific Linux.” http://www.scientificlinux.org/ . last visited on 19.8.2011.

Appendix

Test Bed	machines	Processor	Cores	RAM / GByte	Network Link / Mbps
lxfsra	3	L5520 2.27 GHz	4	12	1000
lxplus	15	E5410 2.33 GHz	8	16	1000
	13	L5420 2.5 GHz	8	16	1000
	47	L5520 2.27 GHz	16	48	1000
dmstestvm	2	2.27 GHz	8	24	10000

Table 2: Hardware Description of the test beds used. Note that 10 virtual machines run on one server in the dmstestvm test bed and they have a 10 Gbps network link. The exact processor type of these machines is not known to me. All processors used are Intel Xeon x86.