ISTITUTO NAZIONALE DI FISICA NUCLEARE

Sezione di Bari

INFN/CCR-07/06
25 Maggio 2007

Commissione Calcolo Reti

CCR-09/2006/P
Dicembre 2006

dCACHE, STORM/GPFS AND DPM: PERFORMANCE TESTS, SRM COMPLIANCE, ADVANCED CONFIGURATIONS

Giacinto Donvito, Vincenzo Spinoso

INFN, Sezione di Bari, I-70100 Bari, Italy

Abstract

Grid storage managers represent a new way to deal with large sets of files in a Grid environment. They are growing quickly, meeting LHC experiment requirements; in particular, nowadays they must honour SC4 requirements as a subset of the whole complete set of features to be implemented for 2008.

The SRM interfaces are needed, both in version 1 and 2, and higher and higher transfer rates (i.e. in SC4, 100MB/sWAN/disk and 1GB/s inside LAN) can't be managed through a classic SE. Possible candidates as SRMstorage managers are dCache, DPM and StoRM.

This work aims to test installation, configuration, features and limits of those storage managers; the resulting overview will be useful both for Tier 2 sites and dCache/DPM developers, and will represent a basis on which to start a regular testing activity about grid storage management.

All the results reported in this note are updated to January 2006. All the software tested is now released in a newer version, with more functionalities added. This work is the result of a working activity born as a requirements of the INFN "Commissione Calcolo e Reti".

PACS:11.30.Er; 13.20.Eb; 13.20Jf; 29.40.Gx; 29.40.Vj

*Published by **SIS–Pubblicazioni***
Laboratori Nazionali di Frascati

1 Our experience on dCache performances at CNAF

The testbed described here has been deployed at CNAF as part of the activities of INFN Storage Group[5]. The target has been the analysis of dCache and GPFS performances and limits in a Tier2-like infrastructure; in particular we are interested in measuring:

- performances during concurrent access conditions, as concerns open, read and write operations;
- performances during opening and reading a high number of file descriptors; in particular, these tests have been inspired by CMS-experiment applications.

To achieve our measures, we have written two ad-hoc clients, called respectively *bench* and *bopener*.

1.1 *bench* and *bopener*

Both the clients are *multilibrary* (libdcap, GFAL, libshift) and *multiprotocol* (DCap, RFIO, SRM[7]).

Library	***Protocol***	***Access to***
system	file	any local/network fs
libdcap	DCap, GSIDCap	dCache
GFAL	SRM	dCache, DPM, CASTOR, DRM
libdpm	GSI-enabled RFIO	DPM
libshift	RFIO	CASTOR, classic SE

Table 1: Linked libraries (client side) and available protocols (server side)

We have written both clients in C, in order to:

- minimize the overhead of execution;
- maximize client-side performances;
- guarantee portability and small size of binaries;
- use already available C/C++ libraries.

Through **bench** you can write an arbitrary number of files, with arbitrary filesize; then, you can read a group of files, previously written, even in a randomized order. All

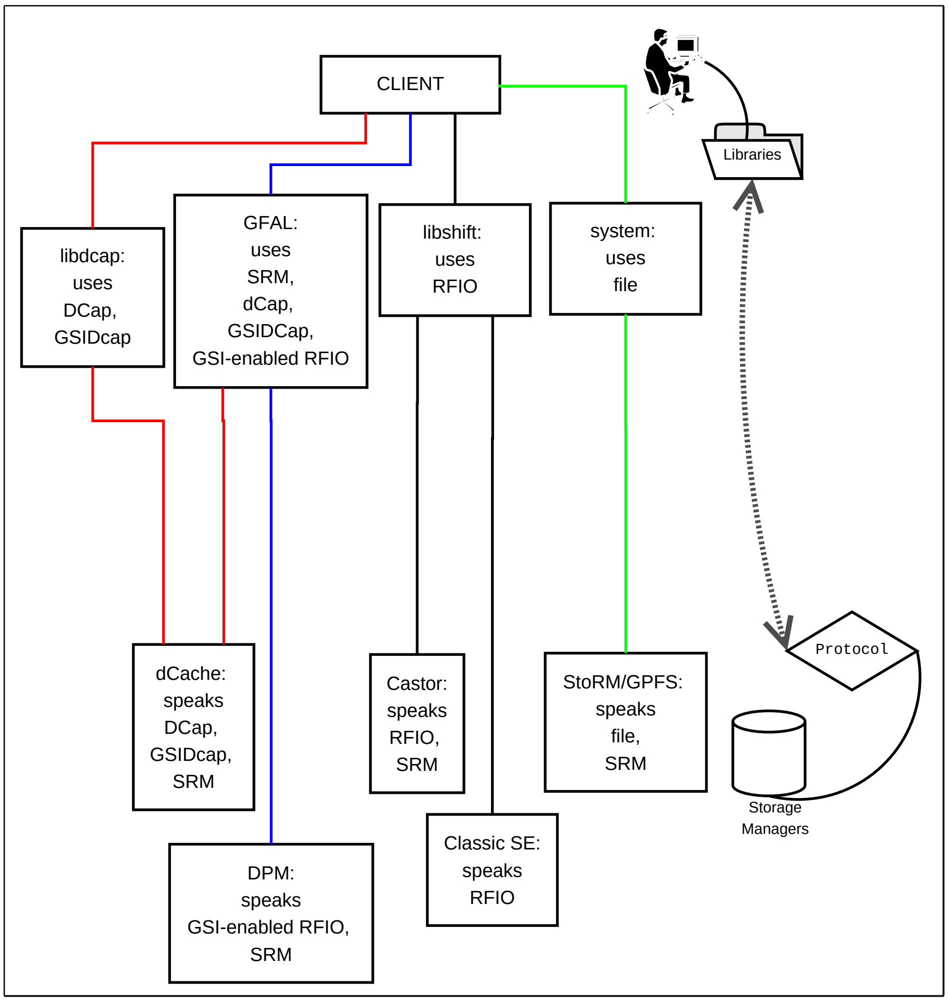
CLIENT
Libraries
libdcap: uses DCap, GSIDcap
GFAL: uses SRM, dCap, GSIDCap, GSI-enabled RFIO
libshift: uses RFIO
system: uses file
Protocol
dCache: speaks DCap, GSIDcap, SRM
Castor: speaks RFIO, SRM
StoRM/GPFS: speaks file, SRM
Storage Managers
DPM: speaks GSI-enabled RFIO, SRM
Classic SE: speaks RFIO

Figure 1: The client interfaces to may libraries, the server uses many protocols

open, read, write and close operations are logged on a MySQL server, so that you can process data off line afterwards. The content of each generated test file is random and alphanumeric.

The other client (**bopener**) is a stress test, so it behaves mostly like a *CMS analysis job*: each job opens about ten files; in a Tier2-like farm, with about a hundred worker nodes, we expect about a thousand files opened at the same time. Each file contains about a GB; **bopener** reads chunks of about one megabyte, in a random order (seek), until it reads the entire file. **bopener** never stop during reading, so its way of reading is much more I/O intensive than a CMS analysis job.

See figure 3 for details on how *bopener* works: for example, let's suppose you read 3 files, each of them subdivided into 4 chunks, and so 12 chunks to be read in a random

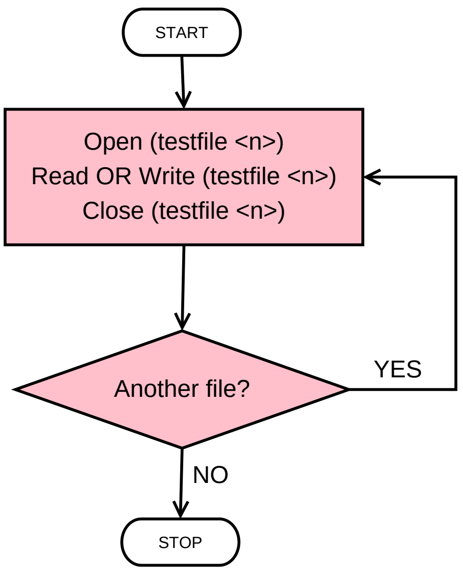

Figure 2: Executing *bench*

order. Our tests have opened about 200 files of 1 GB, subdivided into about a thousand of chunks. When you start it, *bopener*:

1. opens *all the test files*, and keeps them opened;
2. draws a random order of reading for chunks;
3. then, starts reading *all the chunks of all the files* in that order;
4. when it finishes reading, it closes the file descriptors.

Each client logs I/O operations on a MySQL server, in order to permit the reconstruction of the operations.

Each entry in the database contains, in particular:

- the library name (lib) and the protocol name (proto) used to perform the test;
- name (fullfilename) and other parameters of the file;
- type of operation (open, read, write, close);
- transfer parameters (transfer buffer, read-ahead buffer. . .);
- client hostname, endpoint (server) on which the file is allocated;

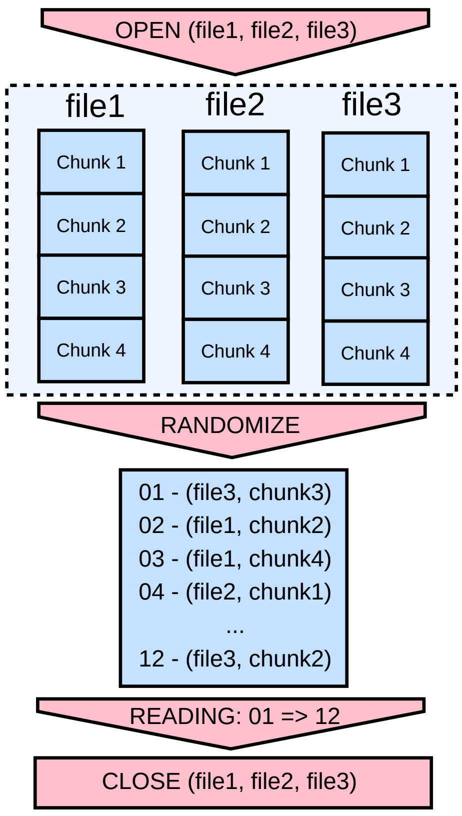

Figure 3: *bopener* on 3 files, 4 chunks/file

- the most important, *the time of execution of each single operation*.

It is possible to start many contemporaneous sessions of *bench* or *bopener* on many clients, in order to carry out an "aggressive access" on a storage manager.

When the test is finished, it is possible to read the log through Bash/Perl scripts, to generate interesting graphics.

1.2 Description of the testbed

The testbed installed at CNAF consists in:

- 4 GPFS[8] servers (installed and configured by CNAF administrators), which are used by StoRM developers to test StoRM[9];

- 5 servers, available for a large dCache installation.

The main features of the servers are reported in table 2. Besides we have used 34 client hosts to run 34 contemporaneous sessions of *bench* or *bopener*. Each client host is

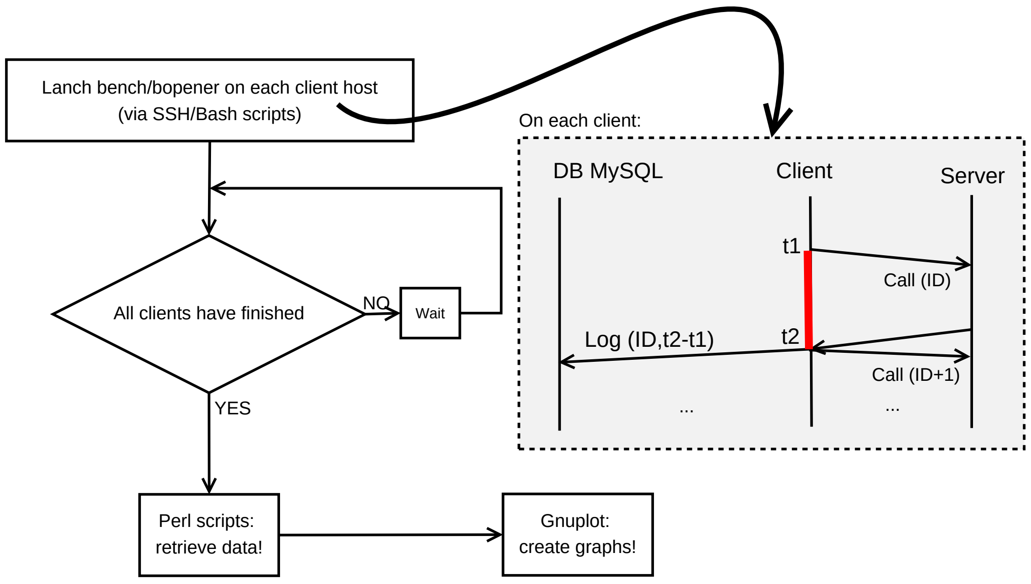

Figure 4: Performing a multi-client test, with log on DB (MySQL)

Hostname	***Role***	***CPU***	***RAM***	***Disk space***
diskserv-san-28	Admin node	4 X Intel Xeon 3200 MHz	4GB	0
diskserv-san-29	Pool node	2 X AMD Opteron 2500 MHz	4GB	10.8TB
diskserv-san-30	Pool node	2 X AMD Opteron 2500 MHz	4GB	10.8TB
diskserv-san-31	Pool node	2 X AMD Opteron 2500 MHz	4GB	10.8TB
diskserv-san-32	Pool node	2 X AMD Opteron 2500 MHz	4GB	12.6TB
diskserv-san-33	GPFS	2 X AMD Opteron 2500 MHz	4GB	10.8TB
diskserv-san-34	GPFS	2 X AMD Opteron 2500 MHz	4GB	10.8TB
diskserv-san-35	GPFS	2 X AMD Opteron 2500 MHz	4GB	10.8TB
diskserv-san-36	GPFS	2 X AMD Opteron 2500 MHz	4GB	10.8TB

Table 2: Hardware used in the testbed

used as grid worker node in production, and has got Opteron biprocessor architecture and 4GB RAM.

Figure 5 describes the connections between the servers and the *Storage Area Network* (SAN). The SAN consists in a StorageTek Flexline Raid-5 disk array, connected to a Fiber Channel (FC) switch. The SAN exports (via SCSI) 24 volumes of 1.8 TB and 2 volumes of 0.9 TB for 4 *dCache pools*, and 20 volumes of 1.8 TB on 4 GPFS servers; thus GPFS manages 36 TB of storage space, while dCache manages 45 TB.

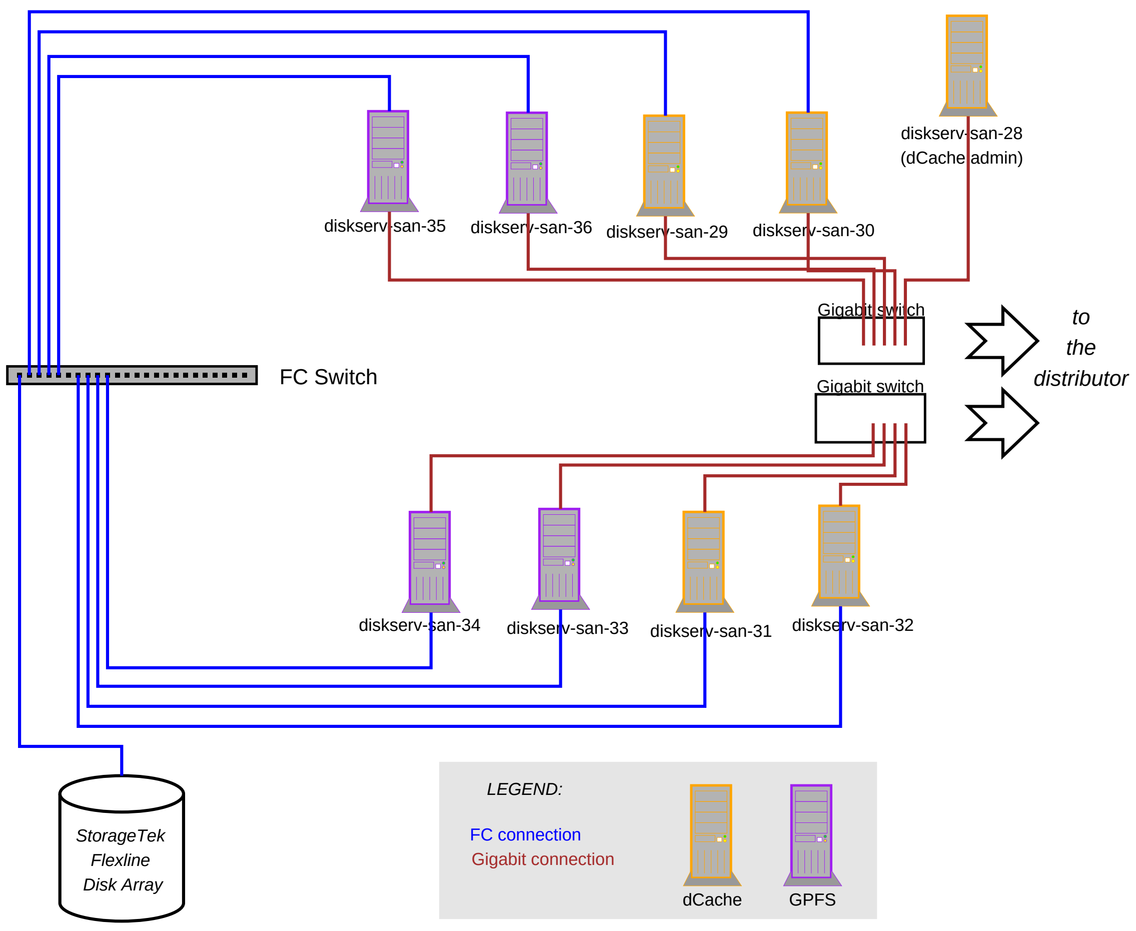

Figure 5: CNAF Testbed - Connection scheme

Figure 6 shows the connection between the switches and the WN container through the distributor. dCache and GPFS servers are connected to different switches; thanks to this topology, if you run tests on dCache or GPFS, but not at the same time, then you will enjoy a bandwidth of 4Gbit/s. In reality, one of the two uplinks is broken, so we can experience at most 3Gb/s, that's to say 375MB/s.

The 4 GPFS servers export the GPFS on all the worker nodes, so that you can enter the file system under */gpfs*. We have used it to copy and run *bench* and *bopener*.

1.3 Tuning and preparation

The proposed configuration requires all the control connections to be established with the admin node, and all the data connections with the 4 servers mounting the SAN via SCSI.

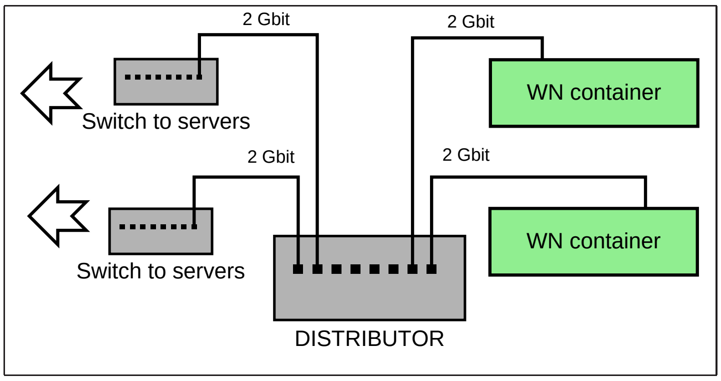

Figure 6: Testbed CNAF - Connection to the distributor

The operating system used on each machine is Scientific Linux CERN 3[6], using the Linux kernel 2.4.

The first step has been the tuning of the operating system, to be sure that the network interfaces and the disks were correctly configured to support our tests: in this phase, the program **iperf**[10] has been useful to test the quality of a sustained disk-to-disk data transfer.

Then we have installed MySQL 5.0 on a separate host in LAN, to register the logs of **bench** and **bopener**; the last step, of course, has been the standard installation of dCache (version 1.6.6).

1.4 Performance comparison between dCache and GPFS

The following tests consist in writing and reading files, running a single *bench* processor each WN, having 34 WN. Each WN writes and reads 20 files; so the storage manager must manage a total number of 680 files. Each test file has a size of 1 GB.

The first comparison concerns contemporaneous write operations from the 34 WN to dCache or GPFS. In this case, the “bottle necks” are respectively the dCache *admin node*, which has to schedule the write operations on the pools, and the shared access to the GPFS.

As you can see in figure 7, dCache wins: it opens the files at a flat rate (under 0.6[s/open] in open for write), while GPFS becomes slower and slower, until it exceeds more than 80 seconds.

dCache seems also to be faster in the “write” operation (fig. 8) with a flat rate of about 250MB/s; GPFS, on the contrary, starts writing at the same rate, but with decreasing performances.

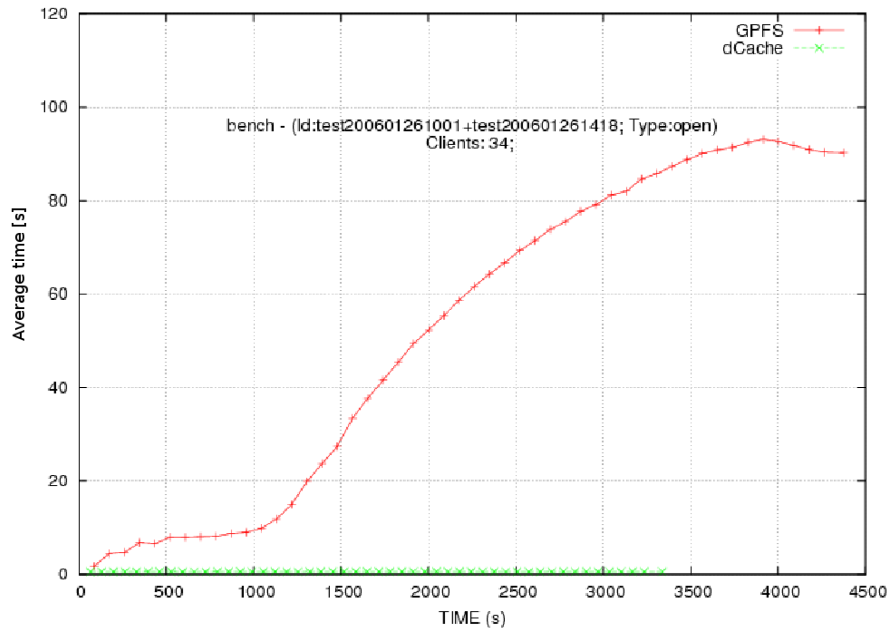

Figure 7: Writing - Average time to execute “open”

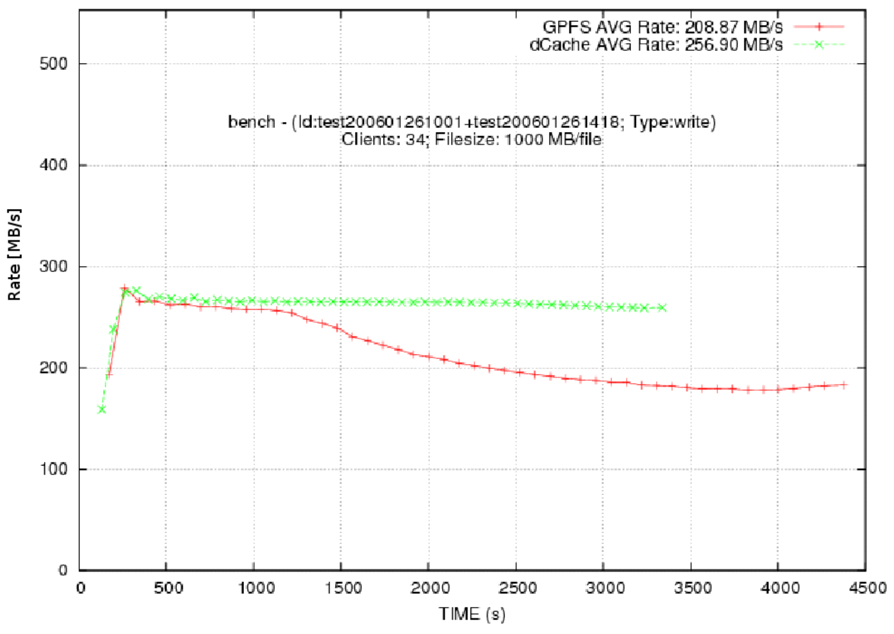

Figure 8: Writing - Average time to execute “write”

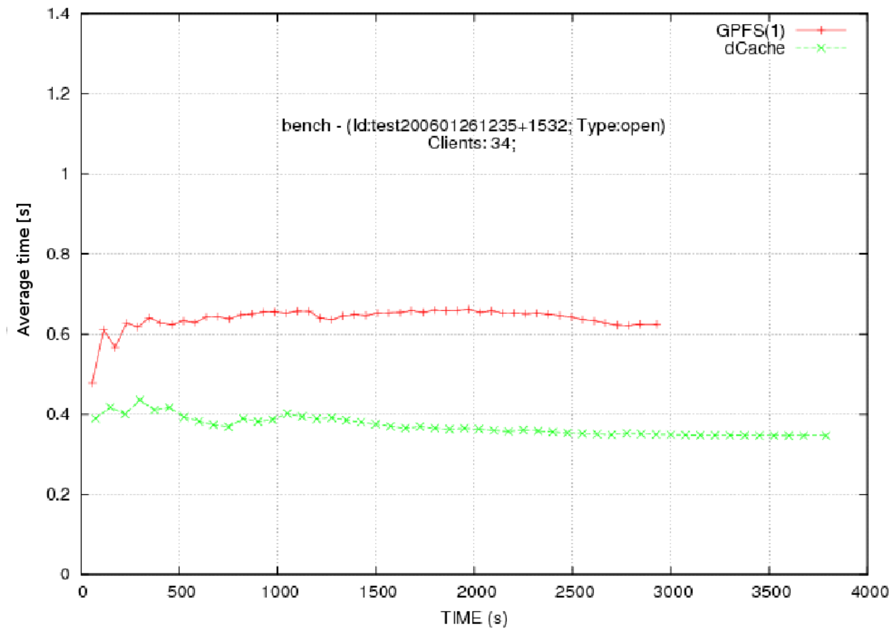

Figure 9: Reading - Average time to execute “open”

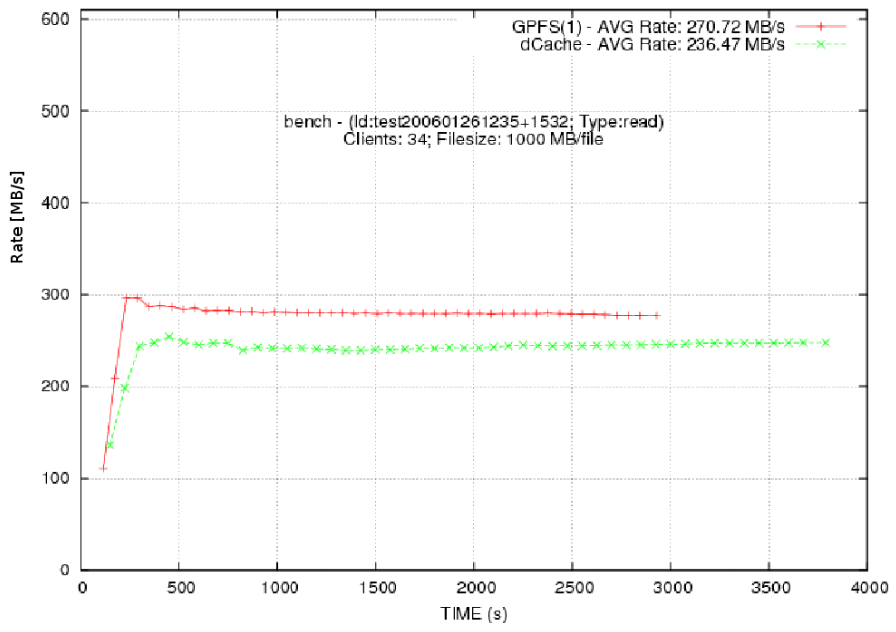

Figure 10: Reading - Average time to execute “write”

The read test shows about the same behaviour of dCache and GPFS: *a file is opened in less than a second* (fig. 9), and the transfer rate is flat 10 in both cases. GPFS is a little bit faster (270 MB/s against 235 MB/s of dCache).

1.5 "bopener" on dCache

The read test with *bopener* acts in this way:

- 34 processes are started at the same time on the 34 worker nodes;
- each process reads 60 files;
- each file has a size of 1 GB;
- each file is subdivided into chunks of 1 MB, so we have 1000 chunks/file.

The 34 processes:

- open a total number of $60 \times 34 = 2040$ descriptors on the server side;
- read all the chunks of all the files, for a total number of

  $$1000[chunks/file] \times 60[files/WN] \times 34[WN] = 2.040.000[chunks],$$

  and so a total amount of $2.040.000[chunks] \times 1[MB/chunk] \cong 2TB$;
- close, finally, all the files.

Let's notice that, during the test, the same file can be read simultaneously from two or more processes; in this way the test is closer to the real behaviour of CMS analysis jobs.

We have considered a set of $N = 720$ files, generated by *bench*. Let's say n is the number of files opened by a single process of *bopener* on a WN, and D is the total number of opened descriptors; the scheme used in the test is the following:

- the first WN opens files from number 1 to number n;
- the second WN opens files from number $1 + x$ to number $n + x$;
- the third WN opens files from number $1 + 2x$ to number $n + 2x$;
- and so on, until WN number 34, which opens from number $1 + 33x$ to number $n + 33x = N$.

It’s clear that, when you fix n (number of file to be opened from a single WN), you also fix x:

$$x = \frac{N - n}{33}$$

WN	***Read from file # . . .***	***. . . to file #***
1	1	60
2	21	80
3	41	100
4	61	120
5	81	140
. . .	. . .	. . .
30	581	640
31	601	660
32	621	680
33	641	700
34	661	720

Table 3: Example ($n = 60$) of scheduling read operations for a “bopener test”

It’s easy to compute D, that’s to say the number of opened descriptors:

$$D = 34n$$

and also *the average amount M of descriptors per file in shared reads*:

$$M = \frac{D}{N} = \frac{34n}{N}$$

When $n = 60$ ($N = 720$), we obtain the scheme in table 3 ($x = 20$, $D = 2040$).

Logging information in the DB has been helpful to control the correct end of each operation. The test has been repeated increasing the number n of files opened on each WN, from 60 to 120 ($2X$)), 147 ($2.5X$) and 180 ($3X$).

Let’s show the results in table 4 and figure 11: *when opening 180 files/WN the test has failed*. The number of “read” in the last test had to be 6 120 000, while we verify a clear inconsistency with the expected values.

As you can see in table 5:

- not all the open operations seem to be committed;
- not all open files have been completely read;

	n	***D***	***M***	***Read***	***Rate*** $[MB/s]$	***Durata***	***TB Tot***
$\surd$	60	2 040	2.9	2 040 000	214	3h 49m	2TB
$\surd$	120	4 080	5.8	4 080 000	238.5	5h 44m	4TB
$\surd$	147	4 998	7	4 998 000	247	6h 36m	5TB
$\times$	180	6 120	8.7	6 120 000	105	17h	6TB

Table 4: “bopener” on dCache (chunksize = 1MB, 1000 chunks/file)

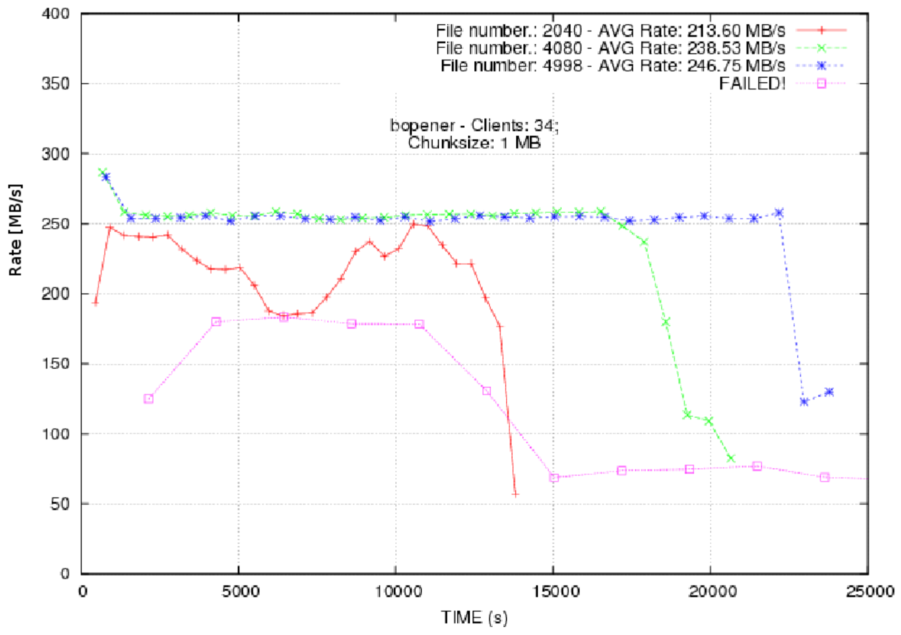

Figure 11: “bopener” on dCache

- not all open files have been closed.

The test, of course, has been re-executed several times.

	Expected	***Measured***	***Missed***
Open	6 120	5 835	285
Read	6 120 000	5 126 252	993748
Close	5 835	5 040	795

Table 5: Log inconsistency with expectation, bopener on 6120 files

It has not been possibile to catch any systematic error message to identify the problem; anyway, we received several *timeout messages* from the *libdcap* library on some *bopener* command line, while running on a WN.

We think that the failure is due to the small amount of *movers* on the dCache servers[1]. We tried to increase the maximum number of movers, but the behaviour didn't get better: the Java threads overloaded each server ($load > 10$).

1.6 The replica manager of dCache

The replica manager is a module which can be started *on demand*, and whose target is to duplicate files and monitor the copies in the cache. It wakes up with a given customizable frequency, so let's say 10 minutes (which is the default). Every time it wakes up, the replica manager counts the number of replicas for each file: if it is lower than the minimum required (let's say 2, which is the default), it schedules a new copy. In the same way, if it is higher than the maximum required (let's say 3, which is the default), it schedules a removal.

The advantages are many. The *read access* gets better: if two clients read the same file, then the admin node can forward the requests to different copies of the same file, decreasing the overhead on a single pool (fig. 12).

Other (more) obvious advantages are: high fault tolerance and high availability. If a pool experiences a "downtime" the replica manager orders, if needed, a new copy.

The tests shown here have the following targets:

- comparison between a dCache on 4 pool servers without replica manager, and dCache with replica manager using 34 WN as pool nodes along the 4 servers;
- comparison using dCache on 4 pool servers, with or without replica manager, just to verify if the replica manager introduces overhead.

1.6.1 Pools on worker nodes

As just said, we have configured *all the 34 WN as dCache pools*, along the 4 disk servers; each WN offers 40 GB of disk space. Doing so, we have added 34 WN * 40 GB/WN = 1.3 TB to the 45 TB on the 4 servers[2].

[1]A *mover* is a Java thread which is responsible, on a pool node, of the I/O on a single file. A maximum number of movers is assigned to each pool; over that limit, requests are forwarded to other pools, hoping they have free movers available.

[2]Let's underline this new way of using replica manager to recover unused disk space on grid worker nodes!

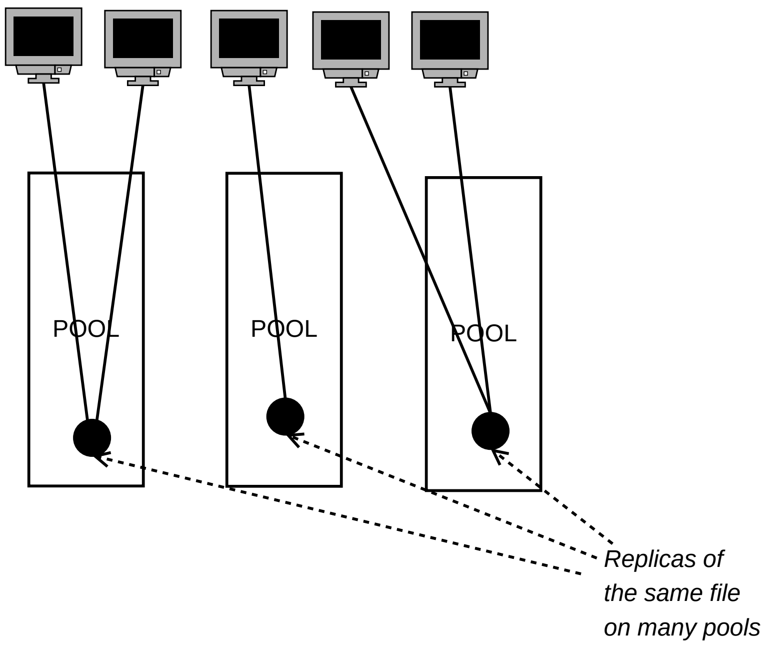

Figure 12: Multiple reads are forwarded to multiple replicas

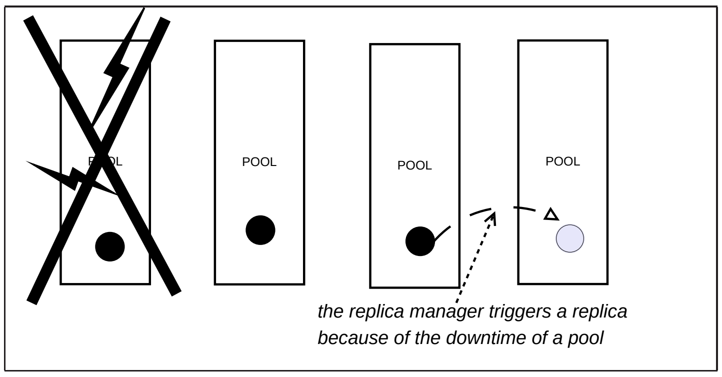
POOL
POOL
POOL
POOL
the replica manager triggers a replica
because of the downtime of a pool

Figure 13: Automatic replication of files

We have started the replica manager: we have observed a perfectly redistribution of the files on the new pools.

After having waited enough to be sure that any WN had registered itself on the admin node, and that the replica manager had triggered all needed replicas, we have started the test. Results are in fig. 14.

How can the rate be higher than “the possible one”? A suggestion follows:

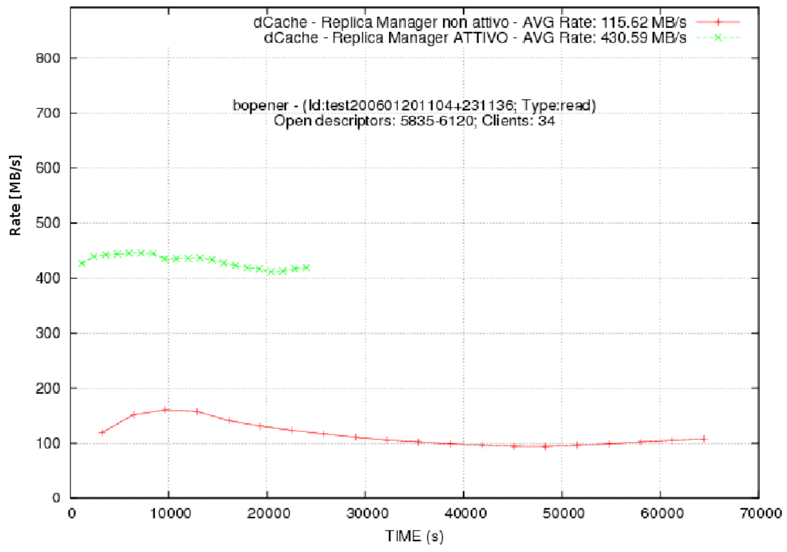

Figure 14: Read dCache files with or without *replica manager*

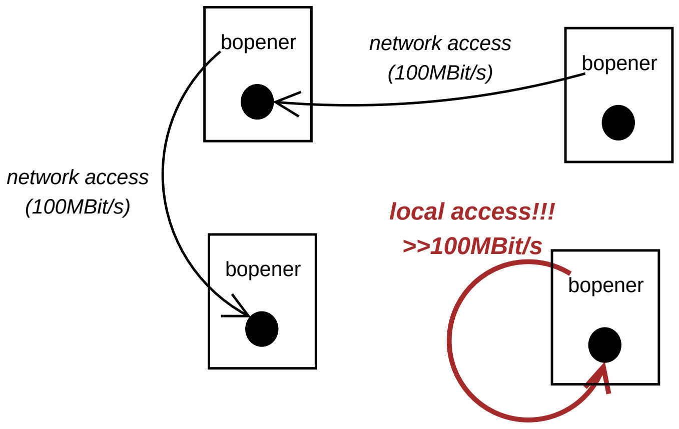

Figure 15: bopener reads replicas on the WN

1. a WN makes a request to the admin node to read a file;
2. the admin node chooses the best pool, among the pools containing that file (see 7.2 on "dCache, the Book"[3]);
3. the file is handled by the elected pool.

Due to the way in which the replica manager works, the replicas are never put on the same pool. If a WN, being a client, opens a file shared by itself (as pool node), then the cost value for that pool is the lowest! The biggest speed of that data transfer is NOT 100 Mbit/s (of the NIC), but much more, because it is a *local access*.

1.6.2 Write test

We want to measure the overhead produces by the activation of the replica manager on the 4 servers.

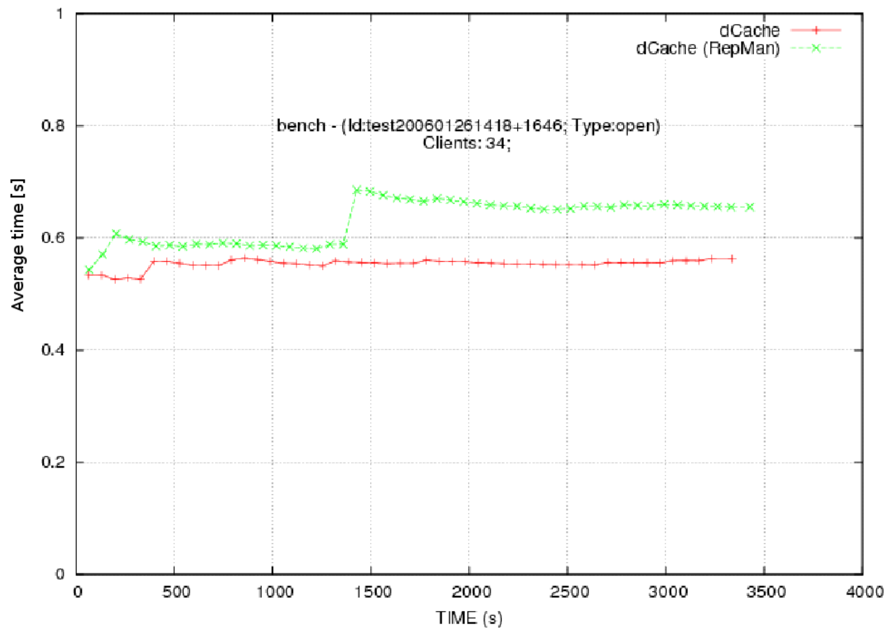

Figure 16: Replica manager - Write - Average time to open

So the result is that *activating the replica manager doesn't modify the average time to open files* (fig. 16); this time still remains *shorter than the GPFS average time to open files* (fig. 17). *Neither the average time to write has been modified* (fig. 18).

2 How to distribute administrative tasks on dCache

Let's give an overview of a dCache installation with a spreaded admin node. We are going to use five hosts, following the layout in table 6. The operating system installed on each server is Scientific Linux Cern 3.

The "classic" dCache *admin node* has been distributed on three different hosts, moving away the PostgreSQL server and the PNFS server. Disk space has been allocated on 3 *pools*, each one offering about 23 GB.

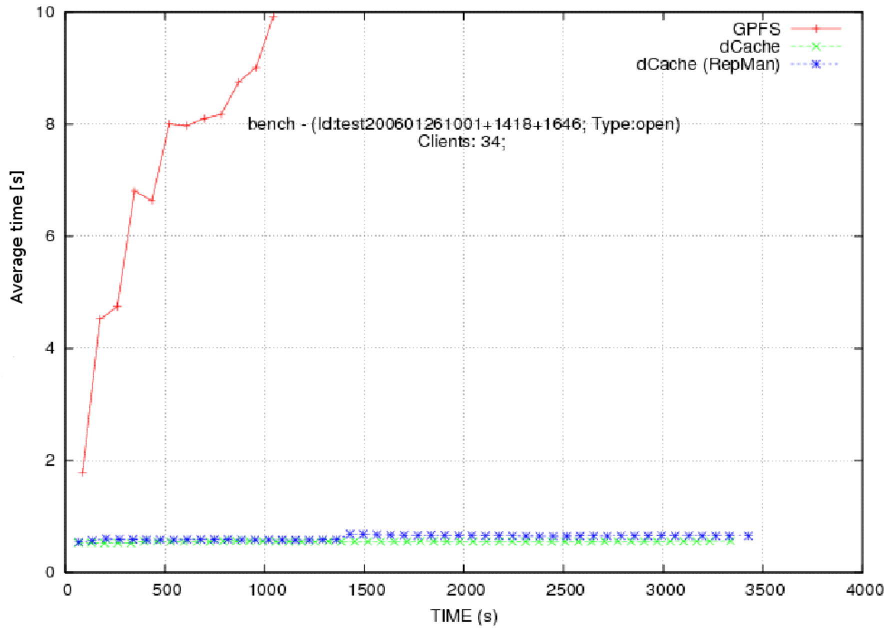

Figure 17: Replica manager - Average time to open on write (cmp. to GPFS)

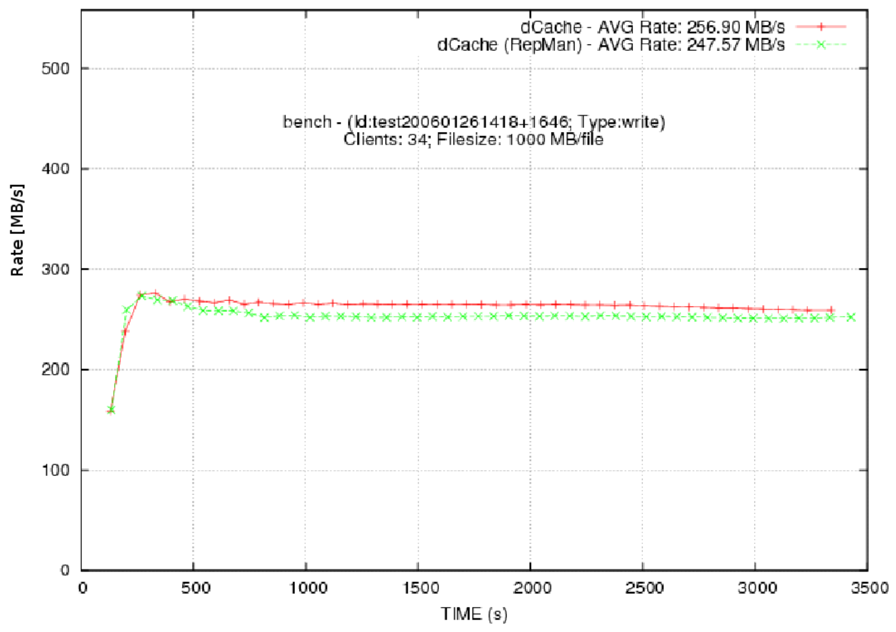

Figure 18: Replica manager - Write - Average time to “write”

Host	IP	Role	Cert.
vgridba4.ba.infn.it	192.167.40.6	Admin node + Pool	YES
vgridba6.ba.infn.it	192.167.40.8	PostgreSQL server	NO
gridtutorial5.ba.infn.it	192.167.40.105	PNFS Server	NO
vgridba3.ba.infn.it	192.167.40.5	Pool	YES
gridtutorial6.ba.infn.it	192.167.40.106	Pool	NO

Table 6: Layout of the dCache installation

Two of five servers have got host certificates, which allow authentication on typical grid protocols through *authenticated doors* (GridFTP, GsiDCap, SRM).

Let's give full details on how to deploy this kind of installation (see fig. 19).

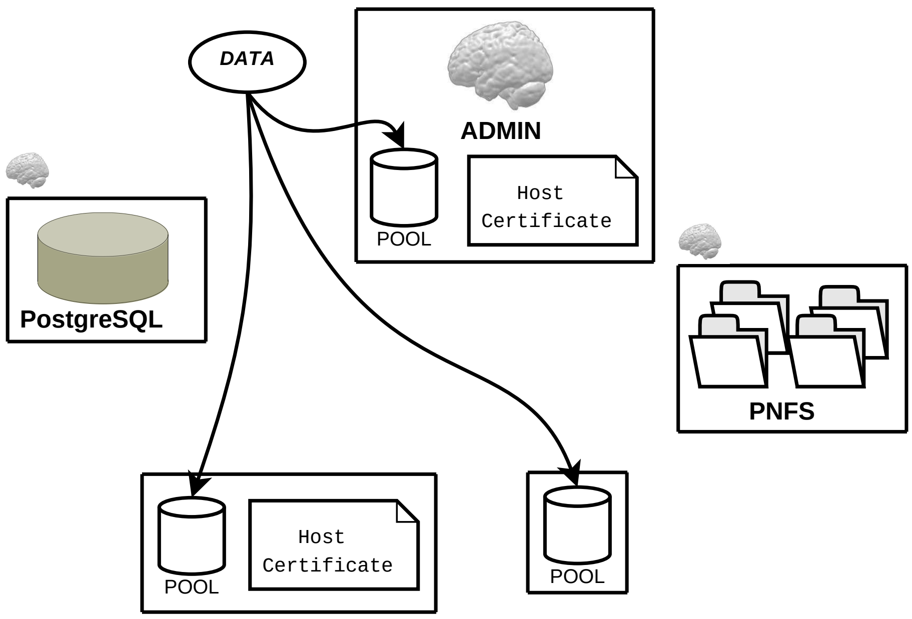

Figure 19: Distributed dCache admin node

2.1 Installation

First of all, we must download all needed RPMs. PostgreSQL can be downloaded from its official site[1]. Then, we can install RPMs on our PostgreSQL host:

```
# ls
postgresql-8.1.3-1PGDG.i686.rpm
postgresql-contrib-8.1.3-1PGDG.i686.rpm
postgresql-devel-8.1.3-1PGDG.i686.rpm
postgresql-docs-8.1.3-1PGDG.i686.rpm
postgresql-jdbc-8.1.3-1PGDG.i686.rpm
```

```
postgresql-libs-8.1.3-1PGDG.i686.rpm
postgresql-pl-8.1.3-1PGDG.i686.rpm
postgresql-python-8.1.3-1PGDG.i686.rpm
postgresql-server-8.1.3-1PGDG.i686.rpm
postgresql-tcl-8.1.3-1PGDG.i686.rpm
# rpm -ivh *
```

Now, let's download dCache 1.6.6-5 from it's official site[2]. It is distributed in a tarred "bundle", which contains the RPMs:

```
# ls
dcache-client-1.6.6-5.i386.rpm
dcache-server-1.6.6-5.i386.rpm
INSTALL
pnfs-gdbm-3.1.10-3.i386.rpm
pnfs-postgresql-3.1.10-3.i386.rpm
```

We have to install just the PNFS server on its host:

```
# rpm -ivh pnfs-postgresql-3.1.10-3.i386.rpm
```

All hosts must have installed both dCache client and server:

```
# rpm -ivh dcache-server-1.6.6-5.i386.rpm dcache-client-1.6.6-5.i386.rpm
```

After having installed dCache stuff on all the machines, let's start with PostgreSQL configuration. With the service stopped, modify `/var/lib/pgsql/data/postgresql.conf` so that it listens to client requests (`listen_addresses` must be `'*'`, not `'localhost'`):

```
listen_addresses = '*'
```

Besides, `/var/lib/pgsql/data/pg_hba.conf` must be altered:

```
# TYPE DATABASE USER CIDR-ADDRESS METHOD
local all all trust
host all all 127.0.0.1/32 trust
host all all ::1/128 trust
# PNFS server
host all all 192.167.40.105/32 trust
# Admin
host all all 192.167.40.6/32 trust
# SRM door - Cert Pool
host all all 192.167.40.5/32 trust
```

Doing so, you are allowing only needed hosts to connect to PostgreSQL. Then, let's follow the Book[3] in creating DB stuff on PostgreSQL:

```
# createuser -U postgres –no-superuser –no-createrole –createdb –pwprompt pnfsserver
# createuser -U postgres –no-superuser –no-createrole –createdb –pwprompt srmdcache
# createdb -U srmdcache dcache
# createdb -U srmdcache companion
# psql -U srmdcache companion -f /opt/d-cache/etc/psql_install_companion.sql
# createdb -U srmdcache replicas
# psql -U srmdcache replicas -f /opt/d-cache/etc/psql_install_replicas.sql
# createdb -U srmdcache billing
# /etc/init.d/postgresql start
```

We must tell the PNFS server that the PostgreSQL server is not on the local host. To do so, we must modify some files. First of all, let's open `/opt/pnfs/etc/pnfs_config` and add a new line:

```
PNFS_PSQL_HOST = vgridba6.ba.infn.it
```

This config file is generally called `$PNFS_CONFIG` inside the installation scripts. Then, modify `/opt/pnfs/tools/autoinstall-s.sh`, replacing this line:

```
export dbConnectString="user=$PNFS_PSQL_USER"
```

with this one:

```
export dbConnectString="host=$PNFS_PSQL_HOST user=$PNFS_PSQL_USER"
```

and adding these following lines to match the new variable `PNFS_PSQL_USER` in `$PNFS_CONFIG`:

```
if cat $PNFS_CONFIG — grep PNFS_PSQL_HOST ; then
PNFS_PSQL_HOST=‘cat $PNFS_CONFIG — grep PNFS_PSQL_HOST — awk '{print $3}'‘
else
PNFS_PSQL_HOST="localhost"
fi
```

Finally, run these commands:

```
# /opt/pnfs/install/pnfs-install.sh
# /opt/pnfs/bin/pnfs start
```

dCache is based on Java[4]. You must install a JVM everywhere but the PostgreSQL server:

```
# rpm -ivh j2sdk-1_4_2_08-linux-i586.rpm
```

PostgreSQL is ready, but we still have to work on the configuration of the PNFS server and the Admin node. The file `/opt/d-cache/config/dCacheSetup` has the same content for all the machines. Let's give a look just at the customized value:

```
serviceLocatorHost=vgridba4.ba.infn.it
java="/usr/java/j2sdk1.4.2_08/bin/java "
companionDatabaseHost=vgridba6.ba.infn.it
srmDatabaseHost=vgridba6.ba.infn.it
spaceManagerDatabaseHost=vgridba6.ba.infn.it
pinManagerDatabaseHost=vgridba6.ba.infn.it
replicaManagerDatabaseHost=vgridba6.ba.infn.it
billingDatabaseHost=vgridba6.ba.infn.it
defaultPnfsServer=gridtutorial5.ba.infn.it
billingToDb=yes
billingDatabaseHost=vgridba6.ba.infn.it
```

You can easily see that we have set the following common parameters:

- LocatorHost is the admin node;

- the JVM is located under our (eventually) customized path;
- the PostgreSQL server is different from the admin node: this is set by the "*Database-Host" variables;
- the PNFS server is a machine apart from the admin node and the PostgreSQL server.

We can leave defaults in `/opt/d-cache/etc/node_config`, except for some global parameters:

```
pnfsServer=gridtutorial5.ba.infn.it
ADMIN_NODE=vgridba4.ba.infn.it
```

and some other parameters which is very host-peculiar (see table 7).

	Admin	**PNFS server**	**Pool (no-cert)**	**Pool (cert)**
NODE_TYPE	admin	pool	pool	pool
GSIDCAP	**yes**	no	no	**yes**
GRIDFTP	**yes**	no	no	**yes**
SRM	**yes**	no	no	**yes**
replicaManager	**yes**	no	no	no
pnfsManager	no	**yes**	no	no

Table 7: Writing the `node_config` parameters

Another file to be configured on pools is `/opt/d-cache/etc/pool_path`; in our case, we configure it inserting just the following line, which adds 23 GB of free space under `/pool` to the disk cache:

```
/pool 23 no
```

Now we must set access control on to the pnfs filesystem via mount points. This can be done on the PNFS server, just following the Book ([3, § 4$\dot{7}$]):

```
# cd /pnfs/fs/admin/etc/exports
# echo "/pnfs /0/root/fs/usr 0 nooptions " > 0.0.0.0..0.0.0.0
# echo "/pnfsdoors /0/root/fs/usr/ 0 nooptions" >> 0.0.0.0..0.0.0.0
# echo "/fs /0/root/fs 0 nooptions" >> 0.0.0.0..0.0.0.0
```

We have to install some LCG packages useful for authentication purposes both on the admin node and on all those machines which must share an authenticated door. In our case, we do it on the admin node and on the pool with an host certificate:

```
# apt-get install ig-yaim
# /opt/lcg/yaim/scripts/ig_install_node  ig-site-info_new.def SE_dcache
# /opt/lcg/yaim/scripts/ig_configure_node  ig-site-info_new.def ig_SE_dcache
```

The setup is complete. Now, what is the correct startup order for dCache services? We follow these simple guidelines:

1. PostgreSQL (`/etc/init.d/postgresql start`), because several entities in dCache use it both for data and administrative purposes;
2. PNFS server (`/opt/pnfs/bin/pnfs start`) on its host;
3. PNFS domain (`/opt/d-cache/bin/dcache-core start`), still on the PNFS server;
4. admin services (`/opt/d-cache/bin/dcache-core start`) on the admin node;
5. pnfsdoors mounting on all machines which serve GridFTP, GsiDCAP or SRM, with the command:

   ```
   # mount -o intr,rw,noac,hard,nfsvers=2  gridtutorial5.ba.infn.it:/pnfsdoors /pnfs/ba.infn.it
   ```

   followed by starting the related door services:

   ```
   # /opt/d-cache/bin/dcache-core start
   ```

6. pool services (`/opt/d-cache/bin/dcache-pool start`) on all machines which give disk space.

You can access the dCache through `libdcap`, using the POSIX namespace "/pnfs...", just setting the environment variable `DCACHE_DOOR` and mounting PNFS:

```
# export DCACHE_DOOR=vgridba4.ba.infn.it:22125
# mount -o intr,rw,noac,hard,nfsvers=2  gridtutorial5.ba.infn.it:/fs /pnfs/fs
```

2.2 Different DBs for different VOs

By default, the PNFS server uses a single process and a single DB to manage all namespace access to the dCache.

We are going to explain a procedure to create different databases associated to different VOs and different processes. The advantage of this configuration is that the requests to the PNFS server load different processes, each one managing its own queue. Most of these techniques are suggested in the Book ([3, § 4.8]).

You can show databases managed by the PNFS server on its host, with:

```
# /opt/pnfs/tools/mdb show
```

By default, you will see two DBs:

```
ID Name Type Status Path
-------------------------------
0 admin r enabled (r) /opt/pnfsdb/pnfs/databases/admin
1 data1 r enabled (r) /opt/pnfsdb/pnfs/databases/data1
```

Let's say you want to add a new DB for CMS. Prepare the environment:

```
# . /usr/etc/pnfsSetup
# export PATH=$pnfs/tools:$PATH
```

Then create a new database called "cms"[3], and inform (update) PNFS about the changes.

```
# mdb create cms /opt/pnfsdb/pnfs/databases/cms
# mdb update
```

We can use `mdb show` to see if a new DB has been added. A new line must appear:

```
ID Name Type Status Path
-----------------------------------
... ... ... ... ...
2 cms r enabled (r) /opt/pnfsdb/pnfs/databases/cms
```

Every database keeps its own ID, so the ID for the cms database is "2". Now enter `/pnfs/fs/usr/data`, we must set some parameter in PNFS by hand.

```
# cd /pnfs/fs/usr/data
# mkdir .(2)(cms)
# cd cms
# echo "StoreName myStore" > '.(tag)(OSMTemplate)'
# echo "STRING" > '.(tag)(sGroup)'
```

Open `/pnfs/fs/admin/etc/.(id)(config)` and read it's content. Let's say it contains "000000000000000000001F80". Open also `/usr/etc/pnfsSetup` and read the value of `shmkey`; let's say it's "1122". Remembering that the DB's ID is "2", you must run the command:

```
# /opt/pnfs/tools/sclient getroot 1122 2 000000000000000000001F80
```

It's better to create an automatic procedure to create VOs. Write a text file containing the names of the VO you want to create:

```
cms
alice
atlas
lhcb
dteam
```

and save it with the name, let's say, "`vo-list`".

The following script, also available on the Storage Group Website[5], reads that file and creates the corresponding VOs:

```
#-----------------create-VO.sh--------------------
#!/bin/bash
VO_LIST=vo-list # VO list
PNFS_MOUNT=/pnfs # PNFS mountpoint
PNFS_DIR=/opt/pnfs # PNFS admin stuff
$PNFS_DIR/tools/mdb show
let "COUNT = 1"
```

[3]Don't use "tmp" as VO name, or you'll get into trouble, because a directory called `.()(tmp)` is already used by dCache for administrative operations.

```
for VO_NAME in ‘cat $VO_LIST‘ ; do
$PNFS_DIR/tools/mdb create $VO_NAME /opt/pnfsdb/pnfs/databases/$VO_NAME &&
touch /opt/pnfsdb/pnfs/databases/$VO_NAME &&
$PNFS_DIR/tools/mdb update && $PNFS_DIR/tools/mdb show &&
cd $PNFS_MOUNT/fs/usr/data/ &&
DBID=‘$PNFS_DIR/tools/mdb show — grep $VO_NAME — awk ’print $1’‘ &&
mkdir ”.($DBID)($VO_NAME)” &&
cd $VO_NAME &&
echo ”StoreName myStore” > ’.(tag)(OSMTemplate)’ &&
echo ”STRING” > ’.(tag)(sGroup)’ &&
cat ’.(tags)(all)’ &&
IDCONFIG=‘cat ’/pnfs/fs/admin/etc/.(id)(config)’‘ &&
SHMKEY=‘cat /usr/etc/pnfsSetup — grep shmkey — awk -F = ’{print $2}’‘ &&
$PNFS_DIR/tools/sclient getroot $SHMKEY $DBID $IDCONFIG
echo ”Done ($VO_NAME)”
let ”a+=1” ; done &&
$PNFS_DIR/tools/mdb show
#——————————create-VO.sh———————————-
```

2.3 Read and write pools

Everytime dCache has to write some file, it chooses a file using a “cost function”, as explained in the Book.

A pool can serve 3 kinds of operations: read, write, cache:

- a *write* pool can store data from the outside;
- a *read* or *cache* pool can provide data to the outside; in particular, a *cache* pool loads data from a tape library to its disk space to serve read requests, while a *read* pool serves precious space itself.

So it is interesting, in our use case, to examine *write* and *read* pools.

Per default, all pools are both read and write, but you can change this behaviour; so let’s get an example of a pratical configuration.

Let’s say we’ve got 3 available pools: vgridba3_1, vgridba4_1, gridtutorial6_1; they are located on three different host pools: vgridba3, vgridba4, gridtutorial6. Among these, vgridba4 e vgridba3 come with host certificates.

We should create two groups into which to include, respectively, read and write pools. Let’s call them, for example, “readgroup” and “writegroup”. We’re going to configure vgridba3 and gridtutorial6 as read pools, and vgridba4 as a write pool, so that all write operations will load the only write pool (vgridba4), and read operations will be available only on the others.

Log into dCache admin node, using the SSH administration interface (or, if you prefer, the dCache GUI[3]:

```
ssh -c blowfish -p 22223 -l admin vgridba4.ba.infn.it
```

Now let’s enter the PoolManager-cell and create the groups:

```
cd PoolManager-cell
psu create pgroup readgroup
psu create pgroup writegroup
```

and define the right associations (adding to the new groups and removing from the default group):

```
psu addto pgroup readgroup vgridba3_1
psu addto pgroup readgroup gridtutorial6_1
psu addto pgroup writegroup vgridba4_1
psu removefrom pgroup default vgridba3_1
psu removefrom pgroup default vgridba4_1
psu removefrom pgroup default gridtutorial6_1
```

The following commands are suggested by the dCache Book[3] as the simplest configuration you can do:

```
psu create unit -net 0.0.0.0/0.0.0.0
psu create ugroup allnet-cond
psu addto ugroup allnet-cond 0.0.0.0/0.0.0.0
psu create link read-link allnet-cond
psu set link read-link -readpref=10 -writepref=0 -cachepref=10
psu add link read-link readgroup
psu create link write-link allnet-cond
psu set link write-link -readpref=0 -writepref=10 -cachepref=0
psu add link write-link writegroup
save
reload -yes
```

Note “save and reload” at the end of the previous commands: they allow to make dCache aware of modifications. Refer to chapter 7 of the dCache Book[3] for details on the following commands.

If you try to use this configuration, you will see that:

- if you write a file on dCache, then the file will be always written on the write pool; so all the files will be stored on that pool, which provides the so called *precious space*;
- if you read a file from dCache, and that file is not already stored on any read pools, then dCache will trigger a copy of that file to one of the two read pools, choosing it on a cost-basis;
- if you try to read more than one file, you will see dCache load balancing on the two read pools.

Anyway, if a copy of the requested file is already on one of the two read pools, then it is unconditionally choosen; so if you read a bunch of ten files, for example, already stored on the first read pool, then it will be overloaded even if the second one is free.

3 Tests on SRM interfaces

Nowadays, every grid storage element should provide all SRM 1.1 features, together a subset of SRM 2.1 features required by LHC experiments, paying particular attention to CMS requirements.

Storage managers put through tests are dCache, DPM[11] and StoRM[9] (see table 3).

SRM server	***Site***	***Hostname***
dCache 1.6.5	Bari	pccms2.cmsfarm1.ba.infn.it
DPM 1.5.4	Bari	pccms5.cmsfarm1.ba.infn.it
dCache 1.6.6	CNAF	diskserv-san-28.cnaf.infn.it
StoRM 1.0.3	CNAF	storm01.cr.cnaf.infn.it

Table 8: Installations used in SRM tests

In particular:

1. it seems to be no differences between dCache 1.6.5 and 1.6.6; both of them fully support SRM 1 and have no support for SRM 2;
2. DPM 1.5.4 has been released officially with full support to SRM 1 and almost all features belonging to SRM 2;
3. StoRM 1.0.3 has been design to provide only SRM 2 features, so no SRM 1 support is given.

Thus dCache-StoRM interoperability will be impossible, until dCache releases SRM 2.

3.1 Available clients

Above all, the dCache client provides *srmcp*.

DPM provides two *test suites* inside DPM CVS, written by Jiri Kosina (for SRM 1) and Gilbert Grosdidier (SRM 2). We will show how to use the DPM test suites in a semi-automatic way, organizing commands in a Bash script .

The StoRM client has been released with StoRM just for test purposes; when we performed the tests we found it fairly incomplete.

The table 3.1 shows *all SRM features that can be tested with some client.* The last column ("Priority") shows how much each feature is wanted by LHC experiments.

We used the symbol "$\surd$ " to mean "used", else we used "$\times$ ".

Feature/Client	**SRM**	***dCache***	***DPM***	***StoRM***	**Priority**
Get	v1	✓	×	×	High
Put	v1	✓	×	×	High
Copy	v1	✓	×	×	High
getFileMetaData	v1	✓	✓	×	High
getRequestStatus	v1	✓	×	×	High
getProtocols	v1	×	✓	×	Norm
AdvisoryDelete	v1	✓	✓	×	High
Copy	v2	×	×	✓	High
Ls	v2	×	✓	✓	High
Mkdir	v2	×	✓	✓	High
Rmdir	v2	×	✓	×	High
Rm	v2	×	✓	×	High
Mv	v2	×	✓	×	High
Copy	v2	×	×	✓	High
GetRequestSummary	v2	×	✓	✓	High
GetRequestID	v2	×	✓	✓	High
PrepareToGet	v2	×	✓	✓	High
PrepareToPut	v2	×	✓	✓	High
RemoveFiles	v2	×	✓	×	High
ReleaseFiles	v2	×	✓	×	High
Putdone	v2	×	✓	✓	High
ReserveSpace	v2	×	✓	✓	High
UpdateSpace	v2	×	✓	×	High
ReleaseSpace	v2	×	✓	×	High
AbortRequest	v2	×	✓	×	High
SuspendRequest	v2	×	✓	×	High
ResumeRequest	v2	×	✓	×	High
ping	N/A	×	✓	×	?

Table 9: SRM client: provided features

3.2 SRM tests

Table 3.2 describes SRM test results on dCache, DPM and StoRM, as concerns both SRM versions:

- the symbol "✓" stands for "good result and draft compliance";
- the symbol "×" stands for "bad result", or "not supported", or no draft compliance;

- elsewhere we use “ *?* ”, meaning that this tests have to be done in the future.

  Section 3.3 and 3.4 shows further details.

Feature/Server	SRM	Client	*dCache*	*DPM*	*StoRM*	Priority
Get	v1	dCache	✓	✓	×	High
Put	v1	dCache	✓	✓	×	High
Copy	v1	dCache	✓	×	×	High
getFileMetaData	v1	dCache	✓	✓	×	High
getRequestStatus	v1	dCache	✓	✓	×	High
getProtocols	v1	DPM	×	✓	×	Norm
AdvisoryDelete	v1	DPM	✓	✓	×	High
ping	N/A	DPM	×	✓	?	?
PrepareToGet	v2	DPM-StoRM	×	×	×	High
ReleaseFiles	v2	DPM	×	?	?	High
RemoveFiles	v2	DPM	×	×	?	High
PrepareToPut	v2	DPM-StoRM	×	×	✓	High
PutDone	v2	DPM	×	×	?	High
Copy	v2	DPM-StoRM	×	×	×	High
GetRequestSummary	v2	DPM	×	✓	?	High
GetRequestID	v2	DPM	×	✓	?	High
AbortRequest	v2	DPM	×	✓	?	High
SuspendRequest	v2	DPM	×	×	?	High
ResumeRequest	v2	DPM	×	×	?	High
Ls	v2	DPM	×	×	?	High
Mkdir	v2	DPM	×	✓	?	High
Rmdir	v2	DPM	×	✓	?	High
Rm	v2	DPM	×	✓	?	High
Mv	v2	DPM	×	✓	?	High
ReserveSpace	v2	DPM	×	×	×	High
UpdateSpace	v2	DPM	×	×	?	High
ReleaseSpace	v2	DPM	×	×	?	High
SetPermission	v2	DPM	×	✓	?	High
CheckPermission	v2	DPM	×	✓	?	High

Table 10: SRM test results

3.3 SRM 1 tests

This sections show the tests on dCache and DPM. Both of them provide SRM 1; StoRM doesn’t, so it’s not here.

Anyway, SRM 1 functions already implemented seem to be more than enough to experiment needs.

All tests are made through the dCache client, except for `getProtocols` (for which the DPM test suite is useful).

As you can see in table 3.2 (page 29):

- both dCache and DPM provide *Get, Put, getFileMetaData, getRequestStatus, AdvisoryDelete*;
- dCache doesn't provide *getProtocols*, while DPM does; this is not a problem, because supported protocols can be queried to the *information system*;
- DPM is still lacking in "Copy", so it cannot handle *third-party transfers* yet.
- the *Ping* method is not mentioned in any SRM draft, but it's officially supported by (and only by) DPM; it's useful to check if SRM 1 daemon is alive.

3.4 SRM 2 tests

SRM 2 tests have been made on DPM e StoRM, while dCache officially doesn't provide SRM 2.

Referring to 2.1 [7], SRM functions can be classified as follows:

Data Transfer Functions Generally speaking, these functions don't actually move data, but *prepare* the SRM for data access. The only exception is "Copy", which really moves data from SRM to SRM. In SRM 2, each function returns an error/status message for each managed file.

Drafts classify under this particular class also some functions belonging to *status retrieval*, abort, suspend and resume.

Space Management In SRM 1.1, space reservation is made on a file basis; this means that a transfer request belonging to more than one file could fail, because the user/client cannot pre-allocate space for *all files* in the request.

In SRM 2.1, a system administrator is able to reserve space *a priori*, with a specific *lifetime*. The user request returns a *token*, to be send afterwards together with PrepareToGet/PrepareToPut. If disk space runs out, then all subsequent requests fail with the message "No user space".

PrepareToPut performs an (*implicit space reservation*. SRM 2 provides also *global space reservation*: you allocate a so called *container*, identified by a token; afterwards, you will decide how many and which files to store. Global space reservation has its own methods: (ReserveSpace, ReleaseSpace, UpdateSpace, CompactSpace, GetSpaceMetadata, ChangeFileStorageType, GetSpaceToken).

Directory Functions Posix-style functions to create/remove directories, delete files, rename files and directories.

Permission Functions SRM 2.1 provides permissions on files and directories with Access Control Lists.

3.4.1 Data management functions

PrepareToGet prepares a file for transfer; in particular, it is sent to an SRM to pin (*implicit pinning*) a file until transfer ends up. The file transfer must be made subsequently. After the file transfer, you must call ReleaseFiles or RemoveFiles.

× ***DPM*** PrepareToGet *doesn't pin the file*. Let's use, for example, RFIO:

```
./srm2_testGet rfio cheneso \
srm://pccms5.cmsfarm1.ba.infn.it:8444/dpm/cmsfarm1.ba.infn.it/home/cms/bigfile
```

the server returns both the TURL and the token to be used on the subsequent request:

```
soap_call_ns1__srmPrepareToGet returned r_token 2709cdd2-28a0-4296-9006-40e4f8a82873
TURL = rfio://pccms5.cmsfarm1.ba.infn.it//storage2/cms/2006-01-28/bigfile.72.0
```

Using GridFTP, the server replies correctly:

```
./srm2_testGet gsiftp cheneso \
srm://pccms5.cmsfarm1.ba.infn.it:8444/dpm/cmsfarm1.ba.infn.it/home/cms/bigfile
soap_call_ns1__srmPrepareToGet returned r_token 389aea10-9c07-4937-bd8e-7fe0e6245ac4
TURL =
gsiftp://pccms5.cmsfarm1.ba.infn.it/pccms5.cmsfarm1.ba.infn.it:/storage2/[...]/bigfile.72.0
```

A *globus-url-copy* works fine:

```
globus-url-copy \
gsiftp://pccms5.cmsfarm1.ba.infn.it/pccms5.cmsfarm1.ba.infn.it:/storage2/[...]/bigfile.72.0 \
file:///home/enzo/globbo
```

But if you try to remove the file *while a transfer is running*, then you will succeed. In particular, *it is possibile to remove a pinned file through srm2_testRm*, which is an SRM 2 client. We think it should'n happen.

× ***StoRM*** So does StoRM, pinning doesn't work. `srmRm` was not available on StoRM, so we used a GridFTP client (*edg-gridftp-rm*) to do the removal.

As StoRM developers suggested, the pin concept exists only inside StoRM application: for example, the garbage collector inside StoRM understands when a file is pinned, and doesn't remove it until lifetime expires. On the contrary, pinning cannot be exported "outside StoRM", because it should be supported at file system level (GPFS/Lustre) as a lock function.

ReleaseFiles releases (*unpin*) pinned files.

? ***DPM*** We can't test it, because we cannot pin files (see 3.4.1).

RemoveFiles releases *(unpin)* and removes previously pinned files.

× ***DPM*** RemoveFiles is not supported by DPM 1.5.4[4]

PrepareToPut It performs implicit space reservation on a file or a group of files.

The client calls PrepareToPut, sending a *user token*, the *lifetime* (in seconds), the *type* (volatile, durable, permanent) and, most important, the *filesize*.

PrepareToPut returns a request token and the TURL assigned to the file.

At the end of the transfer, the client calls *PutDone* to tell the server about completion.

× ***DPM*** Files can be removed while transferring, just like happened in PrepareToGet. SRM standards don't want it:

> "srmPutDone() is expected after each file is put into the allocated space. The lifetime of the file starts as soon as SRM get the srmPutDone(). If srmPutDone() is not provided then the files in that space are subject to removal when the space lifetime expires."[13]

PutDone releases (*unpin*) files after PrepareToPut.

? ***DPM*** We cannot test it until pinning works.

Another fact is that *DPM doesn't perform any control about filesize, so it can exceed the size declared in the previous PrepareToPut*. This allocates a 1000 byte file:

```
./srm2_testPut ba12 1 0 srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/oversize 100 0 1000
```

[4]The function returns "error 16"[12] which stands for SRM_NOT_SUPPORTED.

```
returned r_token b27fdc21-46b6-4672-b97b-031b4a23ba28
After the srmStatusOfPutRequest Call ...
state[0] = 23, TURL = gsiftp://pccms5.cmsfarm1.ba.infn.it/pccms5.cmsfarm1.ba.infn.it:/[...]/oversize.661.0
```

Now we are able to store a 200MB file through gsiftp, and then to do PutDone (which returns SRM_SUCCESS):

```
./srm2_testPutDone b27fdc21-46b6-4672-b97b-031b4a23ba28 \
srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/oversize
request state 0
SURL = srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/oversize
```

However, the standards are not clear about this point. Anyway, quoting the developers of DPM:

> "PrepareToPut does not implement an hard limit on filesize. This has been discussed in SRM Collaboration meetings and probably all developpers also implement a soft limit."

ExtendFileLifeTime extends the lifetime of a pinned file.

✓ ***DPM*** ExtendFileLifeTime seems to work. Anyway, we will confirm this test in the future.

GetRequestSummary returns a detailed report on a request.

✓ ***DPM*** After a PrepareToPut, GetRequestSummary returns:

```
Finished files: 1
Processing files: 0
Number of files: 1
Type of req: PrepareToPut
Request token: 7f6950f6-a0d7-40c6-9f73-09455059a4f8
```

and exits with code 0.

GetRequestID returns the *request tokens* associated by the server to a particular user.

✓ ***DPM*** We sent some requests associated to the same user token; then we launched GetRequestID, verifying that it works as expected.

AbortRequest stops/aborts a queued/running request.

✓ ***DPM*** DPM developers said:

> “AbortRequest is partially implemented: you may abort a request in the queue, but you cannot abort a filerequest being processed.”

We have written a “crash test”, which consists in a PrepareToPut iteration, in which each PrepareToPut allocates 1GB, until allocating all available disk space. Then the server has rejected subsequent requests.

AbortRequest has permitted to disallocate all PrepareToPut allocations. However, we noticed that if you don’t call AbortRequest then the space remains allocated over its lifetime.

SuspendRequest/ResumeRequest Suspend/Resume a request.

× ***DPM*** Quoting DPM developers:

> “For the moment we have no plan to implement Suspend/Resume.”

Copy Performs SRM-SRM transfers.

× ***DPM*** DPM developers are currently working on both versions of Copy (1.1 and 2.1).

× ***StoRM*** Copy has been tested on StoRM with its own client, showing a problem related to the mode StoRM GridFTP server uses to code files; that mode must be changed from *ASCII mode* to *binary mode*.

Developers said they would change that behaviour as soon as possible.

3.4.2 Directory functions

Ls performs directory listing. *Recursive listing* is useful, even if it is very expensive.

× ***DPM*** The output of Ls is not a directory listing but a metadata report on that directory:

```
./srm2_testLs -l srm://pccms5.cmsfarm1.ba.infn.it:8444/dpm/cmsfarm1.ba.infn.it/home/cms
request state 0
request state 0 1
Stat: 0
Path: /dpm/cmsfarm1.ba.infn.it/home/cms
Size: 0
```

```
Type: 1
OwnR: root
Crea: 2006-01-26T22:47:12Z
Last: 2006-01-26T22:47:12Z
OwnP: 7
OthP: 5
GrpP: 1 cms:7
```

Mkdir, Rmdir, Rm, Mv let you create and remove (empty) directories, remove files, move directories and files.

✓ ***DPM*** All these functions work as expected.

3.4.3 *Space reservation*

ReserveSpace, ReleaseSpace, UpdateSpace globally reserve/release/update space on the SRM, through allocating a (*container*) and returning *space token*.

× ***DPM*** As stated by developers:

> "Global space reservation will come during the spring. At the workshop in Mumbai[15], the priority assigned to this item was relatively low."

They are work in progress; anyway, we have verified a partial implementation of a metadata management.

srm2_testReserveSpace can be used as follows to allocate 2 GB of *volatile* ("0") disk space, for 10000 seconds (lifetime), identified by the token "blabla".

So sizes must be written in byte, lifetime in seconds; the last parameter (typeOfSpace) is a flag: if set, it means *durable=permanent*, else *volatile*. See these information directly in the C sources.

```
./srm2_testReserveSpace  srm://pccms5.cmsfarm1.ba.infn.it:8444 blabla info 2000000000 2000000000 10000 0
request state 0
soap_call_ns1__srmReserveSpace returned
s_token: 361f2155-6a59-4c04-bb0c-e143b4dd705f
srmReserveSpace provided
actual_s_type: 0
actual_t_space: 2000000000
actual_g_space: 2000000000
actual_lifetime: 10000
```

Space reservation metadata have been written in the DPM database, but this is the only reservation operation made by DPM belonging to reservation.

× ***StoRM*** ReserveSpace belonging to the StoRM client can reserve up to 2GB, *which is the biggest value that can be represented by the integer type in XML-RPC*[14]; to exceed 2GB, you must do more than one request.

Besides, the garbage collector doesn't care about lifetime expiration. Developers told us that the garbage collector just works (up to now) on implicitely reserved space (PrepareToPut).

3.4.4 Permission functions

SetPermission/CheckPermission lets you set/get Posix-style permissions on files and directories.

✓ ***DPM*** SetPermission and CheckPermission work:

```
./srm2_testSetPermission 2 4 4 srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/bigfile \
1 - gridit 4
request state 0
./srm2_testCheckPermission srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/bigfile
request state 0
state[0] = 0, Perm = R, SURL = srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/bigfile
./srm2_testSetPermission 2 6 6 srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/bigfile \
1 - gridit 4
request state 0
./srm2_testCheckPermission srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/bigfile
request state 0
state[0] = 0, Perm = RW, SURL = srm://pccms5.cmsfarm1.ba.infn.it:8444//dpm/cmsfarm1.ba.infn.it/home/gridit/bigfile
```

3.4.5 SRM-SRM interoperability with Copy

As you see in table 3.2:

- dCache provides SRM 1 Copy;
- Storm provides SRM 2 Copy;
- DPM provides no Cpoy.

In SRM-SRM transfer, *at least one of the two SRMs must provide the Copy support*; that SRM will be the server side of the connection; thus, all transfers from an SRM to itself are possible just if that SRM provides Copy. dCache and StoRM can do that, while DPM doesn't: it can be client of dCache or StoRM.

Doing some transfer through dCache *srmcp*, we used the *"push mode"* option to perform a *srmCopyBySource* instead of *srmCopyByTarget*. Infact transfers from dCache to DPM will work fine if you specify *-pushmode=true*: in this way the request can be forwarded to dCache (source). If you leave *"pull mode"* (default), then the request is forwarded to DPM (target) which is not able to manage it.

		target		
		dCache	**DPM**	**StoRM**
	dCache	✓	✓	×
source	**DPM**	✓	×	?
	StoRM	×	?	?

Table 11: Interoperability with SRM Copy

Let's summarize the situation of interoperability, concerning Copy in in table 11, in which:

- the symbol ✓ tells us that Copy is possible;
- the symbol × tells us that Copy is not possible;
- the symbol ? means "tests not performed".

StoRM-StoRM transfers have not been possible, due to logistical troubles in installing a second instance of StoRM. However, a StoRM-StoRM Copy using a single instance of StoRM both as source and target, but it doesn't work.

3.5 Semiautomatic SRM test script

We wrote down a script called *satest2.sh*, and used it on DPM 1.5.4 (and previous versions). This script wraps SRM client, mainly the DPM test suites, testing the "macroscopic" behaviour of the storage manager in certain circumstances: this includes pinning, space reservation constraints, mo more free space left on disk pools etc.

You can configure it by few parameters, and performs these tests:

put PrepareToPut of a 200MB file, and then its *globus-url-copy* and PutDone;

get PrepareToGet of a 200MB file, and then its *globus-url-copy* and ReleaseFiles;

rmonput/rmonget as above, but making an attempt to remove the file while the transfer is in progress; this tests *implicit pinning*;

putoversize performs a globus-url-copy of a 200MB file, but preallocating just 300 bytes, to verify if there is any control on filesize by the SRM;

putoverspace is a *crash test*: it repeatedly executes PrepareToPut *until free space runs out*; this verify if the SRM is notified about saturation;

dirs tests on Mkdir, Rmdir, Mv, Rm, Ls through a 3-level tree with files and directories.

All tests above are based on the tools listed in table 12.

Test name	**Function**
get	PrepareToGet, GetRequestID, GetRequestSummary, ReleaseFiles, RemoveFiles
rmonget	PrepareToGet, GetRequestID, GetRequestSummary, ReleaseFiles, RemoveFiles, Rm
put	PrepareToPut, GetRequestID, GetRequestSummary, PutDone
rmonput	PrepareToPut, GetRequestID, GetRequestSummary, PutDone, Rm
putoversize	PrepareToPut, GetRequestSummary, PutDone
putoverspace	PrepareToPut, AbortRequest
dirs	PrepareToPut, PutDone, Ls, Mkdir, Rmdir, Mv, Rm

Table 12: List of functions tested through *satest2.sh*

By default, the script stops on all errors.

To use the script, you must run it once in an empty directory, so it create a *configuration template* (*CONFIG*), which must be customized. Then you should run it again to let it download, build and install the DPM test suites. If it builds successfully, then you can run it and choose the test to execute.

The script *satest2.sh* is available on the Storage Group Website[5].

4 Conclusions

At the moment, no storage manager seems to satisfy completely the experiments' needs as far as SRM is concerned. In particular, *there is no SRM server compatible with both versions of SRM*. We must wait for dCache to release its SRM 2 interface, and DPM to complete and correct its own implementation of SRM 2. StoRM risks not to be used in production as long as SRM 1 is required together with new SRM 2 functions.

All the result reported in this note is updated on January 2006. All the software tested is now released in a newer version, with more functionalities added, and we hope to publish soon a newer note in order to evaluate this improvement.

As far as performances are concerned, we tested dCache and GPFS, verifying that:

- dCache spends less than a second in opening files, sustaining a rate of about $250MB/s$ both while reading and writing;
- GPFS is a little faster in reading files, but *it gradually collapses while opening for writing and writing itself*.

  Besides, *the use of the replica manager in dCache can dramatically improve performances if you configure your grid worker nodes as dCache pools: it happens that a job performs its file accesses on the worker node he runs on*.

  During the "SC4 / pilot WLCG Service Workshop"[15], which had place in February 2006 at Mumbai, many developers discussed about the common direction to follow about storage manager implementations.

5 Acknowledgements

The test production has been possible with the needful contribution of CNAF and INFN Storage Group. Here is the complete list of authors:

- V. Vagnoni and D. Bonacorsi (INFN Bologna);
- L. Dell'Agnello, G. Lore, L. Magnoni, B. Martelli, P.P. Ricci, V. Sapunenko and R. Zappi (INFN-CNAF).

References

[1] "PostgreSQL: The world's most advanced open source database",
http://www.postgresql.org/

[2] "dCache.ORG",
http://www.dcache.org/

[3] "dCache, the Book",
http://www.dcache.org/manuals/Book/

[4] "Java technology",
http://java.sun.com/

[5] INFN - Storage Group Web Site,
http://grid.ct.infn.it/swiki

[6] "Scientific Linux CERN 3",
http://linux.web.cern.ch/linux/scientific3/

[7] “SRM Working Group”,
http://sdm.lbl.gov/srm-wg/

[8] “IBM General Parallel File System”,
http://www-03.ibm.com/servers/eserver/clusters/software/gpfs.html

[9] “StoRM”,
http://grid-it.cnaf.infn.it/storm/

[10] “Iperf - Network performance tester”,
http://dast.nlanr.net/Projects/Iperf/

[11] “LCG Disk Pool Manager (DPM) Administrator Guide”,
https://uimon.cern.ch/twiki/bin/view/LCG/DpmAdminGuide

[12] “SRM 2.1 Status Codes”,
http://sdm.lbl.gov/srm-wg/doc/SRM.StatusCode.v2.1.Expl.pdf

[13] “srmPrepareToPut”,
http://sdm.lbl.gov/srm-wg/doc/SRM.spec.v2.1.1.html#srmPrepareToPut

[14] “XML-RPC Specification”,
http://www.xmlrpc.com/spec

[15] “Conclusions from SC4 workshop - Data management”,
http://agenda.cern.ch/askArchive.php?base=agenda&categ=a056461&id=a056461/document