DOME
A rest-inspired engine for DPM

February 24, 2016

This is a preliminary project document whose goal is to drive the development of DOME. Expect very pragmatic prose. Together with the development itself, the content of the document will converge to becoming a white paper containing the documentation of DOME.

Contents

1 Dome

This initiative aims at augmenting the Disk Pool Manager (DPM) system so that its core coordination functions and inter-cluster communication paths are implemented through open components, and following contemporary development approaches headed to performance, scalability and maintainability. Among our goals we cite:

- Making optional all the so-called legacy components that are provided by the *lcg-dm* code tree, namely *libshift*, *rfiod*, *dpm(daemon)*, *dpnsdaemon*, *CSec* and others.
- provide a software infrastructure where adding new coordination features is easier than with *lcg-dm*
- provide full support for asynchronous calculation of file checksums of multiple types
- provide support for checking the consistence of replicas through their checksums
- provide structure, hooks and callouts that allow the usage of DPM as a fast and large *file cache*
- having a unified configuration file that is readable and synthetic, as opposed to the *lcg-dm* approach of having several configuration files here and there, all with differently over-simplified syntax rules (or no syntax at all, e.g. */etc/NSCONFIG*)

The DOME component has the shape of a *fastCGI* daemon, and has to be triggered by the Apache instances running in the DPM head node and in all the DPM disk servers. A configuration option defines whether it is running as head node or disk server.

For simplicity of expression, in this document we may refer to these modalities as two different components, named *DOMEhead* and *DOMEdisk*. In practice, these are the same component which has been given a different command-line flag to enable/disable a different command set, implemented in the same software skeleton.

DOME is a client of the *dmlite* framework, in particular for the features that now are fulfilled by dpnsdaemon. It's also a service provider for the dmlite framework, through the dmlite plugin *adapter_rest*, described later.

1.1 DOME: Main features

DOME has two modalities: headnode and disknode, which respectively represent evolutions of the *dpm* daemon and of *rfiod*, together with *libshift* and *Csec*. The functionalities are roughly as follows:

- headnode: general coordination function

  spreads load (PUT, GET, checksums) towards the available disk nodes

  keeps an in memory status of the DPM disk/pool topology with disk sizes and free space

  keeps an in memory status of the ongoing asynchronous checksum calculations

  keeps an in memory status of the ongoing asynchronous file callouts

  queues and dispatches to disk nodes the requests for asynchronous checksum calculations that have to be delayed for load balancing reasons

  queues and dispatches to disk nodes the requests for asynchronous file callouts that have to be delayed for load balancing reasons

  uses the dmlite library to access logical information about pools/filesystems and name space content

- disknode: local disk and space-related services

  Allows to *stat* individual physical paths

  Allows to *statfs* of filesystems to get size and free space

  Allows the local submission of checksum calculations

  Allows the local submission of file callouts

The main difference from the legacy components is that DOME does not apply authorization again for individual user file access, as this task is already accomplished by the dmlite frontends. DOME only checks that the sender of a request is authorized to send requests, in a way that is similar to the libshift "root mode". DOME applies strong authentication protocols to this task.

DOME is protocol-agnostic. Its concepts of logical file name and physical file name are not linked to a particular data/metadata transfer protocol. DOME manages paths and filenames, not URLs. URLs can be constructed by the DPM frontends starting from the pfn or lfn information given by DOME.

1.1.1 From spacetokens to quota tokens

Historically, DPM does space accounting through a set of individual named space reservations, kept in the DB in the head node, and associated to pools. Semantically, space reservations are named reservations of a part of the space of a disk pool. Requests to write a replica specify a pool that has to host the replica, hence ultimately the replica will be subject to the space reservations.
One of the weakest points in this schema is that the writer has to know technical details of the destination storage, to be able to write and be properly accounted for.

The development direction of DOME is to evolve this mechanism towards *subdirectory-based space accounting*, instead than pool-based.

In the current production version of the dmlite framework (0.7.3, Sept 2015), space accounting on the first levels of directories is a feature that is already available. DOME has access to this information through the normal dmlite Catalog API.

DOME uses the records describing spacetokens that are kept in the head node DB, with minimal modification. Their meaning is slightly changed, into semantically representing a quota on one and only one directory subtree.

For simplicity of management, a quota token attached to a directory subtree **overrides others that may be attached to its parents**. This also helps reducing the complexity of the checks.
If a directory content (counting all the replicas) exceeds the quota, then new PUT requests on that dir will be denied.

In the most common DPM setups, legacy spacetokens are de facto used to assign space limits to a VO or to one of its service directories, named after space tokens.
The described semantic evolution is supposed to be crafted in order not to interfere with the spacetoken support implemented by the legacy components in the cases where space tokens were used in that way.

1.1.2 Pools and filesystems

A pool is a logical group of mount points in individual disk servers that are used to store replicas.

DOME uses the same concept of Pool than the historical DPM, hence the "Pool management" functionalities of the lcg-dm components will continue to work mostly as they are.

DOME considers pools as referred to path prefixes, that is directory subtrees. A pool assigned to a directory completely overrides pool assignments that belong to its parent directories.

A Pool assigned to a subtree acts as a sort of "replication domain". Replicas of files belonging to that subtree are stored in filesystems belonging to this pool. Multiple replicas are spread through different file systems.

Multiple pools can be assigned to the same directory subtree (e.g. /dpm/cern/ch/home-/dteam/scratch). Writes into this subtree will be space-balanced between all the filesystems composing the pools assigned to it. Some pools may also be used to host only replicas of content that is already available in other pools.

1.1.3 Open checksumming

DOME supports requests for checksums of arbitrary kind. It can:

- return the corresponding checksum that is stored in the name space
- choose an appropriate replica of the file and tell to the disk node managing it to calculate its checksum
- force the recalculation of the checksum and store it into the name space

The checksum calculation request may be queued in the head node, in memory. The architecture is designed to be self-healing in the case the checksum calculations do not end correctly, or some machines are restarted.

1.2 Tech

The architecture of DOME has to be expandable, which does not necessarily mean that it's excessively plugin-based. Where to add the impl of a new request must be clear and simple to understand.
DOME is a client of dmlite, i.e. it can invoke the dmlite catalogue, pool, etc. functions.

1.2.1 Architecture

Deployment diagram adapter_rest take Eric's picture and comments

1.2.2 Security

The configuration file can specify criteria to accept requests that come to DOME. These criteria have the form of a list of allowed DNs (taken from X509 certificates).

Main goals: strong security which can be disabled (for debugging purposes) in a way that is trivial for the sysadmin to understand, apply, disable, check, doublecheck.

The typical configuration is that only the DN coming from the host certificate of the head node can interact with DOME.
The typical configuration uses HTTPS in the frontend configuration, to enforce the usage of a valid certificate.

1.2.3 Checksum queuer

DOME internally queues and schedules checksum calculation requests in the head node.
No more than N checksums will be run per disk mount
No more than L checksums will be run per disk server
No more than M checksums will be run in total

Checksum requests are queued in memory and dispatched to suitable disk nodes that become available with respect to the mentioned criteria. The disk nodes instances constantly update the head node on the running checksums, hance there is no need for persistence, and the system will self-heal on restarts of the head node. When finished calculating a checksum, a disk node will notify the head node and pass the result (or

failure).
Eventually memcached can be used for queue synchronization purposes, only if it turns out that even in SOME cases the code is not totally preventing the spawning of new processes (which has never to happen!!!). This evenience would require more development effort, and would have the advantage of making the dpm service able to scale horizontally. So far, we have no evidence that the dpm service needs that.

1.2.4 File pulls queuer

DOME internally on the head node queues and schedules requests for file pulls from external locations No more than N pulls will be run per disk mount
No more than L pulls will be run per disk server
No more than M pulls will be run in total

The pull itself is implemented as a simple callout in the disk server, that can invoke any file movement mechanism, from dd to create an empty file to a simple copy to uber-complex multi hop FTS xfers. The pull callout in the disk server is complemented by a stat callout, which is able to stat an external system for the presence of an offline file.
This mechanism should be polished enough to support the construction of simple file caches, without necessarily needing external, complex components. Invoking FTS instead than *dd* or *davix-get* must be an option.

Pull requests are queued in memory and dispatched to disk nodes that match the request and become available. Please note that stat requests are not queued. Please also note that the DOME API has no stat primitive.
The disk nodes instances constantly update the head node on the running callouts, hance there is no need for persistence, and the system will self-heal on restarts of the head node. When finished pulling a file, a disk node will notify the head node and pass the result (or failure).

Eventually memcached can be used for queue synchronization purposes, only if it turns out that even in SOME cases the code is not totally preventing the spawning of new processes (which has never to happen!!!). This evenience would require more development effort, and would have the advantage of making the dpm service able to scale horizontally. So far, we have no evidence that the dpm service needs to scale horizontally the head node.

1.2.5 Only one process

The fastcgi app named DOME has only have one process and multiple internal thread pools. This simplifies a lot the development.

NB: lcg-dm contains generic utilities too, sql stuff, DB upgrade scripts, metapackages etc. These things should be moved somewhere else, possibly a place that refers to a part of the project that is not optional and that has low maintenance needs.

1.3 Application programming interface

Historically DPM implements low level functionality that is used by frontends to coordinate their activities of exposing data access protocols to clients.
In some cases, the historical DPM API has been also exposed to clients/users, eventually through a Storage Resource Manager (SRM) server.

DOME is not supposed to be used by remote clients and users. Users interact with DPM through a suitable frontend (e.g. gridFTP, xrootd, Apache) that relies on the services of dmlite and DOME in the background.

1.3.1 DPM historical primitives

The goal of this section is to present a quick list of the historical API of the DPM daemon, for subsequent reference. For details about the various calls, the reader is encouraged to refer to the respective manpages or to the code.

- DPM_ABORTFILES:
- DPM_ABORTREQ:
- DPM_ADDFS:
- DPM_ADDPOOL:
- DPM_COPY: copy from surl to surl. Bound to rfio.
- DPM_DELREPLICA:
- DPM_EXTENDLIFE: unclear, not manpage documented

- DPM_GET:
- DPM_GETPOOLFS: mgmt, already in dmlite
- DPM_GETPOOLS: mgmt, already in dmlite
- DPM_GETPROTO: unclear, not manpage documented
- DPM_GETREQID: explicit async way to queue requests. Never used AFAIK ?
- DPM_GETREQSUM: unclear, not manpage documented
- DPM_GETSPACEMD: get spacetoken info. Unclear why it's bulk request. Some of the fields are unnecessarily complex or with involuted definitions.
- DPM_GETSPACETKN: unclear, not manpage documented
- DPM_GETSTSCOPY: unclear, not manpage documented. Seems related to the COPY command. Never used AFAIK ?
- DPM_GETSTSGET: unclear, not manpage documented. Seems related to the status of GET command through SRM
- DPM_GETSTSPUT: Not manpage documented. Polling mechanism to accommodate writes to disks or tapes.
- DPM_INCREQCTR: unclear, not manpage documented. Never used AFAIK ?
- DPM_MODFS: mgmt, already in dmlite
- DPM_MODPOOL: mgmt, already in dmlite
- DPM_PING: the best!
- DPM_PUT: main functionality that we miss
- DPM_PUTX: do we really need to make it a bulk request ? Maybe yes if we define hooks and callouts
- DPM_PUTDONE: sob
- DPM_RLSSPACE: unclear, not manpage documented
- DPM_RELFILES: documented as "release a set of files" . not clear if it makes sense
- DPM_RSVSPACE: unclear, not manpage documented

- DPM_RM: mgmt, already in dmlite
- DPM_RMFS: mgmt, already in dmlite
- DPM_RMPOOL: mgmt, already in dmlite
- DPM_SHUTDOWN: OK, we get it
- DPM_UPDSPACE: unclear, not manpage documented
- DPM_UPDFILSTS: unclear, not manpage documented
- DPM_ACCESSR: checks the existence or the accessibility of the file replica according to the dpm. The name server entry for the replica is taken into account and that of the associated pool and, if relevant, the status of an ongoing put request. The physical file name pfn is checked according to the bit pattern in amode

NOTES:

- chksum calculation/mgmt is incomplete, in the best cases it's inflexible
- NOTE: the PUT polling mechanism is among the main responsibles for the latency of writes into DPM
- NOTE: many of these requests have been exposed through the SRM layer. It's unclear what influenced what. We should feel free to be non-SRMish
- The DPM daemon uses rfio as generic subsystem for inter-cluster communication and data sharing

1.3.2 RFIO historical primitives

These are used in the adapter, mainly for GridFTP tunnelling purposes:

- rfio_lseek
- rfio_parse
- rfio_open
- rfio_close
- rfio_write

- rfio_read
- rfio_flush
- rfio_stat64

These are used in the DPM daemon, mainly for metadata and disk stats:

- rfio_errno
- rfio_serror
- rfio_stat
- rfio_mkdir
- rfio_chown
- rfio_stat64
- rfio_allowed
- rfio_statfs64
- rfio_rcp (used to replicate files... maybe we don't need this)

1.4 Command sets of DOME

Goals:

- keep the system architecture, databases, format of the physical file names
- coherent support for multiple types of checksums
- substitute dpmd, libshift and rfio, in favor of HTTP and REST
- simplify the semantics of the commands with respect to the SRM-ish one

Each command is encoded as a RESTful request, where only the command name is URL-encoded, and every other dpm-specific parameter is encoded in a JSON snippet supplied as BODY of the request.

A legitimate response can be a 202-Accepted with or without a body field that gives a token to be used to retry.

Each client request is implemented on a simplified client API based on davix. Each command is ALSO implemented on a command line tool that has the same name, the same parameters and prints the output in a pretty and readable way

1.4.1 Common header fields

remoteclientdn

The DN of the original client that submitted the request. Typically, a Grid user with X509 credentials.

remoteclientaddr

The IP address of the original client that submitted the request.

1.4.2 Head node operation

dome_put

initiates a replica upload. The client is given a location where to write the replica (redirection)

Command: `POST /dome/<logical file path>`

Request header:

`cmd=dome_put`

Params:

- lfn: logical file name of the entry
- additionalreplica: true—yes—1 specify to upload one more replica to an lfn that already has
- pool: suggested pool where to write (optional)
- host: suggested host where to write to particular filesys (optional)
- fs: filesystem prefix where to write the new file (optional). If specified, then host becomes mandatory. DOME will compute the remaining part of the full physical filename.

Returns:

- 200 if OK. Other HTTP codes for the corresponding errors.
- pool: chosen pool
- host: chosen host:port
- pfn: physical filename to be used

dome_putdone

Notifies that the upload of a replica finished successfully. It also can carry a checksum type/value that may have been calculated during the upload.

Workflow:
The notification from the data access frontend (e.g. GridFTP) always goes to the DOME instance that is running in the **disk node**. This means that generally the notification will be sent to *localhost*.
The DOME in the disk node doublechecks the existence of the file, the correctness of the path and the correctness of the file size that the frontend presumes.
If the local checks are passed, the request is forwarded automatically to the instance of DOME running in the head node.

Command: `POST /dome/<logical file path>`
Request header:
`cmd=dome_putdone`

Params:

- pfn: Physical filename (doublecheck if it's the case not to use the vetust rfio syntax)
- size: size of the file
- checksumtype: Type of checksum (optional)
- checksum: Checksum value (optional)

Returns:

- 200 if OK. Other HTTP codes for the corresponding errors.

dome_getspaceinfo

Returns total and free space information for all the pools and filesystems at once (the list is supposed to be in memory all the time)

Command: `GET /dome/`

Request header:

`cmd=dome_getspaceinfo`

Params:

-

Returned information:

- all the pools
- **poolstatus**: the status of the pool. 0 means active.
- **physicalsize**: the total space physically available for this pool
- **freespace**: the free space
- all the server and mountpoints that this pool contains

  for each server:mountpoint, the total space, and the free space, and the status of the mountpoint (0 means active)

JSON example:

```
{
    "fsinfo": {
        "fab-dpm-dev0.cern.ch": {
            "\/testfsdata": {
                "poolname": "fabpool",
                "fsstatus": "0",
                "freespace": "0",
                "physicalsize": "0"
            },
            "\/yukyuk": {
                "poolname": "fabpool",
                "fsstatus": "0",
                "freespace": "0",
                "physicalsize": "0"
            }
```

```
        },
        "pcitsdcfab.cern.ch": {
            "\/tmp": {
                "poolname": "fabpool",
                "fsstatus": "0",
                "freespace": "194393481216",
                "physicalsize": "228677218304"
            }
        }
    },
    "poolinfo": {
        "fabpool": {
            "poolstatus": "0",
            "freespace": "194393481216",
            "physicalsize": "228677218304",
            "fsinfo": {
                "fab-dpm-dev0.cern.ch": {
                    "\/testfsdata": {
                        "fsstatus": "0",
                        "freespace": "0",
                        "physicalsize": "0"
                    },
                    "\/yukyuk": {
                        "fsstatus": "0",
                        "freespace": "0",
                        "physicalsize": "0"
                    }
                },
                "pcitsdcfab.cern.ch": {
                    "\/tmp": {
                        "fsstatus": "0",
                        "freespace": "194393481216",
                        "physicalsize": "228677218304"
                    }
                }
            }
        }
    }
}
```

dome_getquotatoken

Gets a quota token, using the path prefix as a key. The path prefix must be an existing directory. If the path has an appended "*", the output will contain all the quota tokens that belong to the given directory subtree. It also returns the space that is still available for each of the quota tokens listed.

Command:

`GET /dome/path`

Request header:

`cmd=dome_getquotatoken`

Params:

- getsubdirs: if true, the output will include quotatokens that refer to the parent directories of the query
- getparentdirs: if true, the output will include quotatokens that refer to the subdirectories of the query

Returns: 200 if OK. Other HTTP codes for the corresponding errors.

- a sequence of :

  path: the logical path a quota token is referring to

  quotatkname: Human readable name for the quota

  quotatkpoolname: The pool that serves this quotatoken

  quotatktotspace: The max number of bytes that anyone will be allowed to write into this path if there is enough free space in the pool.

  pooltotspace: total space on the assigned pool

  pathusedspace: how much space is occupied by files in that path

  pathfreespace: how much data one could still write into that path

dome_setquotatoken

Sets or create a quota token, using the path prefix and the poolname as a key. The path prefix must be an existing directory, the poolname should be the name of an existing pool.

Command: `POST /dome`

Request header:

`cmd=dome_setquotatoken`
Params:

- path: the logical path a quota token is referring to
- poolname: the pool that will host the replicas that are written into paths associated to this quotatoken
- quotaspace: the maximum number of bytes that the subtree rooted at path can acquire through write operations
- description: a human readable description, e.g. ATLASSCRATCH

Returns: 200 if OK. Other HTTP codes for the corresponding errors. If the quota being set exceeds the size of the directory subtree it refers to, DOME will set the quota anyway, and give a warning in the body of the response. The result will be that noone will be able to write in that subtree until a sufficient number of files is removed.

A file write is accounted in the directory tree that contains the logical file name. A quotatoken must be assigned to one of the parent directories to tell DOME which pool the physical write should be directed to. At the same time, the quotatoken will also set a limit (quota) to the maximum number of bytes that can be written.

dome_delquotatoken

Deletes a quota token, using the path prefix and a poolname as a key. The path prefix must be an existing directory.

Command: `POST /dome`
Request header:
`cmd=dome_delquotatoken`
Params:

- path: the logical path a quota token is referring to
- poolname: the pool that will host the replicas that are written into paths associated to this quotatoken

Returns:

- 200 if OK. Other HTTP codes for the corresponding errors.

dome_getdirspaces

Computes used/free space for a path. The path must be an existing directory.

Command: `GET /dome/path`

Request header:

`cmd=dome_getdirspaces`

Params:

Returns:

- 200 if OK. Other HTTP codes for the corresponding errors.

dome_chksum

Checks, calculates or recalculates the checksum of files/replicas.

Command: `GET /dome/pathfile`

Request header:

`cmd=dome_chksum`

Params:

- checksum-type: Kind of checksum that is requested (e.g. adler32, MD5, etc...)
- pfn: Physical filename as it appears in the db (optional)
- force-recalc: true—false—yes—no—0—1 (optional)

Returns:

- Checksum
- PfnChecksum (optional)

Behavior with the ForceRecalc flag unset

If the `ForceRecalc` flag is **not** set, then DOME will check the namespace for an already stored checksum of type X. If it's found in the namespace then it's returned in the body with a return code 200 'Ok'.

If a pfn is provided, then DOME will return the private checksum of that replica **and the one of the lfn**. A client will be able to compare them.

If the requested checksum is not found, then DOME will:

- if checksum of type X is already being calculated for the given resource or one of its replicas, return 202 'pending'.
- if checksum of type X is not being calculated for the given resource or one of its replicas, enqueue the request for calculating it asynchronously, and return 202 'pending'

Behavior with the ForceRecalc flag set

If the `ForceRecalc` flag is set, then DOME will unconditionally recalculate one checksum, using a random replica or the one that is specified in the Pfn parameter. If a Pfn is not specified, then the result of the calculation will be set into the metadata associated to the lfn.
A client sending this request with the `ForceRecalc` flag **set**, and getting a 202 'pending' response will have to retry the request with the `ForceRecalc` flag **unset** in order to get the result.
When the calculation task finishes, the database is

dome_chksumstatus

A disk node that has calculated a checksum (or failed) will invoke this function to store it and notify the head node that it has finished. This is also used as a sort of heartbeat to notify the head node about checksum calculations that are pending or running.

Command: `POST /dome/pathfile`
Request header:
`cmd=dome_chksumstatus`
Params:

- checksum-type: Kind of checksum that was requested (e.g. adler32, MD5, etc...)
- force-recalc: tells if the original request was for a forced recalculation
- checksum: value of the computed checksum (optional if still being calculated)
- update-lfn-checksum: true — false. Tells the head node whether it also needs to update the lfn checksum. (optional if still being calculated)
- pfn: Physical filename that was used to calculate it
- status: Pending—Done—Aborted

- reason: Free string describing errors or similar (for logging)

Returns: 200, unconditionally No response body.

dome_ispullable

Tells whether a file can be pulled from somewhere else, through an appropriate callout in the disk server

Command: `GET /dome/pathfile`

Request header:

`cmd=dome\_ispullable`

Params:

- Params: Free string that can be passed to the callout that verifies the existence of the file

Returns:

- IsOffline: true—false—yes—no—0—1
- Info: app-dependent free string that may be used to notify (or log) location information about the file (optional)

dome_get

Returns a pfn that can be used to read a file, usual stuff.

If the CANPULL flag is set, the file puller callout may be invoked if the file is absent AND is not being already pulled. The result may be 'pending' if the file is being pulled.

Command: `GET /dome/pathfile`

Request header:

`cmd=dome_get`

Params:

- Canpull: true—false—yes—no—0—nonzero

Returns: Code: 200 or pending

- Host: server name
- Pfn: physical filename

dome_pulldone

Notifies that a file pull has finished. Until that moment it can be invoked to send a sort of progress report or heartbeat. This notification is usually sent by a disk server
Command: `POST /dome`
Request header:
`cmd=dome_pulldone`
Params:

- Host: server name
- Pfn: physical filename
- Lfn: Logical filename
- Status: Pending—Done—Aborted
- Reason: Free string describing errors or similar (for logging)

Returns: Code: 200 always No body is required

dome_statpool

Gets total and free space information for one pool
`GET /dome`
Request header:
`cmd=dome_statpool`
Params:

- **poolname**: pool name to stat

Returned information:

- **poolstatus**: the status of the pool. 0 means active.
- **physicalsize**: the total space physically available for this pool
- **freespace**: the free space
- all the server and mountpoints that this pool contains

  for each server:mountpoint, the total space, and the free space, and the status of the mountpoint (0 means active)

JSON example:

```
{
    "poolinfo": {
        "fabpool": {
            "poolstatus": "0",
            "freespace": "194394103808",
            "physicalsize": "228677218304",
            "fsinfo": {
                "fab-dpm-dev0.cern.ch": {
                    "\/testfsdata": {
                        "fsstatus": "0",
                        "freespace": "0",
                        "physicalsize": "0"
                    },
                    "\/yukyuk": {
                        "fsstatus": "0",
                        "freespace": "0",
                        "physicalsize": "0"
                    }
                },
                "pcitsdcfab.cern.ch": {
                    "\/tmp": {
                        "fsstatus": "0",
                        "freespace": "194394103808",
                        "physicalsize": "228677218304"
                    }
                }
            }
        }
    }
}
```


1.4.3 Disk node operation

The purpose of DOME being executed in the disk node is to give the rfio functionalities that are not given by WebDAV/HTTP, and to control the checksum calculations. The invokation of these primitives must be properly authenticated through URL tokens.

dome_dochksum

Immediately start an external process that calculates the checksum and returns it (or error). Upon return (or error), the the disk node invokes dome_checksumdone in the head node with the result.

The DOME disk node is responsible for keeping the head node informed of the checksums being calculated at regular intervals, through te dome_checksumdone command.

Command: `POST /dome/pathfile`
Request header:
`cmd=dome_dochksum`
Params:

- checksum-type: Kind of checksum that is requested (e.g. adler32, MD5, etc...)
- update-lfn-checksum: tells if this request also needs to update the lfn checksum. To be passed later on to the head node.
- pfn: Physical filename

Returns: 200 or various errors if the calculation process cannot be started.

dome_pull

Invokes the file puller callout. At the end of the pull, the dome_pulldone is invoked towards the head node. The DOME disk node is responsible for keeping the head node informed of the file pulls being performed at regular intervals, through te dome_pulldone command.

Command: `GET /dome/pathfile`
Request header:
`cmd=dome_pull`
Params:

- Pfn: Physical filename
- Info: app-dependent free string that may be used to notify (or log) things

Returns: 200 or various errors if the pull process cannot be started.

dome_pfnrm

Removes a physical file or directory from the disks. Valid only in disk nodes.
Command: `POST /dome`
Request header:
`cmd=dome\_pfnrm`
Params:

- pfn: absolute path to the physical file or directory. The prefix must match an existing filesystem.

Returns:

-

dome_delreplica

Removes a replica, both from the logical name space and physically from the disks. Valid only in head nodes.
Command: `POST /dome`
Request header:
`cmd=dome\_delreplica`
Params:

- server: server hosting the replica
- pfn: absolute path to the physical file or directory. The prefix must match an existing filesystem.

Returns:

-

2 Configuration

Here we list all the directives and parameters of DOME for both disk and head modalities. This chapter will become the full configuration reference.

2.1 Command-line parameters

Coming soon

2.2 Configuration file: Structure and location

The path and filename of the main configuration file is specified as a command-line parameter in the command that starts DOME. A common choice for the configuration file is::

```
/etc/dome.conf
```

The main configuration file may contain an INCLUDE directive, in order to allow a setup that contains multiple partial configuration files into a directory like:

```
/etc/dome.conf.d/
```

At the time of writing this document, the low complexity of the configuration file does not necessarily impose such a structure.

2.3 Configuration file: Directives and parameters

The parameters are subdivided nito three sets, respectively global parameters, parameters that are honoured only by a head node, parameters that are recognized only by a disk server.

2.3.1 Configuration file: Common directives for head nodes and disk nodes

INCLUDE

Interpret as configuration files all the files that are contained in the given directory. Only absolute paths are accepted.

Syntax:

```
INCLUDE: <path>
```

`<path>` is a directory containing DOME configuration files.

The configuration files are loaded and processed by the Ugr configuration subsystem in ascending alphabetic order. It's a good idea to create file names that start with a number, representing their loading priority.

Example:
Load all the configuration files that are contained in `/etc/ugr.conf.d/`
`INCLUDE: /etc/dpm.conf.d/`

glb.debug

Sets the DOME log verbosity.

Syntax:

```
glb.debug: <level>
```

`<level>` is the desired debug level, from 0 to 10.

NOTE: DOME internally uses `syslog` to log its activity, in the `user` class. We refer to the `syslog` documentation for the platform in use in order to configure its behavior, e.g. to output the log to a logfile.

NOTE: a debug level higher than 1 severely affects the performance of DOME. Never set it to a value higher than 1 in a production server.

glb.debug.components[]

Allows selecting the internal components that produce log lines. By default all the internal components produce log activity. The presence of this directive in the configuration file allows only the named components to produce log lines.

Every `glb.debug.components[]` line that appears in the configuration is considered as an individual component to enable log production for.

Syntax:

```
glb.debug.components[]: log_component
```

Example:

```
glb.debug.components[]: queue.CKSUM
glb.debug.components[]: queue.PULLER
```

glb.fcgi.listenport

If Dome is run in external server mode it must specify a TCP port number to bind to. Please note that a fastcgi server like Dome, when run as external server must be run using some custom init script. Please refer to the Fastcgi documentation for more information on these details.
If the given port number is 0, or the directive is asent then Dome assumes that the lifetime of this daemon will be managed by Apache.

Default value: 0

glb.db.host

The host where the MySQL service is available for the dpm_db database.

glb.db.user

The username to be used to connect to MySQL.

glb.db.password

The password to be used to connect to MySQL.

glb.db.port

The port number of the MySQL service.
Default value: 0

glb.db.poolsz

The size of the internal pool of MySQL clients.
Default value: 10

glb.restclient.conn_timeout

When contacting other servers with http/rest, use the provided timeout value.

glb.restclient.ops_timeout

When contacting other servers with http/rest, use the provided timeout value.

glb.restclient.ssl_check

If false, the ssl certificate authority check is not enforced.

glb.restclient.ca_path

Path containing the certificate authority files

glb.restclient.cli_certificate

When contacting other servers with http/rest, use the provided identity/certificate. Normally the host certificate or a service certificate.

glb.restclient.cli_private_key

When contacting other servers with http/rest, use the provided identity/certificate. Normally the host certificate or a service certificate.

glb.reloadfsquotas

Dome will automatically refresh its knowledge of quota tokens and file systems. This parameter is the number of seconds between two consecutive refreshes.
Default value: 60

glb.role

Configures this instance as a head node or a disk node, respectively. Syntax:
`glb.role: head|disk`
Default value: head

glb.auth.authorizeDN[]

Array containing DNs that are authorized to send commands. Commands sent by different senders will not be accepted.
Used in a head node, this list includes all the DNs that are used by disk nodes to communicate with the head node. Among the possibilities, all the disk nodes can use the same certificate, in this case only one entry has to be put here.
Used in a disk node, this list contains the DN that is used by the head node to communicate with the disk server.

glb.put.minfreespacemb

Specifies the minimum free space in megabytesbytes for a PUT requests to consider a filesystem for writing into. Default: 4194304

glb.dmlite.configfile

The full path to a DMLite configuration file. To configure DMLite, we refer the reader to the DMLite documentation.

Example:

```
glb.dmlite.configfile: /etc/dmlite-DOME.conf
```

glb.dmlite.poolsize

The number of dmlite instances that are pooled to give internal dmlite services. This number is likely to be in the order of 50-100 in the head node, and the order of 2-10 in the disk servers.

Example:

```
glb.dmlite.poolsize: 10
```

glb.workers

The number of worker threads that execute the requests.
Default value: 300

2.3.2 Specific to head nodes

head.chksumstatus.heartbeattimeout

Maximum time, in seconds, that an entry about a checksum that is being calculated will stay in memory.
Checksums that have been dispatched and that do not send the heartbeat will be internally treated as failed for unknown reasons.
Default: 60

head.maxchecksums

Maximum number of checksums that can be run concurrently in this DPM instance. Additional requests will be queued until the condition is met.

head.maxcallouts

Maximum number of file callouts that can be run concurrently in this DPM instance. Additional requests will be queued until the condition is met.

head.maxchecksumspernode

Maximum number of checksums that can be assigned to a single disk node. Additional requests will be queued until the condition is met.

head.maxcalloutspernode

Maximum number of checksums that can be assigned to a single disk node. Additional requests will be queued until the condition is met.

2.3.3 Specific to disk nodes

disk.headnode.domeurl

Url prefix for the DOME service in the headnode. This is used to contact the head node.

Internally, this URL is also used in disk nodes to determine the hostname of the head node, and authorize its attempts to connect.

Example:

```
disk.headnode.DOMEurl: https://dpmhead-rc.cern.ch/DOME
```

disk.cksummgr.heartbeatperiod

Number of seconds between notifications to the head node about the status of the checksum calculations that are being calculated.
Default: 10

disk.statfs.restrictpfx[]

Array containing path prefixes. A statfs() request is allowed only if it is for a path prefix that is allowed.
Absence of this option means that all paths are allowed.

Example:

```
disk.statfs.restrictpfx[]: /mnt/dpm_mountpoints
```

3 subsystems and development tasks

This section will not be part of the official documentation. It's here for organization purposes.

- server

  server skeleton

  db pools (borrowed from dmlite)

  logger (borrowed from dmlite)

  config subsystem copied from ugr

  one class containing the internal status. Same ticker-alive objects approach like ugr. A singleton approach seems reasonable and will reduce the dev effort.

  one class describing the status of pools and filesystems, able to gather it through dmlite, refreshing every N seconds

  one class describing a request. Is boost::propertytree fine or over-engineered project poison ?

  checksum queuer and manager for running chacksums This class will need some unit tests, working locally with no setup

  file pull queuer, to queue callout requests and manage the running ones This class will need some unit tests, working locally with no setup

  The tech description of the chksum and filepull queuers is very similar. I am not convinced yet that it can be factorized into one class.

  One class that manages a set of spawned commandlines, regularly checks if they are alive and sends the pending notifications to the head node This class will need some unit tests, working locally with no setup

if DOME needs to talk to the db and do queries, it will do it directly, eventually using the same components used by dmlite-mysql

Question: do we still need to keep in the DB the log of all the PUTs ?

- c++ client class

  Maximum simplicity, just a set of calls wrapping requests and responses

  Call signature must be synchronous. Result is always the error code.

  New eventual purely async calls must be an easy addition if needed one day

  Used in DOME_adapter, avoid things that are not dmlite-friendly

  Used inside dopmrest to communicate between head and disknodes

  Uses davix to communicate

  Investigate on advantage of libs to build/parse json. There are many, I like boost property tree. The advantagte over a 20 lines implementation is not clear to me.

- DOME_adapter, which uses the client class

  implements the calls that are already there, using the current adapter as reference

- command line interface, as a set of executables that also constitute a big part of the tests!!

- scheduled tests that use the command line to do operations towards one of the testbeds

NOTE: investigate on possibility to add a list primitive that works on the physical namespace. An alternative could be to do it through Apache/webdav without passing for DOME.