DPM volatile pools

After only ~18 years

CERN

Volatile History

- DPM inherited its db structure from CASTOR, around 2000. It always had definitions for volatile pools, traditionally unused
- Evident that they were foreseen for tape staging, which never happened
- The design was meant to accommodate “stager processes”, quite CASTOR-ish but not really DPM-ish for that time. Would have been bizarre for small/medium sites of 0.1-10PB

- A few years later (2003) the xrootd framework was doing tape staging using a callout mechanism and an external purging queue. This could work for any external file repo, not only tapes (which are not interesting for DPMs). Immediately entered production for BaBar, at SLAC and various tier-1s
- In 2008 the xrootd framework was using this mechanism with an experimental xrootd fed as data source (cfr. ALICE)

- In 2018… why not merging the two ideas, and give to DPM a callout+purging queue mechanism (xrootd-like, and properly queued) to be used with DPM volatile pools and external arbitrary data repos?
- The substitution (2016-2017) of the older “DPM/DPNS daemons” with the new DOME daemon was the right occasion. Clean code, no awkward CASTOR-isms around, much easier job.

File pulling in DPM volatile pools

- We can then mark a pool as V, meaning that the content is disposable
- DOME has two callouts for these activities:
  - stat() which is launched from the head node
  - pull() which is executed in disk servers
  - These usually are scripts pointing to an external system. The final scripts can be customised trivially, using tools like `gfal-copy` (or `dd` like in the examples :-P)

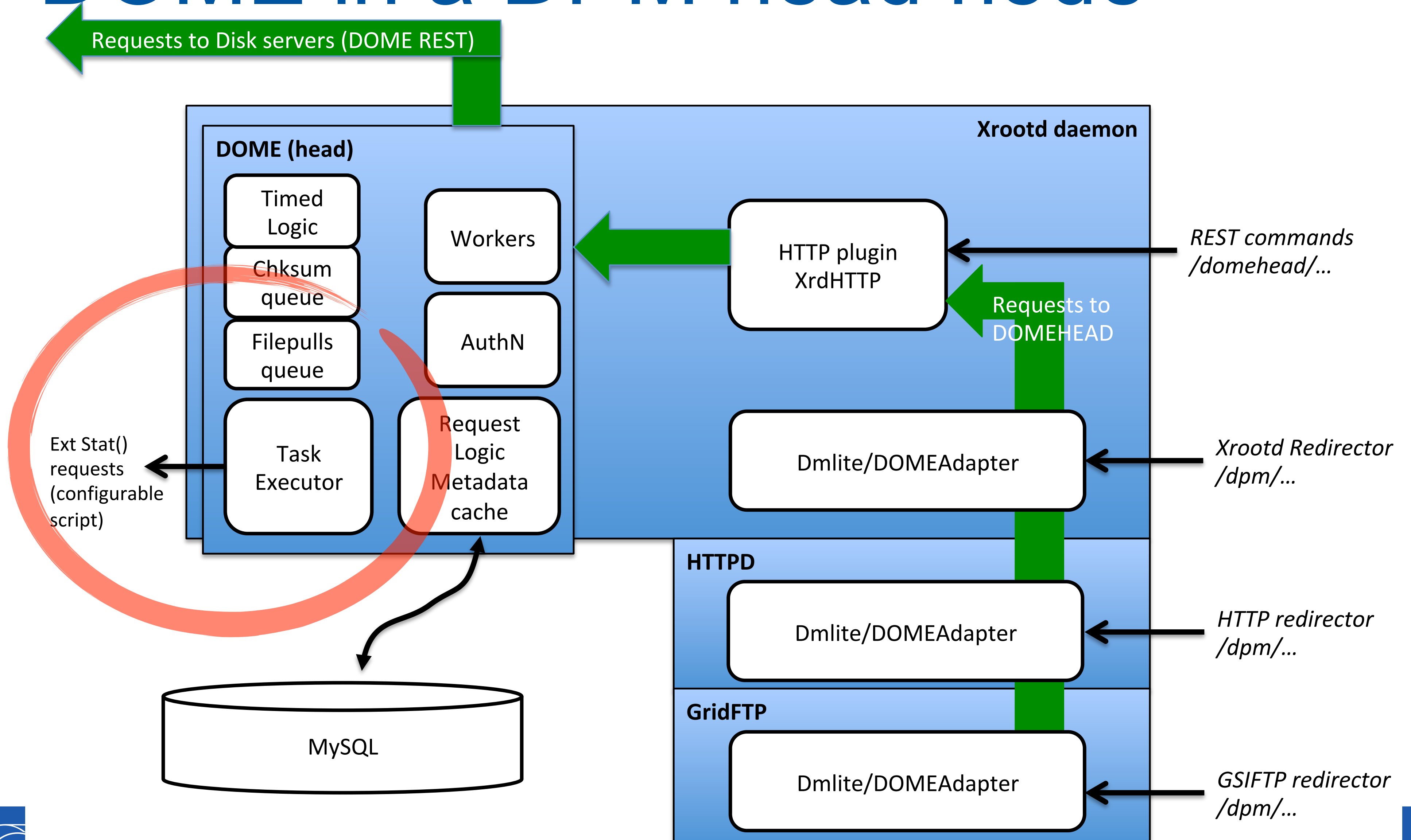
CERN

File pulling implemented

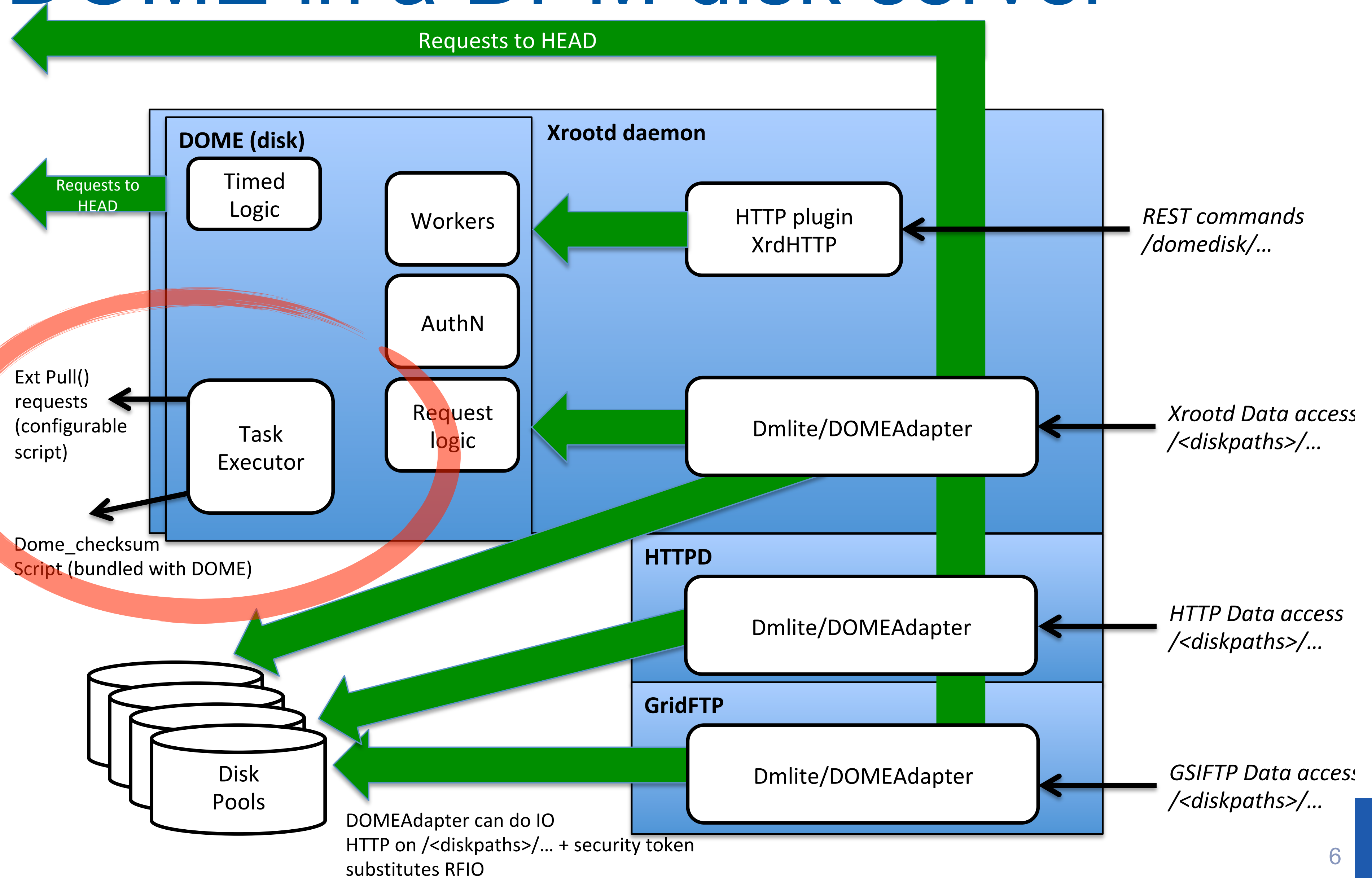
- DOME already had internal components to run nuke-proof callouts (e.g. checksums)
- DOME already had internal components implementing a fast, auto-regenerating in-memory queue to queue and schedule slow external activities. It’s in-memory and survives headnode restarts through a notification mechanism

- DOME got one more instance of such queue, to manage file pulls in disk servers. This allows the sysadmin to prevent overloads and requests storms, by configuring upper limits
- DOME got a dead-simple mechanism to make space in volatile pools when needed, by deleting older files
- That’s it, this is a full file caching mechanism with robustness in mind. Released in DPM since ~Q2/2017.

DOME in a DPM head node



DOME in a DPM disk server



Assumptions on DPM caches

1. A cache is an automatic component that keeps useful data, while discarding less useful data. It's meant to be closer to the data clients (according to some metrics, e.g. throughput, latency, Kms)

2. Access to the cache has to be faster than remote access for a given access pattern
   1. Sometimes true in HEP, depending on the read technique used, e.g. sequential, random, or informed prefetching (e.g. TTreeCache buffers)

4. To bring some advantage, the same data has to be accessed multiple times by clients
   - In HEP workflows this is true only for certain use cases, e.g. an analysis working group
   - Other HEP use cases instead read data embarrassingly once. These do not get performance improvements from caches
   - Pre-placement of data is just one access in a convenient moment

5. In the case of a full-file cache a large portion of each file must be needed by clients
   - Very often true for HEP

- On top of these, there may be other advantages beside performance, e.g. volatility=less site responsibility on hosted data. This kind of aspects are difficult to evaluate.

Full-file caching

- DPM caches whole files
- A job opening a ‘cold’ file starts when the file has come (at the speed of a file transfer), then processes at normal ‘local’ speed
  - Pays the external file xfer latency once. Xfers between sites can be pretty fast.

- Other systems instead cache chunks, the job starts the processing immediately but at the speed of remote data access (for a ‘cold’ file)
  - Hence subject to the efficiency of the job’s data access pattern
  - Pays the external network latency for each external transaction (varying from tens to millions)

DPM file caches

- DPM caches whole files, tape-system-like, different from other chunk-based approaches, e.g. XCache
- The good points are:
  - Predictable behaviour, either the file is there or not
  - Very simple interfacing with any external system, customisable
  - Sysadmin-friendly, e.g. well fitting questions like “does our SE have this file”
- How efficient is it? Interesting and difficult to measure, as it depends largely on the access pattern
- For the ROOT TTree access pattern I (FF) had measured in 2004 (!) it can be sufficiently efficient, and TTree has not changed. I would like to see solid evaluations based on today’s HEP data and real experiment patterns
- How does it perform versus chunk-based caches? Also this largely depends on the access pattern, it could be better in some cases, worse in others

Strong points about DPM caches

- It's there, in the production DPM since mid-2017
- It's a robust thing, easy to understand, easy to integrate in computing models, as a well-defined block
- Fully integrated in DPM (with DOME on), administered with the new DPM admin tools and following ideas that are familiar to sysadmins
  - Works for all the protocols interchangeably: gsiftp, xrootd, http
  - Also authZ/authN is the same… x509, voms, macaroons, …
- Runs on already existing DPMs, no need for parallel installations or new machines, just (re)configuration of a pool
- Full-file makes it easy to federate. One can mix caches and normal storage in the same federation, and redirect to the closest hot cache
- DPM supports also remote pools, residing in other sites. These are good candidates to become volatile caching pools residing in different sites
- If the federation tech is Dynafed/HTTP then the integration with Cloud storage is seamless

- Will be interesting to see use cases, ideas and measurements for new real cases