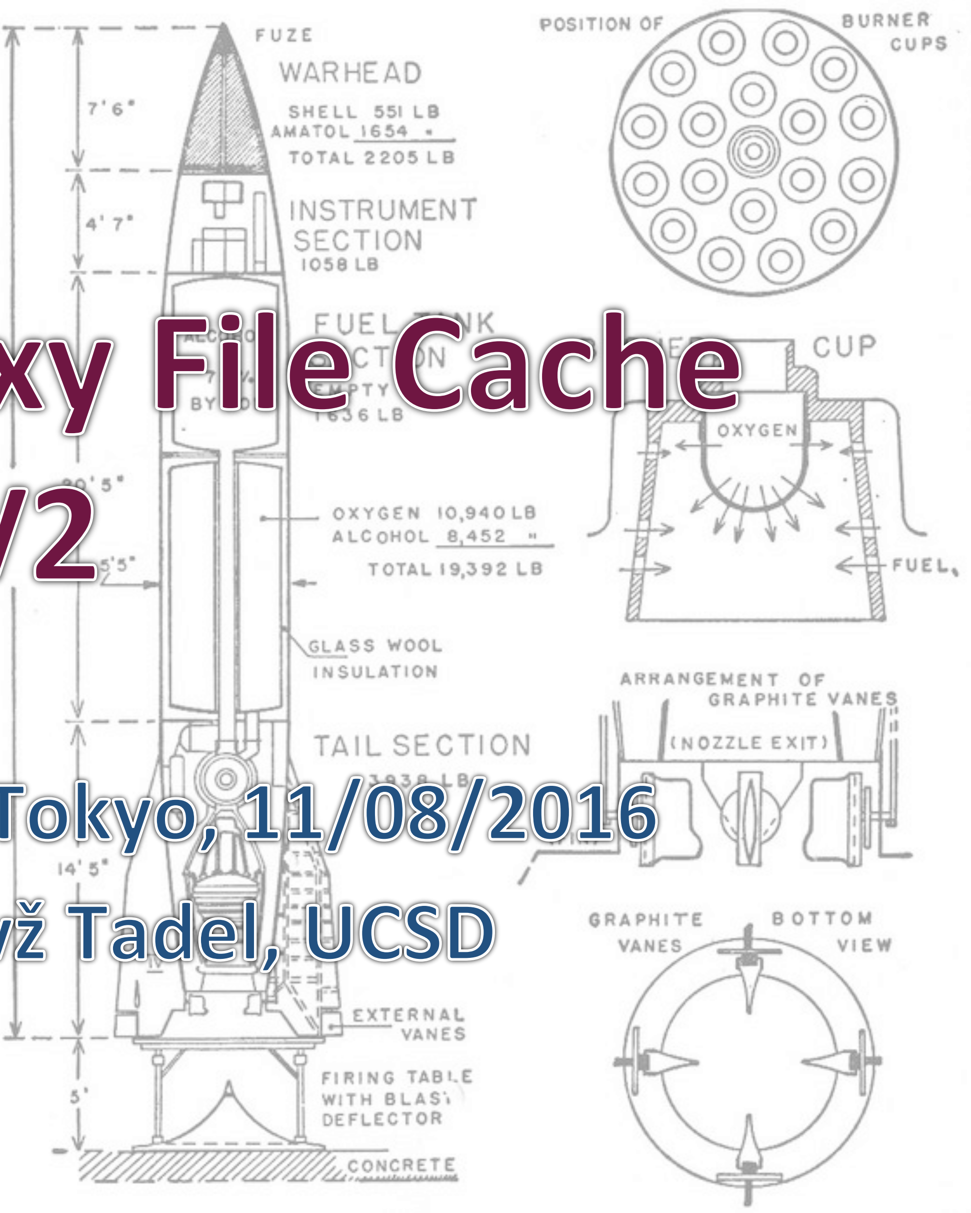


XRootd Proxy File Cache V2

XRootd @ ICEPP Tokyo, 11/08/2016

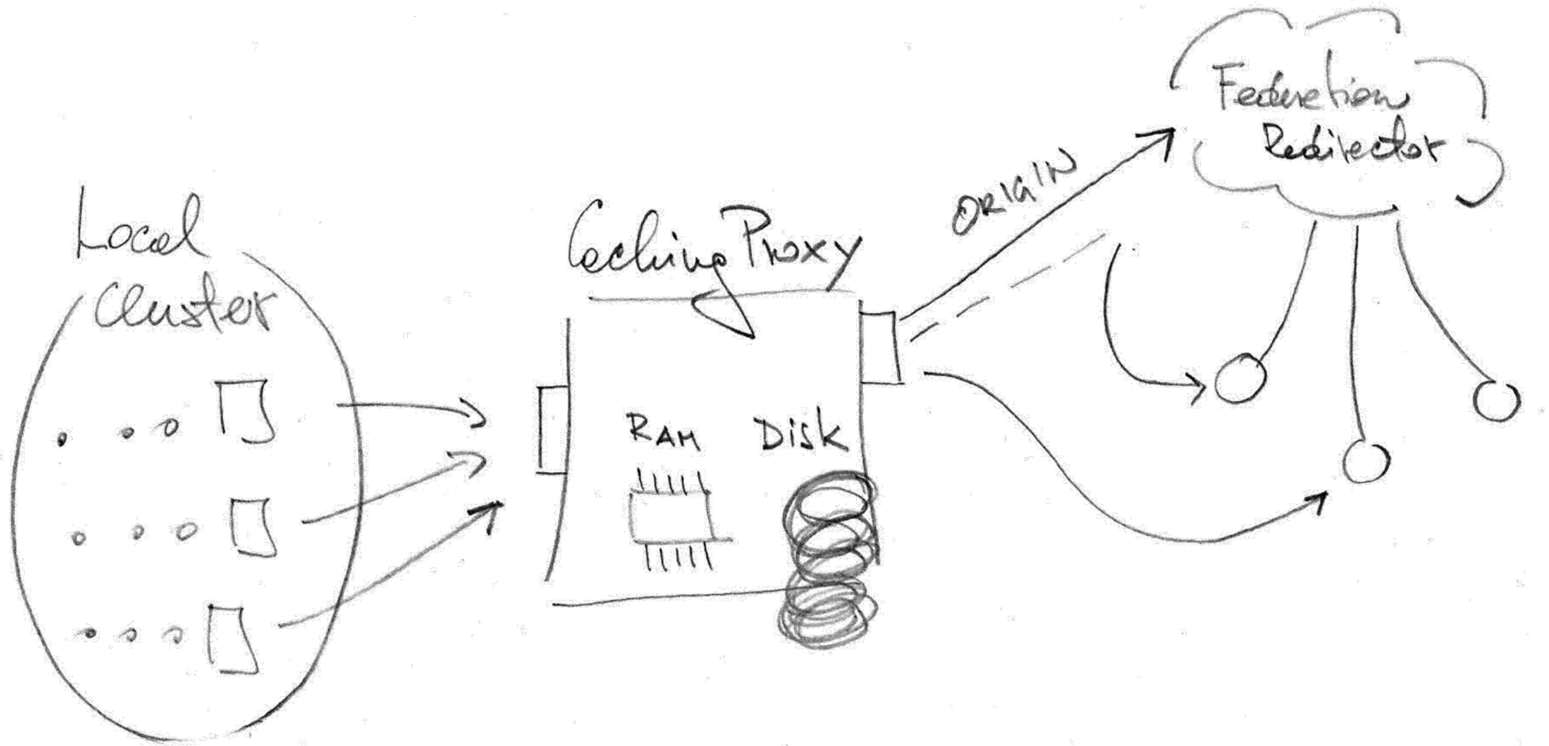
Alja & Matevž Tadel, UCSD

Talk outline

- High level description of operations and components
- Difference between V1 and V2
- Configuration examples
- Existing deployments
- Planned / possible extensions and improvements

HIGH LEVEL DESCRIPTION OF THE PROXY FILE CACHE (PFC)

XRootd Caching Proxy Concept



- Provides disk-based cache for WAN access:
  - performs read-ahead to reduce latency;
  - avoids WAN access on subsequent access.
- Uses fixed-size blocks, between *32 kB* and *4 MB*

Caching Proxy Motivation

- Remove / reduce latencies associated with remote data access
    - Depends on application read rate and cache reuse
- Reduction of WAN usage
    - Requires cache reuse
- Just-in-time Data Placement
    - In testing for CMS computing model
- Data Distribution from sources with limited bandwidth
    - Opportunistic usage of HEP resources by non-HEP VOs

Standard Operations

1. **Open**
   a) *<u>Cold cache:</u>* remote open through Federation
   b) *<u>Warm cache:</u>* opens file on local disk
      Note: remote open is only initiated if/when a requested block is not available in the cache.
2. **Read, Vector Read**
   a) If in RAM / disk → serve from RAM / disk
   b) Otherwise request data from the remote and
      1. serve it to the client
      2. write it to disk via write queue (this way data remains in RAM until written to disk)

All read requests are padded to align with cache blocks.

Authentication

Local authentication

Any method supported by XRootd can be used.

Or none … but this can expose federation data.

Remote authentication

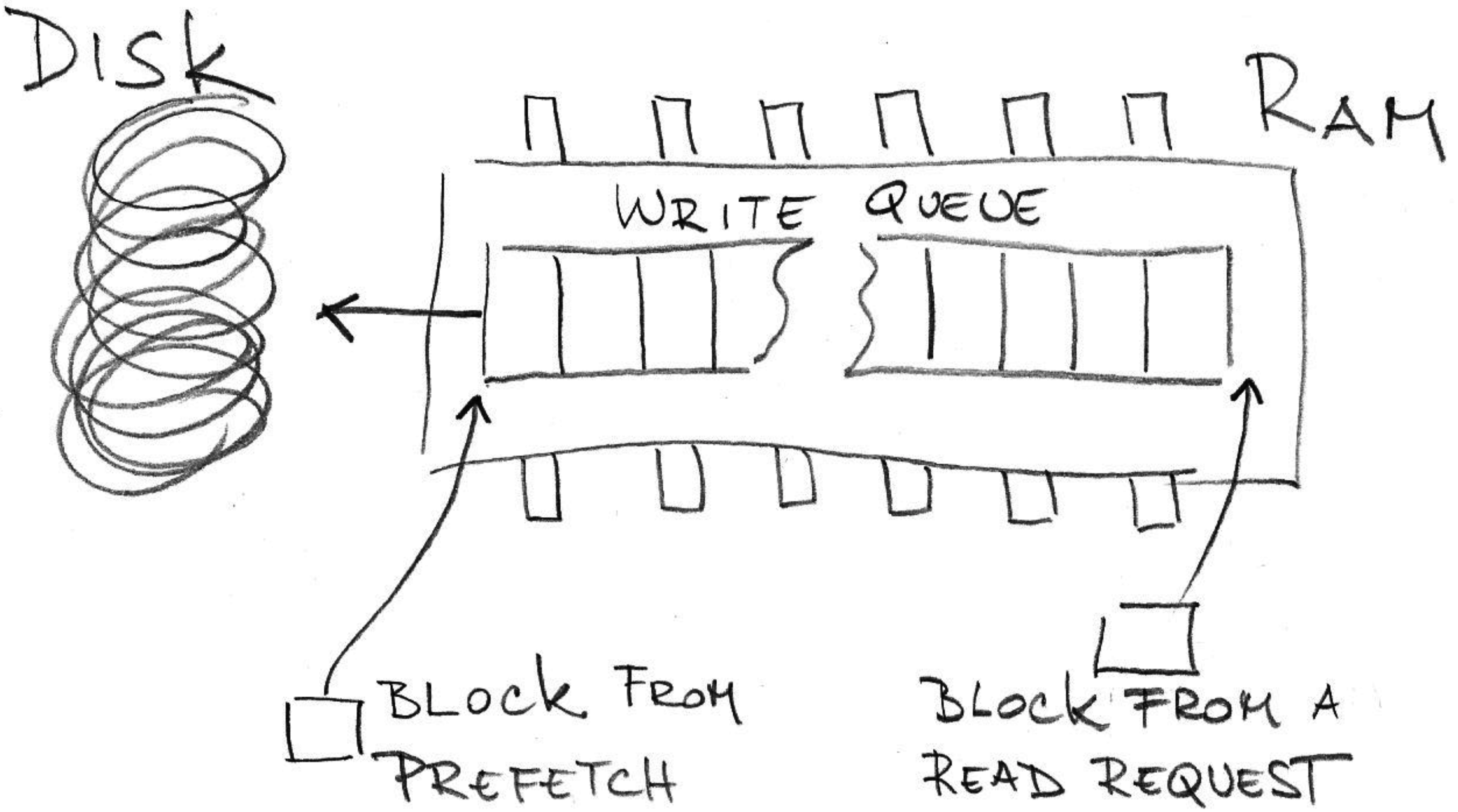
Caching Proxy needs to be able to authenticate itself to the Federation.

- If GSI is used, proxy must have a valid service certificate registered in VO's VOMS

Caching proxy concepts

- ***Cache info file:*** stores details about already downloaded blocks and all local accesses.
- ***Prefetching:*** proxy can issue advance read requests to reduce RTT latency (optional).
- ***Decision plugin:*** allows users to configure which parts of namespace are to be cached.
- ***Cache purging:*** high/low water mark algorithm used to start/stop purging.

Concepts – Write Queue



Write Queue, cont'd.

- Majority of caching proxy RAM usage is the write queue – configurable via the *pfc.ram* directive.
- Prefetched buffers are put to the beginning of the write queue

  → assume they will be needed at a later time so RAM should be vacated as soon as possible.
- Buffers obtained to serve outstanding read requests are put to the end of the write queue

  → assume they will be needed to serve future read requests so they should be kept in RAM as long as possible.
- In future we will provide knobs to tune this behavior.

Sparse files, Virtual FS

- When prefetching is disabled, only blocks actually requested by clients are fetched and written to disk.
    - This leads to sparse files – less disk space is used than the actual file size
    - Purging algorithm handles this properly.
- People often ask if PFC does RAM buffering for read requests that are served from local disk.
    - It does not. We rely on VFS to this for us.
    - Note: also buffers written to disk will remain in memory (shown as *cached* by free / top) as long as RAM is available.
    - Generally recommendation: keep write queue size well below 80% of available RAM – let VFS do its thing, too.

Caching Proxy Overload

Usually disk can not process all write requests and write queue fills up.

- Write queue full: read requests for data not present in the cache are handled as normal remote read calls – no extra buffers are allocated and data is NOT cached.
- Note: Caching proxy will always serve read requests for which it has data on disk from the disk!
- Note: Prefetching is suspended when write queue is 80% full.

- Reducing overload impact:
  - Current tests show using individual disks (binding them via *oss.space*) outperforms RAID-0 by 20-30%.
  - Reduce write affinity of disk I/O scheduler (~10% effect)
  - Provide sufficient number of disks
    - ~20 SATA disks per 10 Gbps.

HDFS fallback / healing mode

- HDFS stores a file in a set of blocks, typically 64 or 128 MB each.
- The blocks are spread around several disks and machines
  - If one machine / disk is lost, the file becomes unreadable, even though majority of its blocks are still available on site
- Solution: pfc.hdfs mode:
  - Proxy separates local files so that they match HDFS blocks
  - Those can persist until storage is fixed … or they can be re-injected into the HDFS cluster.
- Requires a way for HDFS reading functions to fall back to reading through the proxy cache
  - hdfs-xrootd-fallback – HDFS extension developed at UCSD
  - Available through OSG HDFS distribution

DIFFERENCE BETWEEN PFC V1 AND V2

V1 deficiencies

- V1 was designed for a relatively slow read rates and large block sizes
    - prefetching was expected to always provide required data ahead of time
- V1 had several annoying deficiencies
    - resource management was file / connection based
        - no way to control global resource consumption
        - required one thread per connection for prefetching
    - no support for async reads
    - partial support for vector reads
- V2 is practically complete reimplementation

Main improvements in V2

- Always issue async read requests
    - async callbacks happen in Xrd worker threads
- Async read support added to Xrd POSIX / proxy layer
    - so also multiple requests from cache client get processed in parallel
    - e.g. xrdcp issues 4 read requests in parallel
- Common memory pool for cache block buffers
    - allows better resource division among active file reads and prefetching
- Supports deferred opening of remote connections
    - as long as requested data is in cache, remote is not contacted
- Data and cinfo file writes are synced to assure there is no data corruption
- Better cinfo file format, includes checksums and open/close times
- Full support for vector reads, i.e., data returned by vector reads is also written to disk
    - It makes sense to use smaller block size in this case, about 2x average vector read chunk

CONFIGURATION EXAMPLES

Configuring a standalone Caching Proxy Server

- It's a standard xrootd daemon
  - Configure arguments / instances the standard way
- Few directives needed in configuration:

• tell xroot it is a proxy	oss.fslib
• tell proxy it is a file-caching proxy	pss.cachelib
• tell it where remote data is	pss.origin
• tell PFC how much RAM to use	pfc.ram

- RHEL 6 – Use tc- or je-malloc!
  - Buffer memory not released properly in glibc < 2.15
  - See comments at the end of /etc/sysconfig/xrootd

Minimal standalone server configuration

```
all.export /store

ofs.osslib   libXrdPss.so
pss.cachelib libXrdFileCache.so

pss.origin   cmsxrootd.fnal.gov:1094

oss.localroot /data/xrd-cache
pfc.ram       32g
pfc.blocksize 512k
pfc.prefetch  8
```

PFC Configuration Directives

```
pss.cachelib path [libopts]  # required directive
pfc.ram      size[m|g]       # required directive

pfc.blocksize   bytes[k|m]
pfc.decisionlib path [libopts]
pfc.diskusage fracLow fracHigh


pfc.prefetch numPrefetchingBlocksPerFile


pfc.hdfsmode [hdfsbsize bytes [k|m]]
pfc.user      username


pfc.trace <none|error|warning|info|debug|dump>


pfc.spaces data metadata
```

Using multiple disks

- Using oss.space directive one can tell the cache to use all the specified disks / partitions for storage.
  - It turns out such arrangement can outperform RAID 0 when large number of files are accessed concurrently (20-30%).
  - 20 SATA disks needed to saturate 10 Gbps under CMS analysis load.
- oss.localroot still needs to be specified
  - It will contain the actual hierarchy of stored files
  - Files there are symbolic links into one of the given "data" directories.

```
oss.localroot /xcache-root

oss.space public /data1/xcache
oss.space public /data2/xcache
oss.space public /data3/xcache
oss.space public /data4/xcache
```

Separating data and metadata

- Theory: operations on cinfo files interfere with data read/write operations.
  - Provide a way to separate cinfo (meta-data) and data files
  - Can put meta-data on SSD and/or RAID 1 device
  - Note … when cache is full, all cinfo files are scanned every 5 minutes to determine what files to purge.
- Works well also when a single disk fails – we only lose that part of data! This is further argument against RAID-0.
  - Without the separation, cinfo / data files are scattered over all disks

```
oss.localroot /xcache-root

oss.space data /data1/xcache
oss.space data /data2/xcache
…
oss.space meta /xcache-meta
pfc.spaces data meta
```

Caching Proxy Cluster

- Caching Proxy operates in environments where it is expected to fully saturate network AND Disk I/O.
  - Obviously this can only get you that far …
- Solution: *Cluster of Caching Proxies!*
  - Allows horizontal scaling to be able to serve any number of clients.
  - Uses standard XRootd clustering:
    - Redirector + a set of Caching Proxy Servers
    - Configuration sample provided in the documentation

Configuring Caching Proxy Cluster

```
all.manager redirector:1213

# Configuration is different for the redirector, the server cmsd, and for the server xrootd.

if redirector

# stage - if the file isn’t found, redirect to a server with enough free space
all.export /data stage r/o
all.role manager

else if exec cmsd

# stage - this tells manager cmsd we can pull files from the origin.
all.export /data stage r/o
all.role server

# The cmsd uses the standard oss plug-in to locate files in the cache.
# oss.localroot directive should be the same as for the server.
oss.localroot /pfc-cache

else # PFC xrootd server

all.export /data rw  # The server needs to write to disk, stage not relevant
all.role server

# standard server config follows …

fi
```

Some low-level PSS Options

- pss.setopt ParallelEvtLoop 10 [3]
    - Number of polling threads – receivers of remote data
    - Should not be a problem anymore (used to be 1 only)
- pss.setopt.RequestTimeout 120 [1800]
    - We saw cases where failing to open a remote file took forever
    - Very bad for HDFS fallback – let job fail as soon as possible
- Number of connections from a proxy can be significant
    - Defaults for how long XrdCl keeps the connections are too large for a proxy:
        - pss.setopt DataServerConn_ttl 300
        - pss.setopt RedirectorConn_ttl 150

EXISTING DEPLOYMENTS

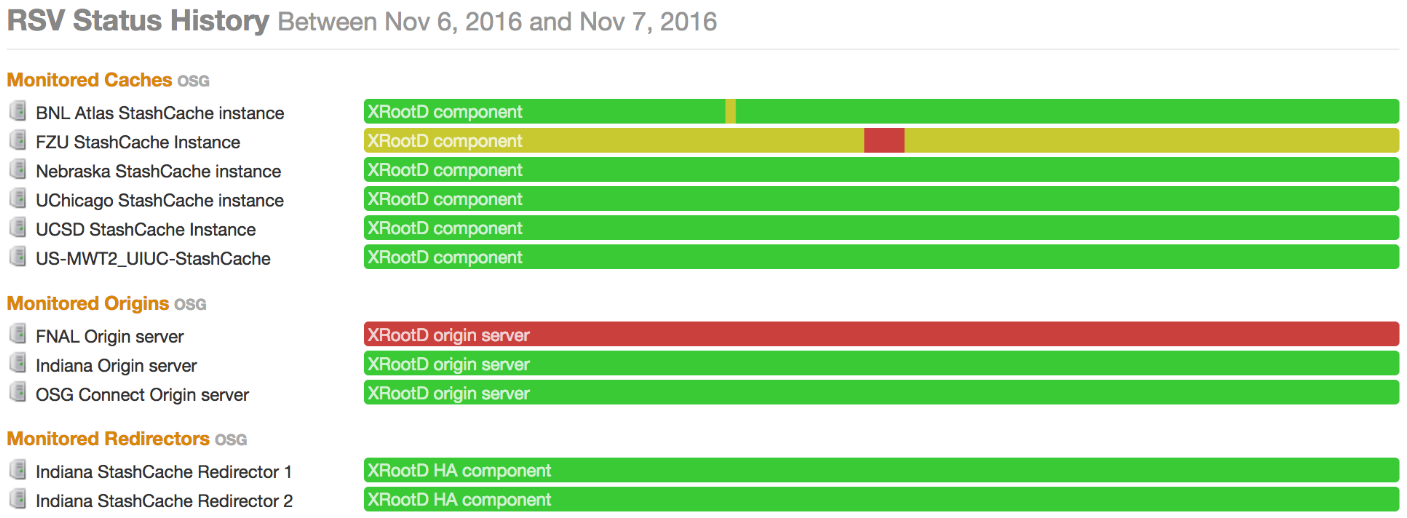
CHRONOLOGICAL ORDER

UCSD HDFS Healing Proxy

- UCSD T2 uses HDFS with replication 2
- For non-custodial data the replication is set to 1
    - HDFS extension put in place to fallback to caching proxy when local read fails
    - Only missing blocks are fetched and stored on the proxy.
    - HDFS healer, reinjecting the missing blocks into HDFS runs daily, going over the list of existing blocks in the cache
- In production since April 2014
- Switched to V2 in May 2016
- At about the same time we introduced preemptive healing:
    - For important datasets scan and fsck the directories.
    - Force the cache to fetch the missing blocks.
    - Healer then reinjects them on the next day.

OSG StashCache

- OSG data distribution service for smaller VOs
    - Stacks XRootd federation, caching proxy, and CVMFS
- In production since early 2015
    - UCSD and UNL use V2 since June
- Largest user is NOvA - access to flux files



RSV Status History Between Nov 6, 2016 and Nov 7, 2016

Monitored Caches OSG

BNL Atlas StashCache instance	XRootD component
FZU StashCache Instance	XRootD component
Nebraska StashCache instance	XRootD component
UChicago StashCache instance	XRootD component
UCSD StashCache Instance	XRootD component
US-MWT2_UIUC-StashCache	XRootD component

Monitored Origins OSG

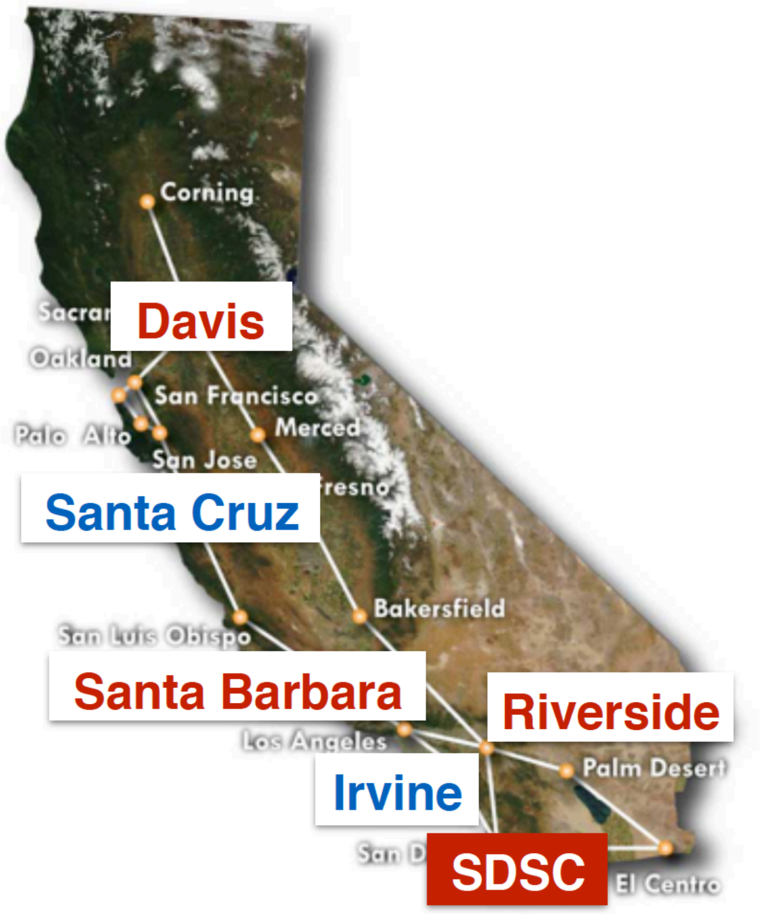
FNAL Origin server	XRootD origin server
Indiana Origin server	XRootD origin server
OSG Connect Origin server	XRootD origin server

Monitored Redirectors OSG

Indiana StashCache Redirector 1	XRootD HA component
Indiana StashCache Redirector 2	XRootD HA component

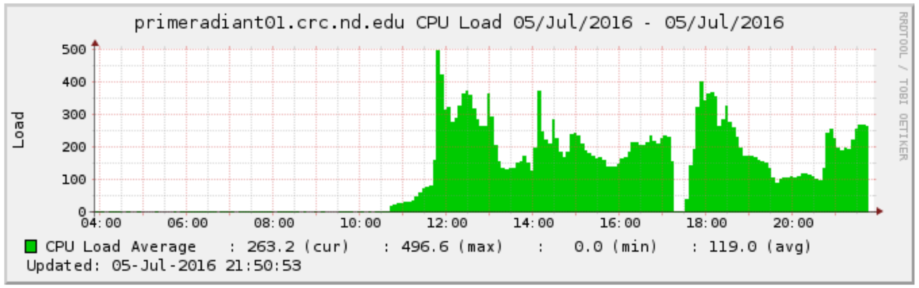
LHC@UC Caches

- Participating UCs:
  🟦 ATLAS 🟥 CMS
- Science DMZs interconnected over 100gbps Pacific Research Platform network
- Resources currently provided:
  - Each UC
  - 50k core Comet cluster at SDSC
- Eventually:
  - Any other non-UC participating PRP site
  - Any OSG site beyond PRP
  - Other NSF XSEDE and DOE super computing centers
  - Cloud resources

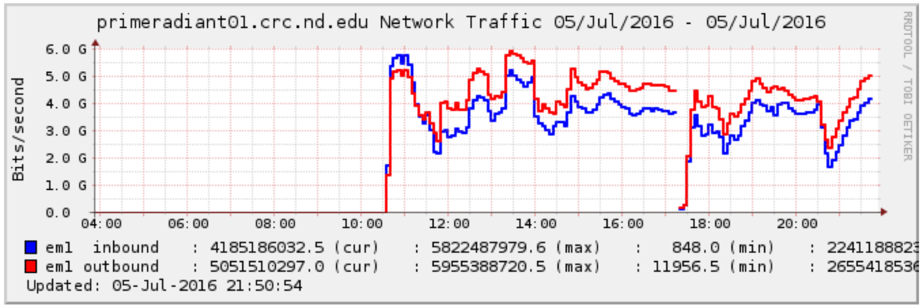


Notre Dame University

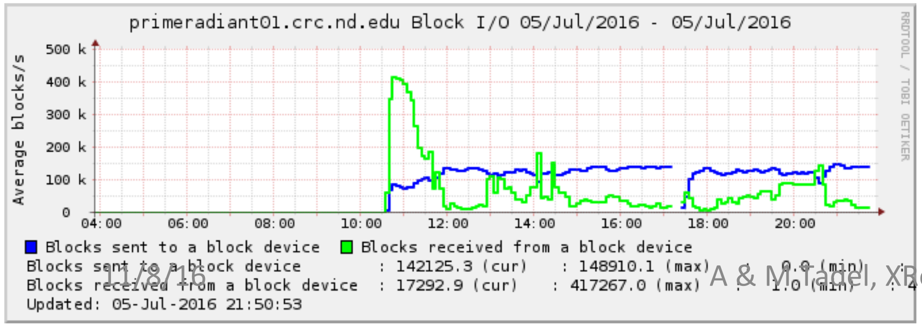
- CMS T3 site
- Campus clusters have 10k opportunistic cores, 100 Gbps WAN connection
- In May 2016 they deployed a 2-server proxy cluster
  - Used V2 from the start … they helped us find and fix several issues that arose under heavy load.
  - I hereby bestow on them the Golden Disk of XRoot Valor for their help!



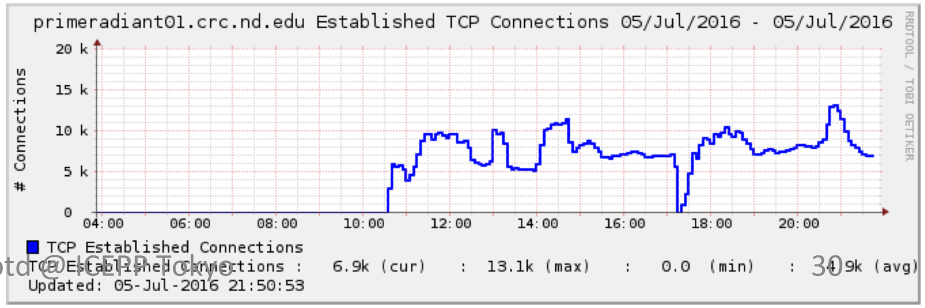
```
🟩 CPU Load Average   : 263.2 (cur)   : 496.6 (max)   :   0.0 (min)   : 119.0 (avg)
Updated: 05-Jul-2016 21:50:53
```



```
🟦 em1  inbound    : 4185186032.5 (cur)    : 5822487979.6 (max)    :     848.0 (min)    : 2241188823
🟥 em1 outbound    : 5051510297.0 (cur)    : 5955388720.5 (max)    : 11956.5 (min)    : 2655418536
Updated: 05-Jul-2016 21:50:54
```



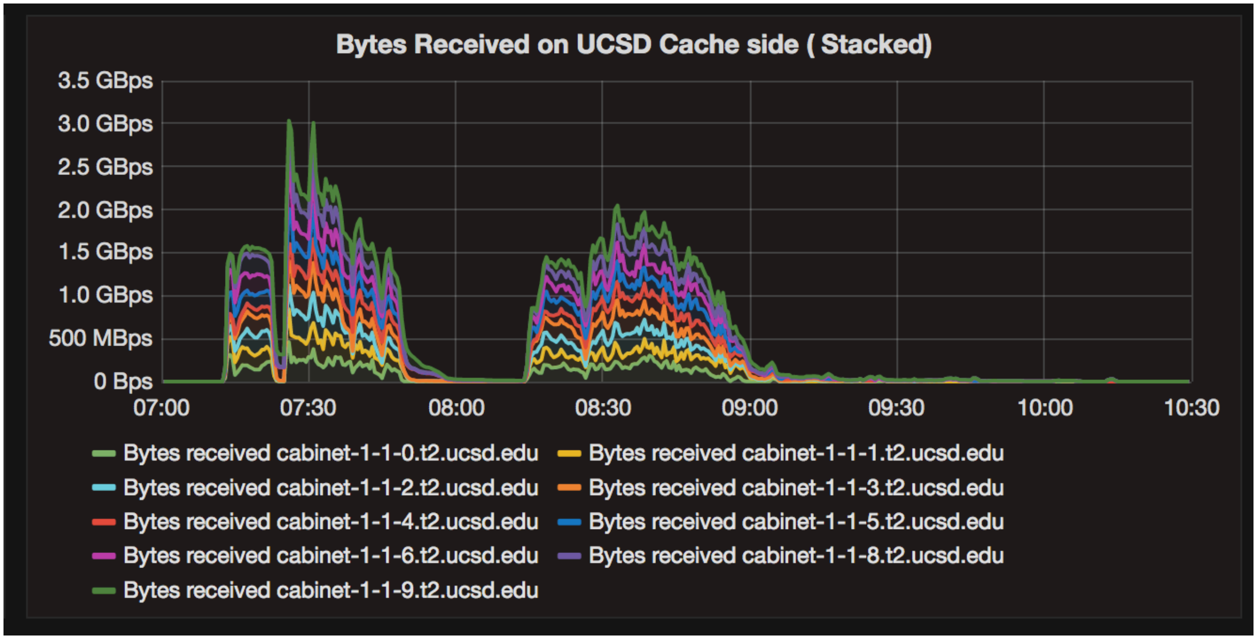
```
🟦 Blocks sent to a block device  🟩 Blocks received from a block device
Blocks sent to a block device         : 142125.3 (cur)    : 148910.1 (max)    :   0.0 (min)    :
Blocks received from a block device   : 17292.9 (cur)     : 417267.0 (max)    :   1.0 (min)    : 4
Updated: 05-Jul-2016 21:50:53
```

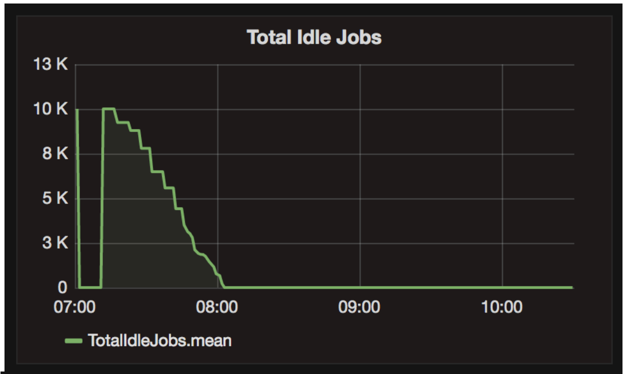


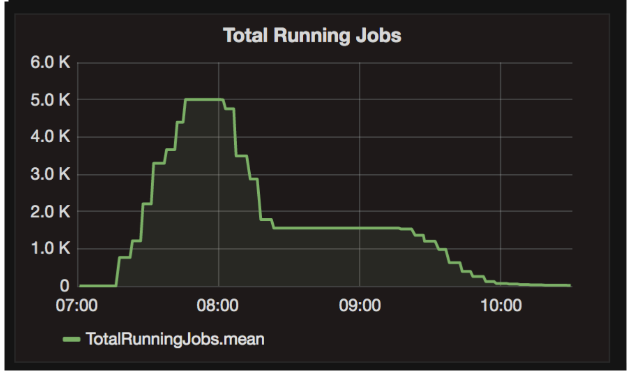
```
🟦 TCP Established Connections
TCP Established Connections :   6.9k (cur)    :  13.1k (max)    :   0.0  (min)    :  4.9k (avg)
Updated: 05-Jul-2016 21:50:53
```

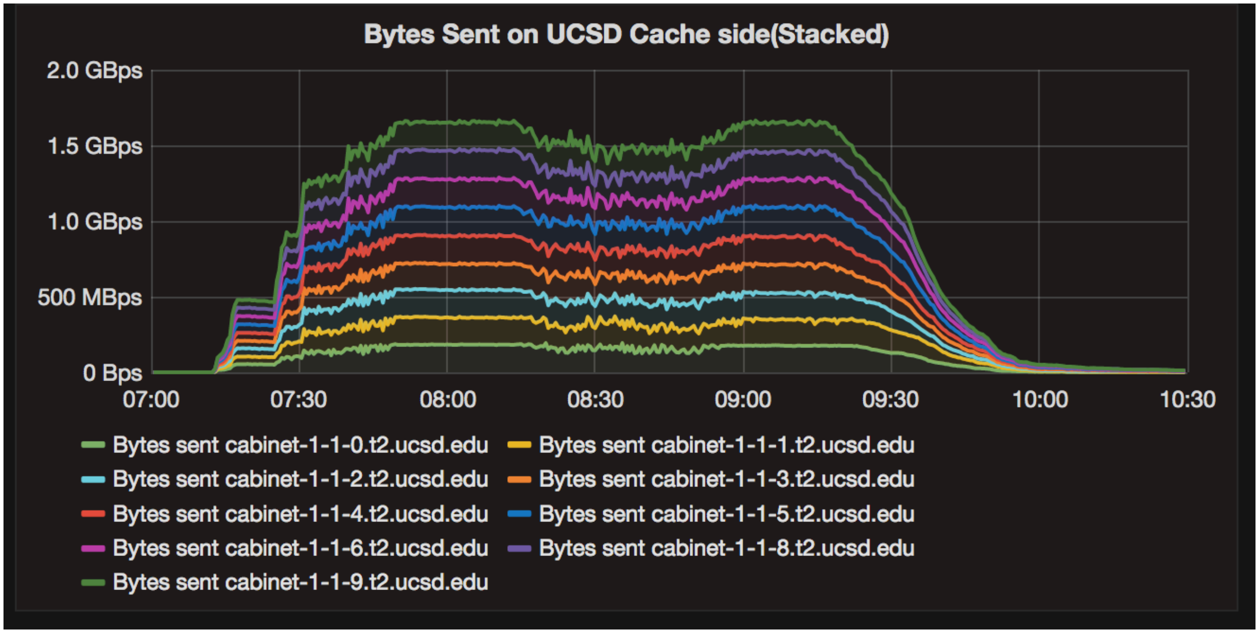
CMS Caltech-UCSD Shared Analysis Cache Pilot

- CMS project to evaluate Cache usage for analysis
    - Enough space to hold latest reprocessing of MiniAODs
    - Add special rule into Crab3 to assume data is there
        - We do not want to register it into PhEDEx
    - Enough capacity to feed 20k jobs (UCSD, Caltech, Comet!)
    - Expect cache turn-over on order of months
- Hardware
    - UCSD: 10 x 12 2TB SATA disks, 10 Gbps
    - Caltech: 2 X 40(?) TB (one SAS, other SSD), 10 Gbps
    - UCSD T2 – Caltech should be 80 Gbps @ 4 ms
- We'll run demo of data transfers at SC'16 next week!
- After that: integration / testing until the end of the year. 3 Month production run Jan – Mar 2017.

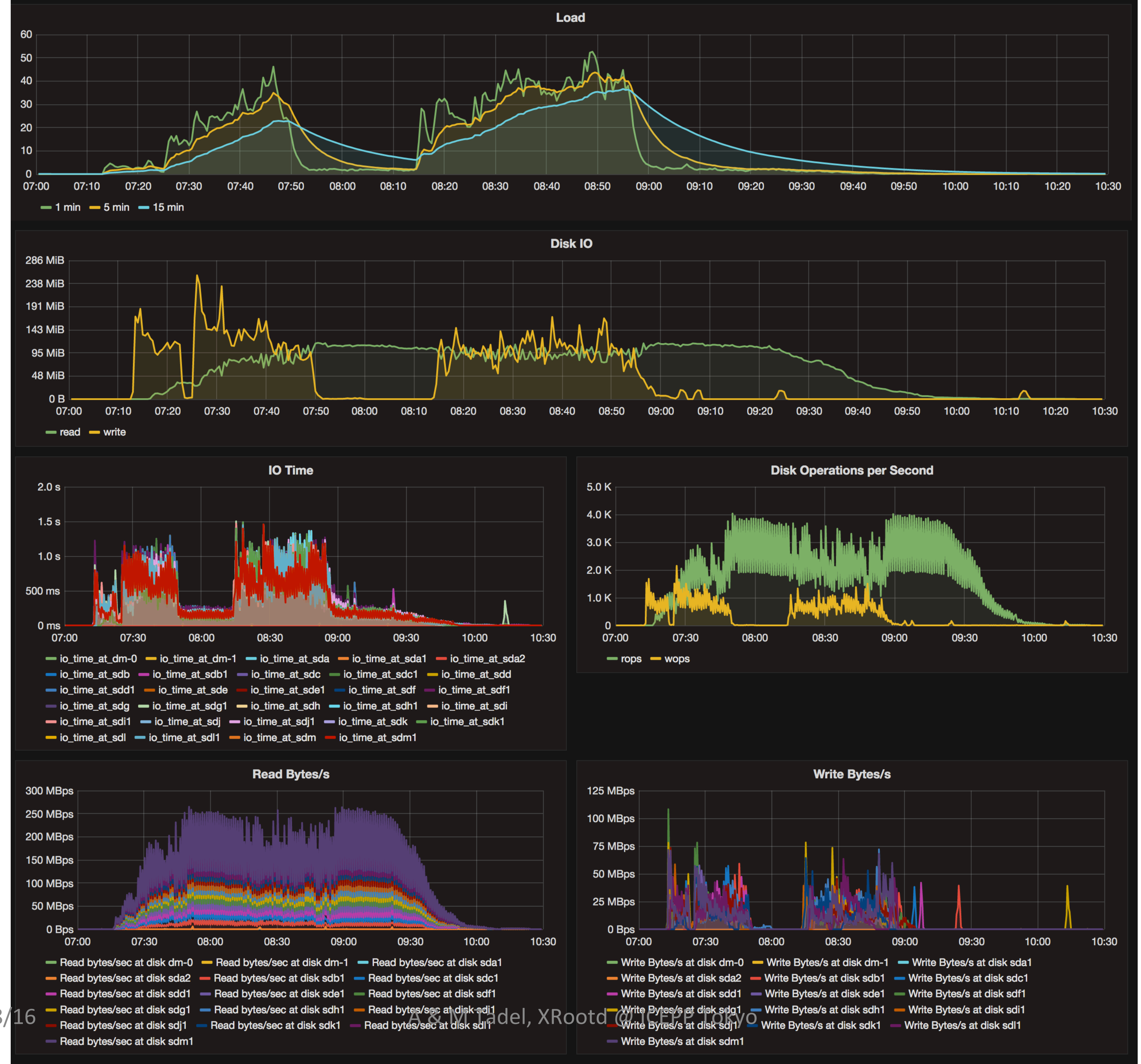








- 1 Job: read 4 GB @ 1 MB/s from two files, consecutively.
- Apparently only 1550 jobs survived
- Note: prefetching.
- Note: read peak when secondset of files gets opened.


Load
1 min
5 min
15 min
Disk IO
read
write
IO Time
io_time_at_dm-0
io_time_at_dm-1
io_time_at_sda
io_time_at_sda1
io_time_at_sda2
io_time_at_sdb
io_time_at_sdb1
io_time_at_sdc
io_time_at_sdc1
io_time_at_sdd
io_time_at_sdd1
io_time_at_sde
io_time_at_sde1
io_time_at_sdf
io_time_at_sdf1
io_time_at_sdg
io_time_at_sdg1
io_time_at_sdh
io_time_at_sdh1
io_time_at_sdi
io_time_at_sdi1
io_time_at_sdj
io_time_at_sdj1
io_time_at_sdk
io_time_at_sdk1
io_time_at_sdl
io_time_at_sdl1
io_time_at_sdm
io_time_at_sdm1
Disk Operations per Second
rops
wops
Read Bytes/s
Read bytes/sec at disk dm-0
Read bytes/sec at disk dm-1
Read bytes/sec at disk sda1
Read bytes/sec at disk sda2
Read bytes/sec at disk sdb1
Read bytes/sec at disk sdc1
Read bytes/sec at disk sdd1
Read bytes/sec at disk sde1
Read bytes/sec at disk sdf1
Read bytes/sec at disk sdg1
Read bytes/sec at disk sdh1
Read bytes/sec at disk sdi1
Read bytes/sec at disk sdj1
Read bytes/sec at disk sdk1
Read bytes/sec at disk sdl1
Read bytes/sec at disk sdm1
Write Bytes/s
Write Bytes/s at disk dm-0
Write Bytes/s at disk dm-1
Write Bytes/s at disk sda1
Write Bytes/s at disk sda2
Write Bytes/s at disk sdb1
Write Bytes/s at disk sdc1
Write Bytes/s at disk sdd1
Write Bytes/s at disk sde1
Write Bytes/s at disk sdf1
Write Bytes/s at disk sdg1
Write Bytes/s at disk sdh1
Write Bytes/s at disk sdi1
Write Bytes/s at disk sdj1
Write Bytes/s at disk sdk1
Write Bytes/s at disk sdl1
Write Bytes/s at disk sdm1

PLANNED & POSSIBLE IMPROVEMENTS AND EXTENSIONS

Load shedding

- Overloaded caching proxy becomes a bottleneck, same as for disk servers
- But we have another option here:
  – Redirect new clients directly to the origin!
    - Existing clients get good performance from the cache
    - Note: when write queue is full, cache does pass-through anyway.
- We can also use this from the decision plugin:
  – Redirect for non-matched paths
    - No need to go through the proxy, either

Cache Purge Algorithm

- Current: high/low watermark; purge files last accessed furthest in the past
- Not necessarily good – provide XrdPfcPurge plugin
    - Let people use their own foot gun
- It could do:
    - Also consider how much data was actually transferred and number of accesses (not only time of last access).
    - Protect certain parts of namespace from deletion.
    - Protect cache as-is … redirect guys trying to open files not existing in the cahce
    - Time-to-live – delete anything older than some date

Write Queue Improvements

- Current: prefetched blocks to front, blocks serving reads to the back.
- Could do:
  - If block is fully consumed by a read, put it to the front of the queue (will not need it in near future)
- Prefetching is stopped when write queue is 80% full
  - This could be a configuration parameter.
  - And be high / low water mark based to simplify state checks.
- Data / cinfo syncing happens every 100 blocks
  - Should also be a config parameter (or size based)

Prefetching Algorithm

- Current: not really smart, just gets the first non-cached block.
  - This is ok if whole file will be read and read rate is low.
- Could do:
  - Get first and last X MB of file right on open.
  - Prefetch from last read position onwards.
  - Use plugin to predict next read position:
    - e.g., could use root to make branch / basket directory
- With smart prefetching it would make sense to put blocks at the end of write queue.

Shared Disk space

- Assume you want to provide cache for several VOs on the same server
    - If VOs require client authentication there is no other way then running several server instances
- We could provide common disk usage / purge limits that assure each VO gets its portion of cache space
    - Could allow one VO to use additional space as long as total usage remains below given setting.
    - After that the VO that uses extra space gets purged first.

CONCLUSION

Conclusion

- Caching proxy V2 is taking off …
- The plan was to include it in 4.5.0
    - Delayed until 4.6.0 due to an issue with semaphores in XrdCl
    - Caching proxy puts XrdCl to the limits of sanity
- Welcome to try it out:
    - http://xrd-cache-1.t2.ucsd.edu/RPMS
        - see ChangeLog.txt in the above directory
    - but let me know … I only build what is actually used