File Caching Proxy Server details & status

Content

- Design
- Configuration
- Tests
- HDFS healing

Project history

- The idea for the file cacheing proxy began 2011 as part of AAA project with an attempt to
  - reduce latency and serve cases where a remote file is read multiple times (e.g. mixing, pileup, analysis)
  - T3 user analysis
- Later we realized it could also be used for:
  - dynamical data placement
  - reduce replication of non-custodial data on sites
    - fetch missing chunks from federation as/when needed
- First version of file caching proxy available in xrootd 4.0
- Currently used at UCSD for HDFS healing, ongoing scaling performance test for just in time data placement

Basic design I – caching proxy

– By the default proxy server forwards open and other posix file like operations to a remote client.

– It is possible to propagate posix file operations through the caching layer. Activated by loading plugin which holds an implementation of XrdOucCache.

– Default cache implementation is RAM based. Implemented in class XrdOucCacheRAM.

– See Andy's talk for more information about xrootd proxy.

Basic design II – file caching workflow

–Read requests coming into XrdOucCacheIO are rounded to a block size (64 kB – 8 MB).

- Data is returned to the client.
- Block is also written to local disk (via a write queue).

–A prefetching thread is started for each open file:

- blocks are downloaded in order and also written to disk
- prefetching continues for as long as the file is open or the whole file is downloaded.

–If the block is already available, data is read from disk.

Configurable parameters

File caching server has configurable parameters to control RAM usage, file purging, change output file system, or set it in HDFS healing mode. None of them are mandatory.

pfc.cachedir *path, /var/tmp/xrootd-file-cache*
pfc.decisionlib *path [libopts], empty*

pfc.blocksize *bytes [k|m], 1MB*

pfc.nramread *num, 1*

pfc.nramprefetch *num, 8*

pfc.diskusage *fracLow fracHigh, 0.9 - 0.95*

pfc.osslib *path* [*libopts*], empty

pfc.filefragmentmode *[filefragmentsize bytes [k|m]], 128MB*

Note: This is xroot 4.1 configuration format which used ‘pfc.’ prefix.

Decision plugin

- By default all incoming files are cached
- Decision plugin – choose which files to cache based on path
- Example in the xrootd repository:
  - class XrdFileCache::AllowDecision which passes any file
    - virtual bool Decide(std::string &, XrdOss &) const {return true; }
- Set in configuration with pfc.decisionlib directive

BlockSize

- All read request are rounded to this value.
- Default size is 1MB (can be changed in configuration).
- Tests and cache simulations for real CMS jobs show that block size has little impact on job performance between 0.5 and 4MB.
- Small block sizes:
  - burden disk with too frequent writes
- Large block sizes:
  - can increase RAM usage
  - increase over-read when only parts of file are read

Handling RAM usage

- Proxy server requires at least one block of RAM for each file. Each block is reused after it is written to disk.
- RAM usage grows with the number of open files:
  - N_read * block_size +
  - N_pref * block_size if prefetching is activated
- This gives equal opportunity to all files.
- If access characteristics of jobs are significantly different, one might desire to allocate blocks to files dynamically.
  - Have a pool of extra blocks for such cases.
  - Extra block allocation could also be handled by a plugin.
  - This would also speed up reading of files under a low load.
  - Allow reasonable RAM usage under heavy load (prefetching would get effectively disabled).

Authentication

- Proxy cache can use any xrootd authentication for incoming connections.
- Identity of proxy process is used to login into remote servers.
  - Is this a problem? Should identity of the client that requested the file be passed through proxy?
- What to do with multiple VOs (open topic)
  - use several caching proxy instances
  - support multiple identities in proxy
  - implement credential forwarding from original client (should be doable for X509 proxies)

Basic Test I. – start a server and copy a file from it

Run xrootd file caching proxy server with configuration :

all.export	*proxy-specific-exports*	
xrd.allow	host *.*domain.edu*	
sec.protocol	*desired-authentication*	
ofs.osslib	***libXrdPss.so***	converts xrootd server to proxy
pss.origin	***x.domain.edu***	location of external redirector
pss.cachelib	***libXrdFileCache.so***	enable file caching mechanism in proxy
pfc.cachedir	***path***	top directory of cachache files (note pfc prefix)

Copy file from the proxy two times and compare speed

```
# xrdcp –f root://genki.physics.ucsd.edu//store/user/alja/data.root .
[xrootd] Total 3959.13 MB     |====================| 100.00 % [2.2 MB/s]
# xrdcp –f root://genki.physics.ucsd.edu//store/user/alja/data.root .
[xrootd] Total 3959.13 MB     |====================| 100.00 % [557.0 MB/s]
```

Basic Test II. – cache inspection

list files in local cache dir

-rw------- 1 xrootd zh 579 Jan 12 14:47 /local-cache/store/user/alja/data.root

-rw------- 1 xrootd zh 579 Jan 12 14:47 /local-cache/store/user/alja/data.root.cinfo

Print info from *.cinfo file: block size, downloaded blocks, and access statistics:

DumpCinfo /local-cache/store/user/alja/data.root.cinfo

Stat version == 0, bufferSize 1048576 nBlocks 3960 nDownlaoded 3960 complete

access 0 >> [Mon Jan 12 14:40:57 2015], bytesDisk=0, bytesRAM=4151446113, bytesMissed=0

access 1 >> [Mon Jan 12 14:47:27 2015], bytesDisk=4151446113, bytesRAM=0, bytesMissed=0

The dump tool needs to be included in xrootd source tree.

Until then it's available from github: alja/xrootd-cache-info

File caching proxy cluster

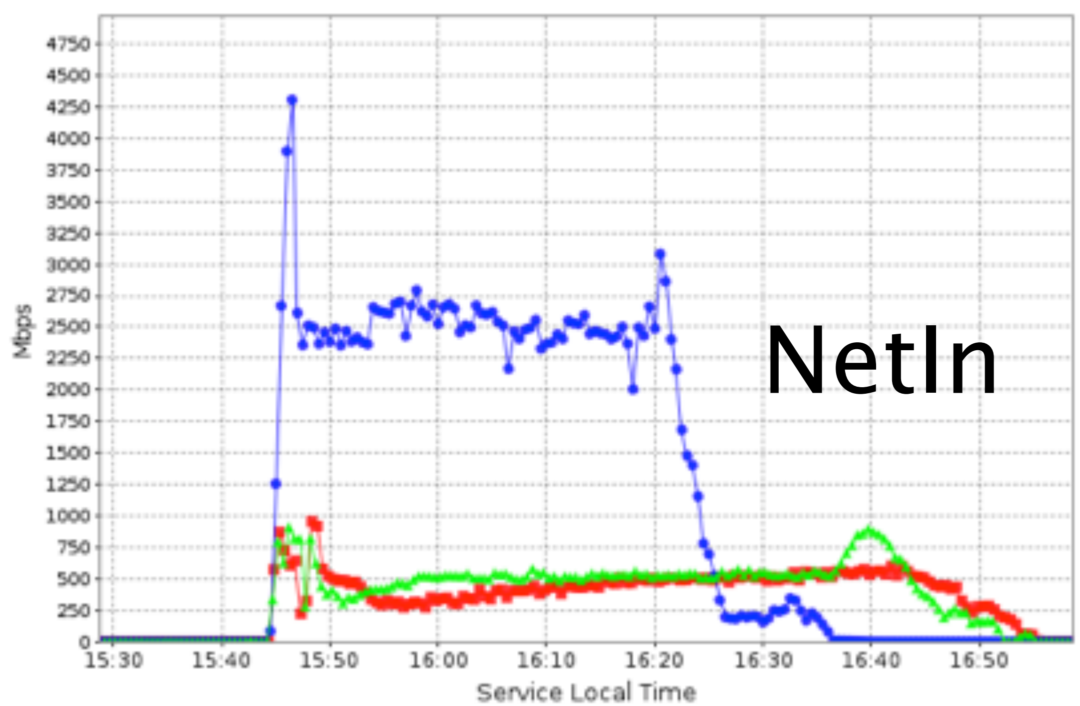
- Intended to distribute heavy disk load among nodes
- Composed of redirector and file caching servers, each one run xrootd and cmsd process

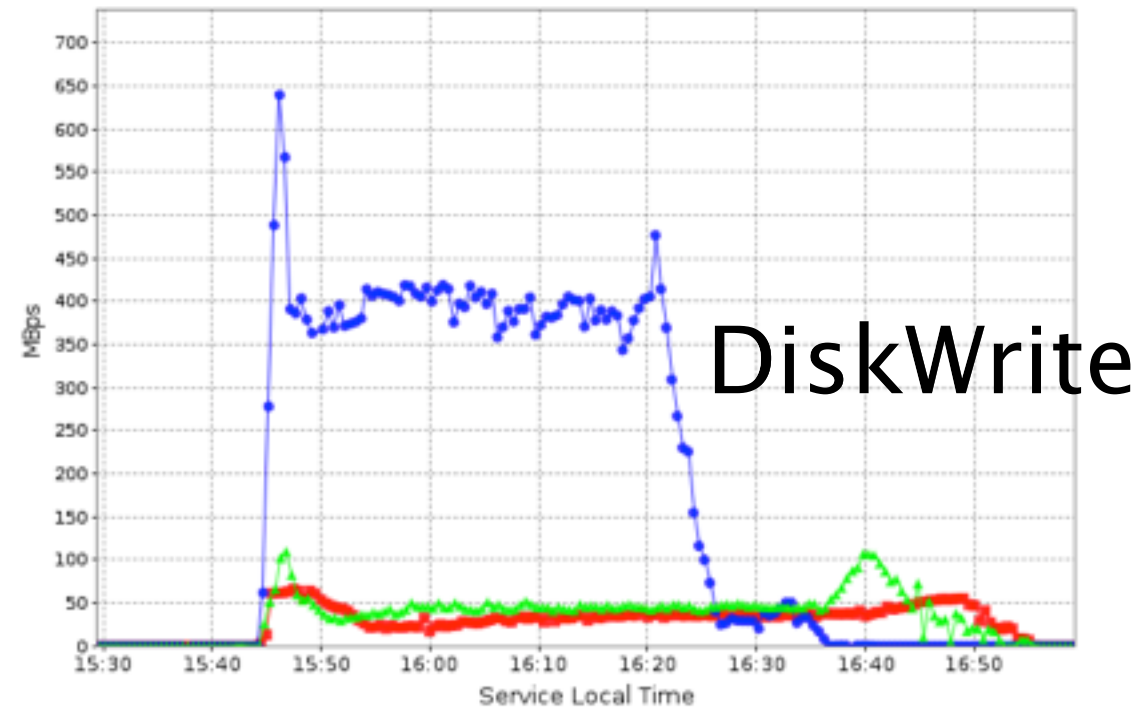
```
all.manager redirector:1213
all.export /data stage r/o
if redirector
all.role manager
else if exec cmsd
all.role server
oss.localroot /pfc-cache
else
all.role server
ofs.osslib   libXrdPss.so
pss.cachelib libFileCache.so
pss.origin someserver.domain.org
pfc.cachedir /pfc-cache
fi
```

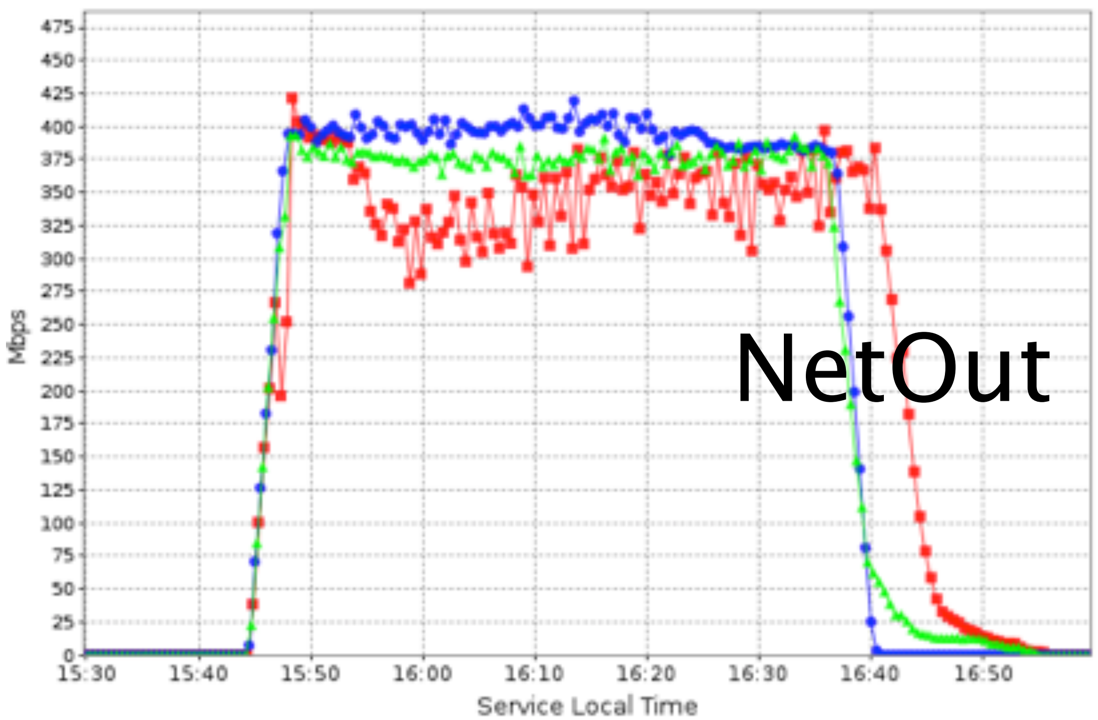
Simulation of CMS jobs on cluster

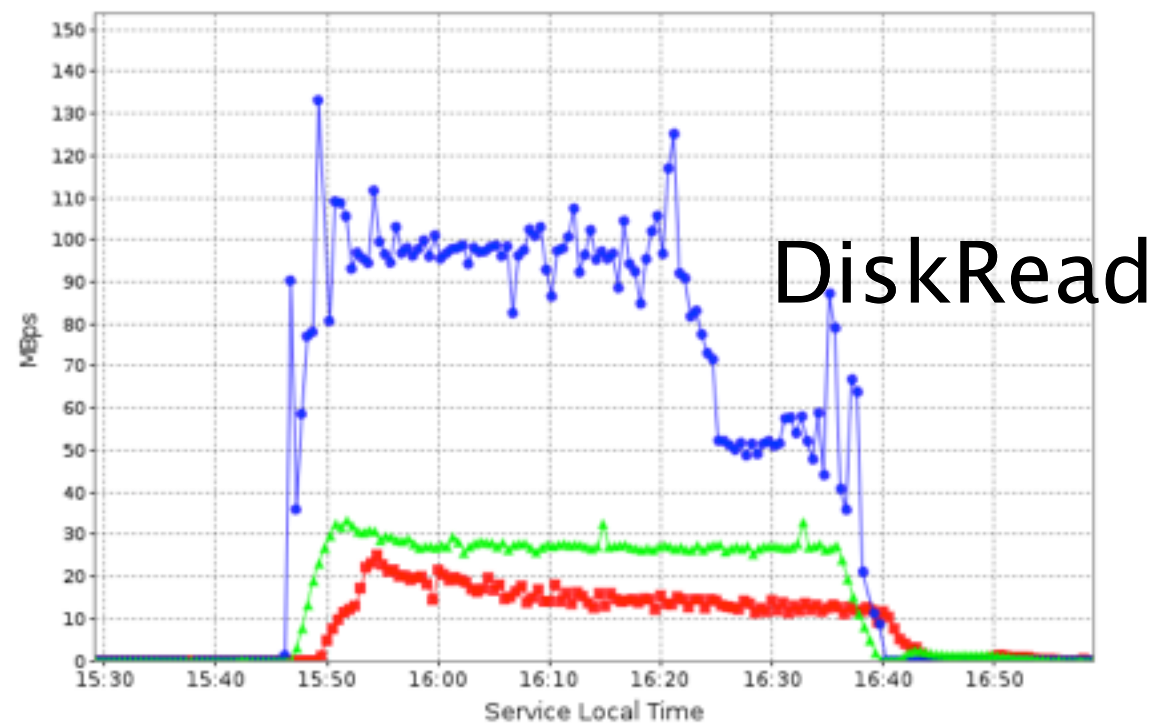
- created a test xrdfragcp to simulate CMS jobs :
  - data rate: reads 2.5 MB every 10 seconds
  - sequential single reads (not realistic, CMS jobs do vector reads in Run1)
- Run 600 jobs on a cluster with 3 inhomogeneous nodes
  - high performance IO node (20Gbps, 20 disk ZFS partition)
  - 2 standard machines (1Gbps, use single disk).
- Plot network / disk I/O rates.
  - **Note:** diskIO is in MBps and netIO in Mbps.

3 Node test, 600 jobs, empty cache



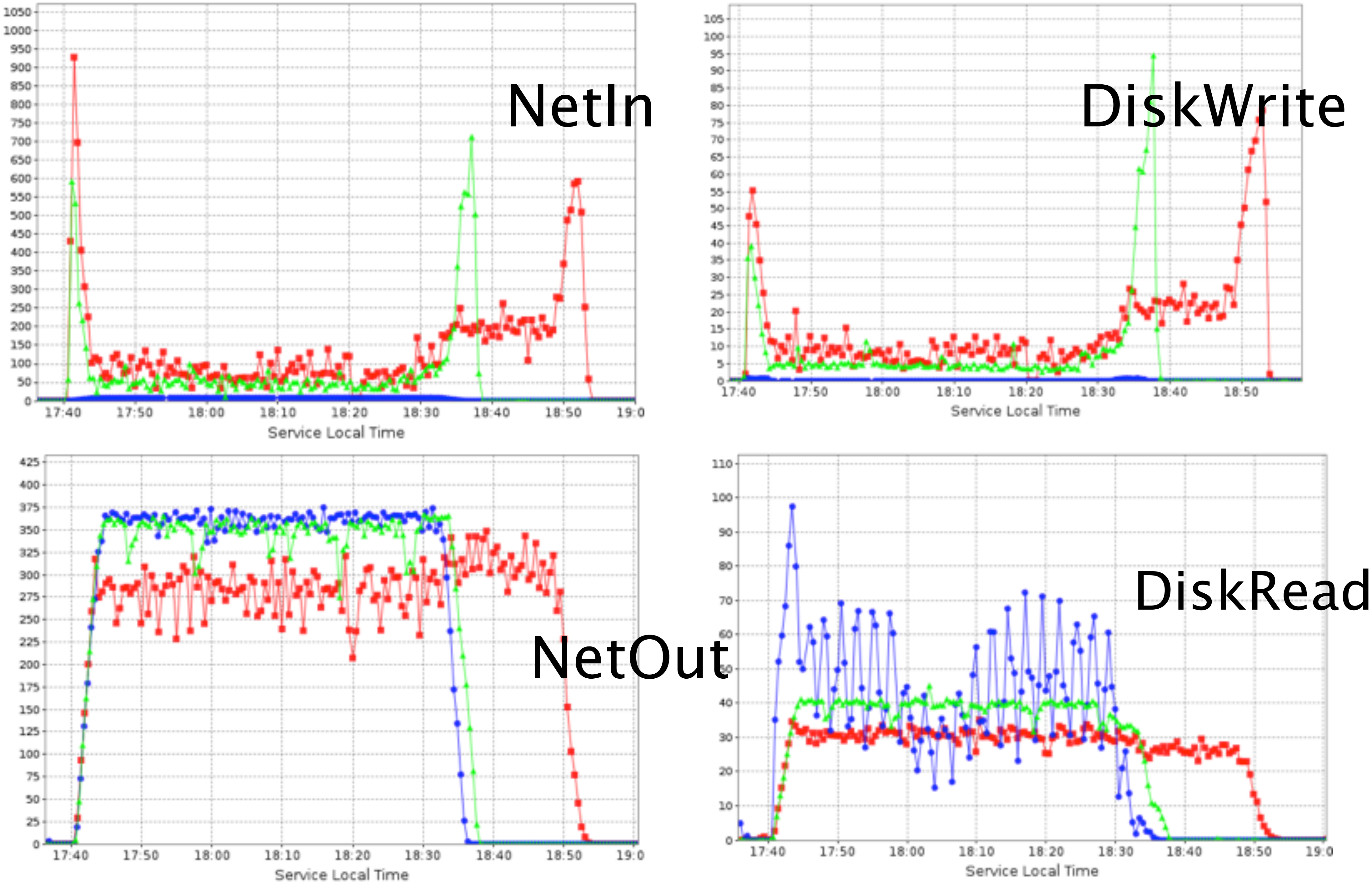






Notes: each got 200 jobs, can see the difference between network traffic and diskIO. The high perf. node finished download of all files while serving data on time.

Re-run the same 600 jobs on the same cluster



Notes: redirector forwarded request to the server with existing file. The standard desktop node needed more time to finish operation, because of more intense disk reads

File caching cluster scaling – conclusion

- Cluster setup is the most easy and successful solution
- High performance IO node can handle heavy load.
  - E.g., Mongo: 1.6 TB/s on 20 ZFS disks
- The third option would be to use multiple disks on the same node. Possible do that with volume manger.
- Need to explore how to do cmsd balancing in inhomogeneous cluster.

HDFS healing with File caching proxy

- Seamless fallback to federation when a corrupt hdfs block gets accessed:
  - exception handler added into HdfsFileStream
  - calls C++/JNI code running xrootd client
  - client connects to given proxy-server to obtain requested data
- Proxy servers file read request and at the same time download a file fragment that corresponds to the corrupted block.
- Downloaded/cache file fragments are injected back to hdfs file system daily.

HDFS healing packaging

- three components
  - proxy server
  - patched hdfs which can run fallback to caching proxy (included in OSG hdfs)
  - scripts that inject healthy blocks back into hdfs (available soon in OSG)

HDFS healing @ UCSD

- Stable running for 9 months on 192 TB of non-custodial Run 1 data.
- Replication set to 1.
- Healed 25.000 blocks (128 MB per block).
  - This only happens if file/block is actually accessed!
- We check checksum before inserting the downloaded blocks. 1% of cached blocks have invalid checksums -- a file had bit switched only on two places in 128M blocks. Later discovered this is caused by errors in network switches.

Conclusion

- File caching server ready for hdfs healing and for low to medium load for dynamic data placement.

  Get in touch with us if you're going to try it!

- We are still testing (vector reads, stability of CMS jobs under high load, cmsd balancing).
- Changes for 4.2 -- new implementation of prefetch to reduce number of threads, better handling of RAM.