XCache - Feature overview, Configuration examples & Development plans

XRootD Workshop @ CC-IN2P3 Lyon

June 11, 2019

Matevž Tadel, UCSD

Overview

Introduction

Features & configuration options

Configuring XCache cluster

Development plan / Conclusion

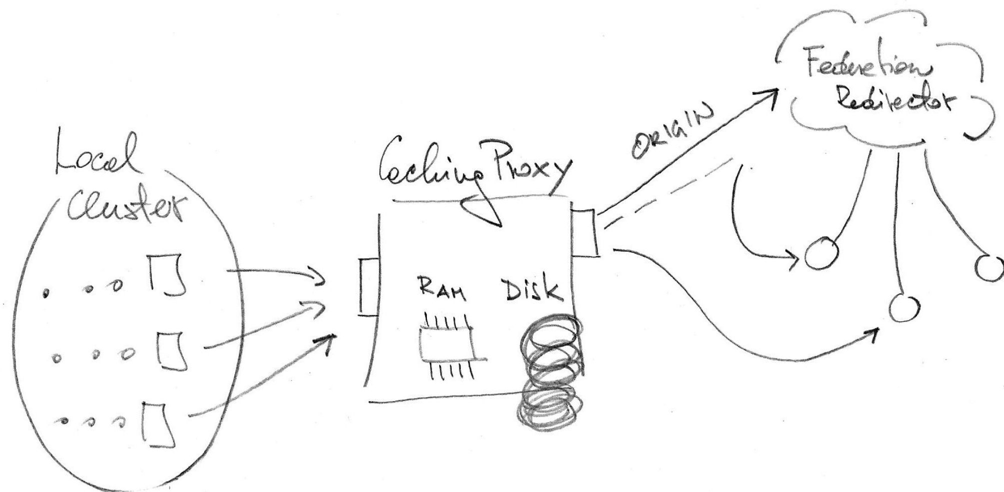
Introduction

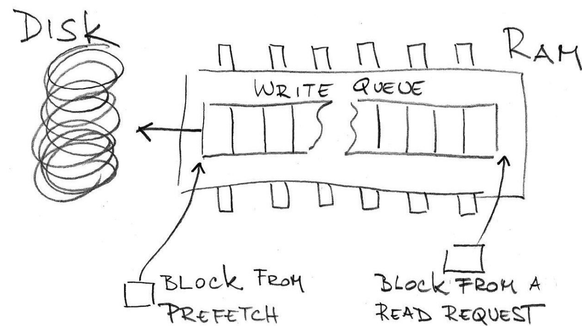
History

- V1 (CHEP 2013)
  - First implementation using "the old client"
  - Hdfs fallback & healing
- V2 (XRootD @ ICEPP Tokyo, Nov. 2016)
  - Reimplementation using "the new client" using async I/O
  - XCache-V2 presentation gives a good overview of XCache
- Now (XRootD @ IN2P3 Lyon, Jun 2019)
  - 2½ years of adiabatic improvements and new ways of using it, e.g.:
    - client-side cache / direct cache access / forwarding mode
  - Used extensively in production (CMS, OSG/StashCache) and in testbeds (ATLAS, PRP, INet2)
    - ??? I actually hope to learn here what is new in this area. :)

One slide overview

- Serve data to local clients:
  - Origin - remote data source (usually data federation)
    - Data read in "blocks"
    - Optional prefetching
  - Store data on local disk via write queue
  - Purge old files as disks get full





- XCache server is a "normal" XRootd server:
  - Authentication / authorization controls
  - LVM / multi-disk support
  - Tracing and monitoring
  - Clustering - Caching Cluster

Features & configuration options

Thou shalt read the Holy Docs!

http://xrootd.org/doc/

Minimal XCache Server configuration

```
# Mandatory directives

all.role      proxy server                  # This is a proxy
all.export    /store                        # Exported namespace

pss.cachelib libXrdFileCache.so             # Request Proxy File Cache / PFC
pss.origin   cmsxrootd.fnal.gov:1094        # Remote data source

oss.localroot /data/xrd-cache               # Where data is stored on local disk

# Frequently used pfc directives (the numbers given are defaults)

pfc.blocksize 1M              {4k,  16M}
pfc.prefetch  10              { 0,  128}
pfc.ram       1G              {1G, 256G}
pfc.diskusage 0.9 0.95    {no limits, can also be given in bytes}
```

pss.origin & remote authentication

- Standard XRootd client (XrdCl) is used to access remote servers
  - If you do manual `xrdcp` from the specified origin, that's mostly what happens in XCache
- If remote sources require authentication, the credentials have to be provided in one of the ways accepted by XrdCl
  - X509:
    - Proxy in `/tmp/x509up_u{user id of xrootd daemon}`
    - Set `X509_USER_PROXY` in daemon startup script
    - The grid proxy needs to be renewed as needed, XrdCl picks up the updated version
    - These days ATLAS and CMS mostly use VOMS membership for XRootd authentication
      - Certificate DN has to be registered in VOMS
      - Same certificate can be used on several cache nodes
    - Note - this is usually NOT the server certificate that is used by local clients / jobs during authentication to the cache.

pfc.blocksize & pfc.prefetch

- XCache downloads data in blocks and stores block status in a bit-vector to keep track of which blocks are available locally.
  - NOTE: When prefetching is disabled, it is common to have sparse files on disk.
- ***`pfc.blocksize`*** sets the size of the block (default 1M)
  - Larger blocks are better for whole-file streaming access
  - Smaller blocks make more sense for sparse vector read type of access:
    - Especially when prefetching is disabled.
    - For ROOT files - ***Thou shalt know your basket sizes and access patterns!***
- ***`pfc.prefetch`*** sets the maximum number of remote block read requests in flight that is allowed to be reached by a prefetching request:
  - Normal reads are not limited by this number (if we need to get 100 blocks to satisfy a read request, all 100 are requested asynchronously)
  - Prefetching is disabled when writequeue is 70% full.

pfc.ram & pfc.writequeue

- **`pfc.ram`** specifies the amount of RAM to be used for outstanding remote read requests:
  - Additional RAM is allocated for local client requests, expect about pfc.ram + 2 GB total usage
  - When RAM is consumed, additional read requests get served in ***direct mode***, serving the request by forwarding the request to the remote as is.
  - ***Calculate:*** $N_{clients} * N_{vread_chunks} * BlockSize$, e.g. 500 * 200 * 0.25M = **25 GB**
    - Again … you really should know your access patterns and expected load!
  - **Beware:** *Some influential people* advertise way too low value for this (1 GB)!
      - This can lead to a lot of direct reads without storing of the data to disk.
- **`pfc.writequeue`** `maxblks nthreads` specifies:
  - number of blocks taken off the write queue in one writer thread iteration {1, 1024; dflt: 16}
  - number of writer threads {1, 64; dflt: 4}
- <u>Dev notes:</u> *(help needed, esp. if you have a monster machine and good testing setup)*
  - we will add support multiple write queues (optionally pinned to LUNs)
    - experiment & prepare recommendations for spinning / solid state disks

pfc.diskusage

```
pfc.diskusage lowWatermark[k|m|g|t] highWatermark[k|m|g|t] # can also be fractions of available space
          [files base[k|m|g|t] nom[k|m|g|t] max[k|m|g|t]]
          [{purgeinterval | sleep} purgeitvl[h|m|s]]
          [purgecoldfiles age{d|h|m|s} period]
```

- Watermarks {0.9 0.95} specify window in which *total* disk usage will be kept
- ***`files`*** allows setting of actual data file usage limits
  - relevant and useful when disk is shared with another service or for client-side caching
    - *max* & *nom*: when max is reached, files are purged down to nom
    - purging below *nom* is done if required by total usage > *highWatermark*
    - *base*: minimum / guaranteed space, files will not be purged below this
- ***`purgeinterval {5m}`*** how often to check the disk usage
  - Total usage is checked, estimation of file usage is done by adding up # of bytes written
    - actual cache scan is only done if needed
- ***`purgecoldfiles {disabled}`*** remove files that have not been accessed in *age*
  - disk scan for cold files is forced every *period* purge cycles

Using OSS Logical Volume Manager & pfc.spaces

```
oss.localroot /xcache-root          # Location where sym-links to files will be kept
                                    # (put this on a SSD)

oss.space data /data1/xcache        # Add all your data disks to LVM space
oss.space data /data2/xcache
oss.space data /data3/xcache
oss.space data /data4/xcache

oss.space meta /xcache-meta         # Another space for meta-data / cinfo files
                                    # (put this on a SSD, too)

pfc.spaces data meta                # Tell XCache which spaces to use
```

pfc.decision

`pfc.decisionlib` *`path`* `[`*`libopts`*`]`

- Plugin that decides whether to cache to disk or not
- Reference implementation: https://github.com/osschar/xrdpfc-decision-ucsd

```
pfc.decisionlib libXrdPfcDecisionUcsd.so \
               +^/+store/data/Run2016[A-Z]/[^/]+/MINIAOD/03Feb2017" \
               +^/+store/mc/RunIISummer16MiniAODv2/[^/]+/MINIAODSIM/PUMoriond17_80X_ \
               +^/+store/user/matevz/
               -.
```

Dev notes: need to add support for "redirect to origin"

Tracing / debugging options

- ***`pfc.trace none | error | warning | info | debug | dump  {warning}`***
  - Trace XCache operations, use:
    - info to see what is happening
    - debug when reporting problems
- To debug connections to the fedration (4 Debug, 3 Error, 2 Warning, 1 Info)
  - ***pss.setopt DebugLevel 4*** # Equivalent to *xrdcp -d2 …*
  - This produces A LOT of output, use only when needed
- Use ***`trace`*** of other components, e.g. (but see the docs):
    - xrd.trace conn
    - xrootd.trace emsg login stall redirect
    - ofs.trace delay

pss options - Honorable mention

```
pss.ciosync ssec msec {30s, 180s}      # Retry interval / total timeout for closing of a local
file.
                                       # These numbers are easily too low for a loaded proxy.
                                       # Increase if you see a growing number of file descriptors.


pss.inetmode {v4 | v6} {v6}            # good thing to try if you suspect ipv6 problems


pss.setopt ConnectTimeout seconds      {120s}     # Some of these timeouts are way too long.
pss.setopt DataServerConn_ttl seconds  {20m}
pss.setopt RedirectorConn_ttl seconds  {60m}
pss.setopt RequestTimeout seconds      {5m}


pss.setopt WorkerThreads number        {64}       # A good number, used to be much lower, 4, IIRC.
                                                  # Remove from config if your setting is lower.
```

Setting up a cache server

1. Install, prepare config file, set pfc.trace info
2. If X509 authentication to origin server/federation is required:
   a. Setup X509 proxy for xrootd user, prepare automatic renewal script
      i. Best to have them in */tmp/x509up_u`id -u xrootd`*
   b. Test: as xrootd user run: *xrdcp -f -d2 root://origin.org//some/known/file /dev/null*
   c. ipv6 problems will also show up here! Try setting env `XRD_NETWORKSTACK=IPv4`
3. Test config file, general sanity:
   a. Test: xrootd -c /etc/xrootd/xcache.cfg - this prints startup info to stdout
4. Test startup through init.d / systemd:
   a. Check /var/log/xrootd/<name-passed-to-xrootd-if-any>/xrootd.log
5. Test copy through the proxy:
   a. *xrdcp -d 2 -f root://localhost:1094//some/known/file /dev/null*
   b. If trouble, look at the log, make sure step 2. above works **for xrootd user**
   c. If more trouble - ask on [xrootd-l <xrootd-l@slac.stanford.edu>](mailto:xrootd-l@slac.stanford.edu)

Serverless / client-side caching

- Idea: Whenever XRootd is used, silently route traffic through an impromptu cache server running on the local machine.
  - Files get stored for subsequent use, even in offline mode (sort of like CVMFS).
    - Remote access is not even tried until a missing file / block is requested.
  - **Available only through the POSIX interface!**
  - **And it only works for a single process concurrently.**

```
export LD_PRELOAD=/usr/lib64/libXrdPosixPreload.so
export XRDPOSIX_CONFIG=/path-to-config/disk-cache.cfg
# Run your command
```

Minimal config example:

```
posix.cachelib   /usr/lib64/libXrdFileCache.so
oss.localroot    cachepath
pfc.diskusage    0.9 0.95 files 10G 40G 50G
pfc.ram          512m {256M, 64G; dflt: 256M}
```

Forwarding mode Proxies

Sometimes the job / client knows where they want the data from - specify it as an URL prefix!

```
all.export   /some/path/             # Standard namespace export
all.export   /root:/                 # Yes, only one trailing / !
all.export   /xroot:/                # Read the docs for details.
pss.origin =                         # Pure forwarding mode.
pss.origin = give.me.data.org:1094     # Combination mode, URL to use when
not
                                     # specified.
```

Then this gets used as:

root://proxy.server.org*[//root://data.source/to/use:6666]*//some/path/to/a_file

Direct Cache Access & pss.dca

- On a shared filesystem, redirect clients to read from the FS directly:
  - Only happens when a file is fully downloaded.
    - PSS calls XCache to check for this and XCache marks a special access record to avoid purging of the file.
  - Useful for HPC sites that usually have some fancy interconnect and RDMA.

pss.dca [**recheck** {*tm* | **off**}]

- ***recheck*** specifies interval (in seconds) between checks if the file is fully downloaded - when it is, clients are redirected to local filesystem.
- By default this is off and check is only done at Open.

Configuring XCache cluster

Essential configuration for caching cluster

```
# Redirector running at ports 2040/2041
all.role    proxy  manager          # only accept proxy servers
xrd.port    2040                    # xrootd at 2040
all.manager xrootd.t2.ucsd.edu 2041 # cmsd at 2041
all.export  /store stage r/o        # all servers can stage new files
cms.sched   maxretries 0 nomultisrc # prevent opening of the same file on several machines
cms.fxhold  noloc 15m 4h            # reduce time file existence info is kept (purging!)
```

```
# XCache servers
all.role    proxy server
all.manager xrootd.t2.ucsd.edu:2041
if exec xrootd
  all.export /store r/w             # readwrite!
else
  all.export /store stage r/o       # stage! (redundant if also specified on redirector)
fi
pss.cachelib libXrdFileCache.so
pss.origin   cmsxrootd.fnal.gov:1094
...
```

Greedy clients & Cache server restart

- CMS XRootdAdaptor tries opening of multiple files - multi source reading
  - Works great for true remote access: best data source is found and used predominantly.
  - Is horrible for caching cluster - it enforces creation of many replicas across servers.

```
cms.sched   maxretries 0 nomultisrc
```

- This blocks:
  - retries on failure (Req URL: tried=<srv-list>)
  - requests for additional sources (ReqURL: triedRC=resel,tried=<srv-list>)

- Restarting servers is safe in view of potential multiple copies:
  - Manager cmsd sees that server cmsd has disappeared
    - Holds / stalls redirection requests for files know to be on the missing server for 10 min
    - 10 min is ballpark maximum time for a machine reboot
    - Service restart is much shorter - but gives you a window for some config mistake fixes :)

Dealing with unbalanced disk usage across servers

- The brute way: fiddle with stage option in server cmsd settings
  - Disable staging for fully loaded caches until the less used machines fill up.
  - Really not what you want to do.

- We are thinking about a reasonable staging policy enforced at the redirector cmsd that makes this automatic.
  - Consider: free space / fraction of space & total available space / fraction of space.
  - A cache server should be scheduled for staging of a new file proportionally to the total disk space -- this enforces balanced repopulation of cluster cache space across machines.
  - On the other hand, when a machine is "empty" it should be selected more often +- fuzz.

Development plan / Conclusion

What is coming soon(ish) - until the next XrdWkShp

1. Monitoring, quota-based purging -> tomorrow
2. Optimizations
   a. RAM management (align on page size, do not zero-out)
   b. Smarter block selection for prefetching
   c. Disk write improvements: multiple write queues (optionally separated by LUN)
   d. Overload control and redirection to origin - infrastructure is ready but decision logic is not
3. Efficient running with smaller block sizes
   a. grouping of blocks on download and during prefetch
   b. rework hdfs mode to operate in this way
4. Cache cluster
   a. Balancing of disk usage & staging frequency

. . .

5. Figure out a way to detect corrupted files in cache
    a. Problem: Cache can get a corrupt file from the origin. Cache has no way of detecting this now.
    b. Possible solutions:
        i. The easiest option would be to get a message from the job that detected the issue.
            1. Error code sent from job to XCache server
            2. Harder to do when xrdcp is used to fetch file for the job.
        ii. Some sort of check-summing will have to be involved and has to be available from a service that can authoritatively provide it. Preferably use cksum type that can be "added" by block.

Ask questions: *xrootd-l <xrootd-l@slac.stanford.edu>*

Report problems: *https://github.com/xrootd/xrootd/issues*