XCache - Developments & Plans

XRootD Workshop @ JSI Ljubljana

March 30, 2023

Matevž Tadel, UCSD

Overview

Introduction: overview & history

Review of features & relevant configuration options

Recent developments & Development plan

Questions & Discussion

Introduction

XCache – brand name for XRoot disk-based file proxy cache

In code referred to as PFC – proxy-file-cache, so you will see:

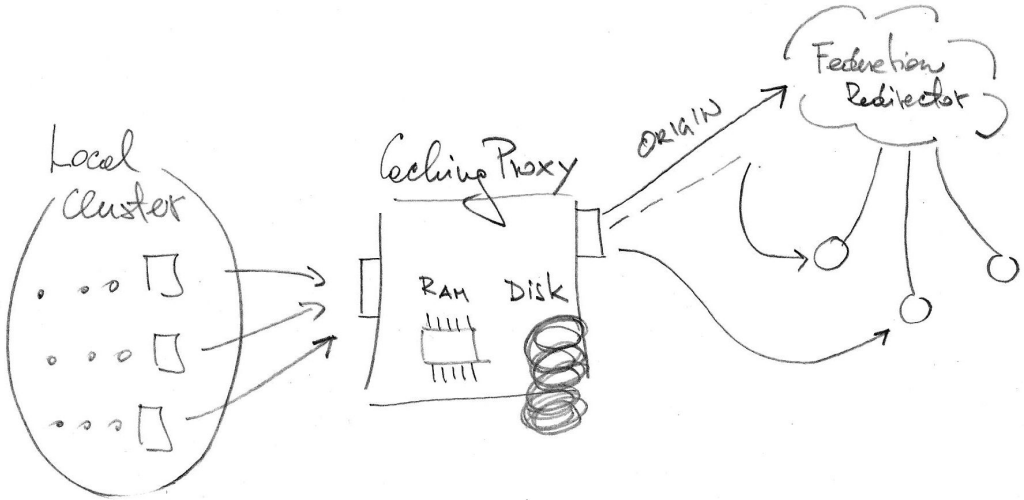
- Classes and file-names prefixed by ***XrdPfc***, e.g. XrdPfcFile and XrdPfcFile.hh/cc
- Configuration options prefixed by ***pfc***, e.g. pfc.blocksize

To Cache … or not to Cache?

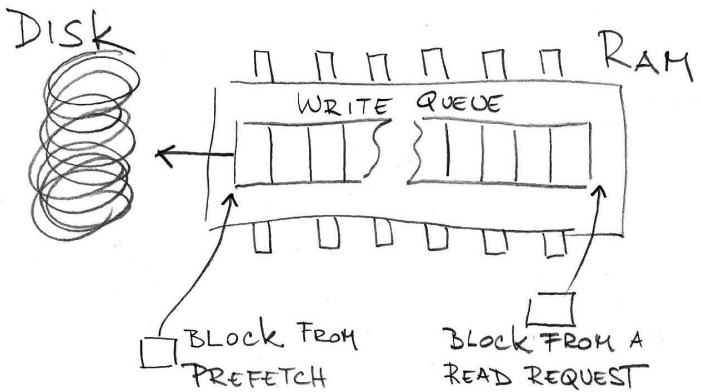
- Benefits of caching
  a. You know you will (or hope to) reuse the data → reduce network use, pressure on data origins
  b. Reduce latency and improve IO efficiency for jobs – especially with prefetching
  c. A way for just-in-time data placement into unmanaged (or semi managed) storage
     - less worry about data/disk loss – if it happens it will get restored with little intervention
- How does this work in real life?
  a. OSG StashCache has some great results for LIGO, Fermilab neutrino experiments, bio-stuff
     - Jobs with varying parameters that all reuse the same input
  b. E.g. US CMS SoCal AOD data cache; all data available at Fermilab
     - caching cluster split between Caltech and UCSD (2ms RTT, 1 Gbps)
     - reasonable data reuse → ballance cached namespace against cache cluster volume
  c. ATLAS – let's see what Ilija tells us :)
- Cache in the world of data lakes, swamps and deserts
  a. Medium-size compute sites subscribe to a portion of possible data namespace
  b. Schedule jobs based on input requirements →caches to pull in data as needed
  c. For large VOs this runs into conflicts with the desire to have tight control over data placement

XCache in one slide

- 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
  - Rely on VFS to help
  - 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
  - Can use http on both ends



On origins, data sources, squids, and uniqueness

- Two modes of getting to the remote data
  a. *direct mode:* specify data origin / redirector, assume all data will be available from there
     - this was original mode of ATLAS (FAX) / CMS (AAA) data federations
  b. *forwarding mode:* source is specified with each request
     - e.g.: root://cache.cluster.here//root://server.to.talk.to//data/silly-user/joe/button.png
     - this is how ATLAS is using XCache now
  c. *combined mode:* forwarding if requested, otherwise use direct mode as default
- Squid / browser-cache is always in mode b.
  a. Further, the host name is part of the caching-object ID, i.e.
     http://foo.org//a_file **is different than** http://bar.com//a_file
- In (XRootd) data federations, and also in XCache, we assume host does not matter – path uniquely identifies a file and its contents.
  a. Data versioning in HEP: directory postfixes (_v2), intermediaries (/Summer2019/) and GUIDs
     - so … at namespace-level
  b. Projects that work around that: StashCache (XCache + cvmfs namespace), XCacheH (?)

History – 10 years!

- V1 (CHEP 2013)
  - First implementation using "the old client"
  - Hdfs fallback & healing
- V2 (XRootD @ ICEPP Tokyo, Nov. 2016)
  - Reimplementation with "the new client" using async I/O for remote access
  - Use XrdOss to access local disk, ability to build caching clusters
  - XCache-V2 presentation gives a good overview of XCache
- Pre xrootd-5.0.0 (XRootD @ IN2P3 Lyon, Jun. 2019; slides)
  - 2½ years of adiabatic improvements and new ways of using it, e.g.:
    - client-side cache / direct cache access / forwarding mode
  - Used in production (CMS, OSG/StashCache) and in testbeds (ATLAS, PRP, INet2)
- Now, xroot-5.6.0 (XRootD @ JSI Ljubljana, Mar. 2023)
  - A number of improvements … to be discussed shorty!
  - Usage: as above + deployments in POPs

Review of

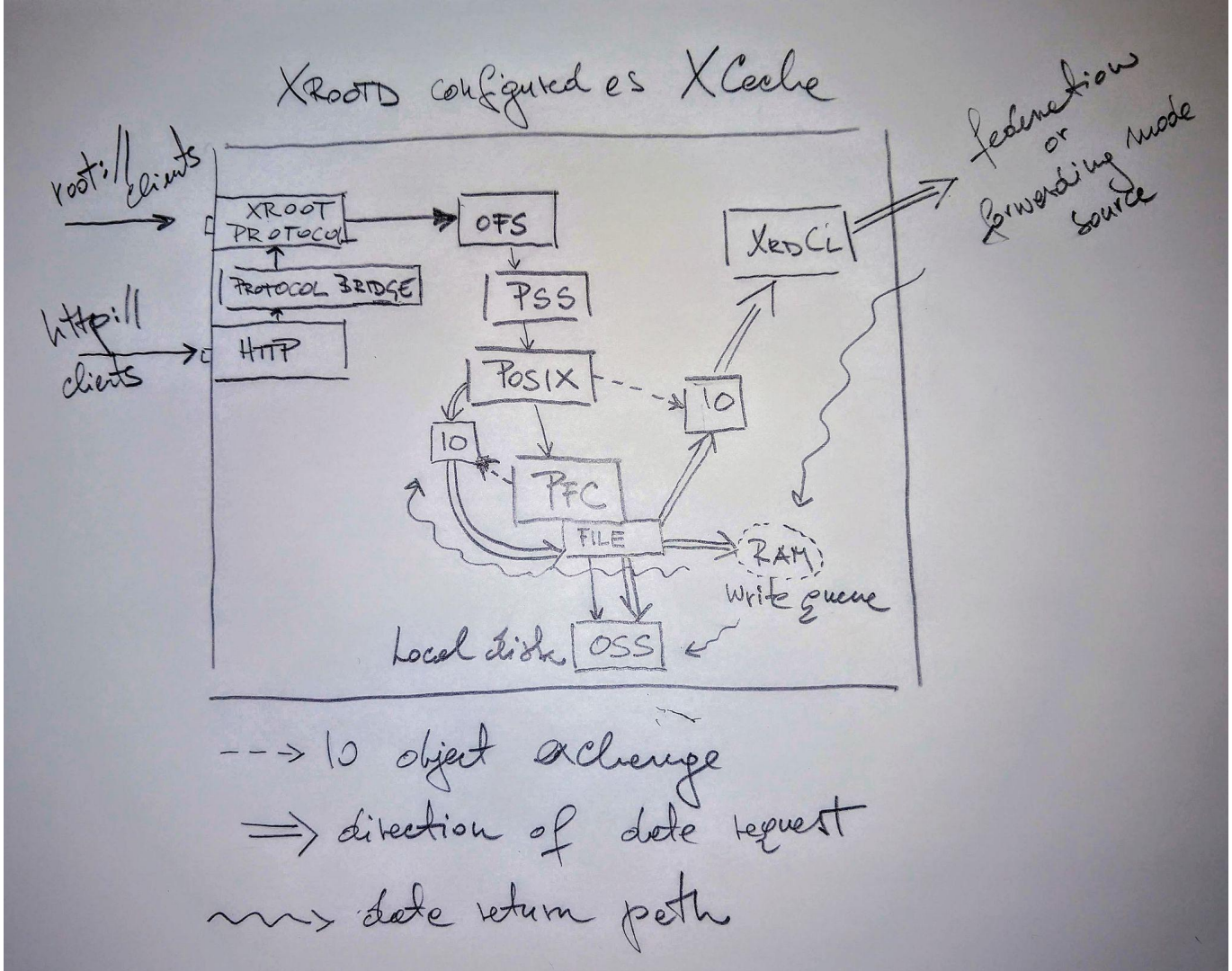
XCache configuration options & features

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 cache              # 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 128k        {4k,  512M}
pfc.prefetch  10              { 0,  128}
pfc.ram       1G              {1G, 256G}
pfc.diskusage 0.9 0.95    {no limits, can also be given in bytes}

xrootd -c hello-cache.cfg
```


XRootD configured as XCache
root:// clients
http:// clients
XROOT PROTOCOL
PROTOCOL BRIDGE
HTTP
OFS
PSS
POSIX
10
10
PFC
FILE
RAM
write queue
Local disk
OSS
XRDCL
federation or forwarding mode source
---> IO object exchange
==> direction of data request
~~~> data return path
~~~

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

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}`
      - or 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.
  - **Tokens** – to be tested / explored

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.
  - Information about downloaded blocks & past accesses is stored in a cache-info (.cinfo) files
- ***`pfc.blocksize`*** sets the size of the block (default now 128k)
  - 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 * 0.125M = **12.5 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}

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 metadata / 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 optional support for "redirect to origin".
Requires jobs to have correct credentials to access federation and not only the cache – which might or might not be true.

There is also: **xrootd.fsoverload bypass redirect origin.org:1094**

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 federation (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

```
# These set parameters for XrdCl - for access to the federation / data sources

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}      # for XCache.
pss.setopt RedirectorConn_ttl seconds  {60m}
pss.setopt RequestTimeout seconds      {5m}


# Number of poller threads
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>*

Essential configuration for a 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 cache            # stage + read-only
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
all.export /store cache            # read-write for xrootd; stage read-only for cmsd

pss.cachelib libXrdFileCache.so
pss.origin   cmsxrootd.fnal.gov:1094
...
```

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** {*time* | **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.

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.**

```
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    # e.g. $JOB_TMP/xcache
pfc.diskusage    0.9 0.95 files 10G 40G 50G
pfc.ram          512m {256M, 64G; dflt: 256M}
```

New developments since the 2019 Lyon workshop

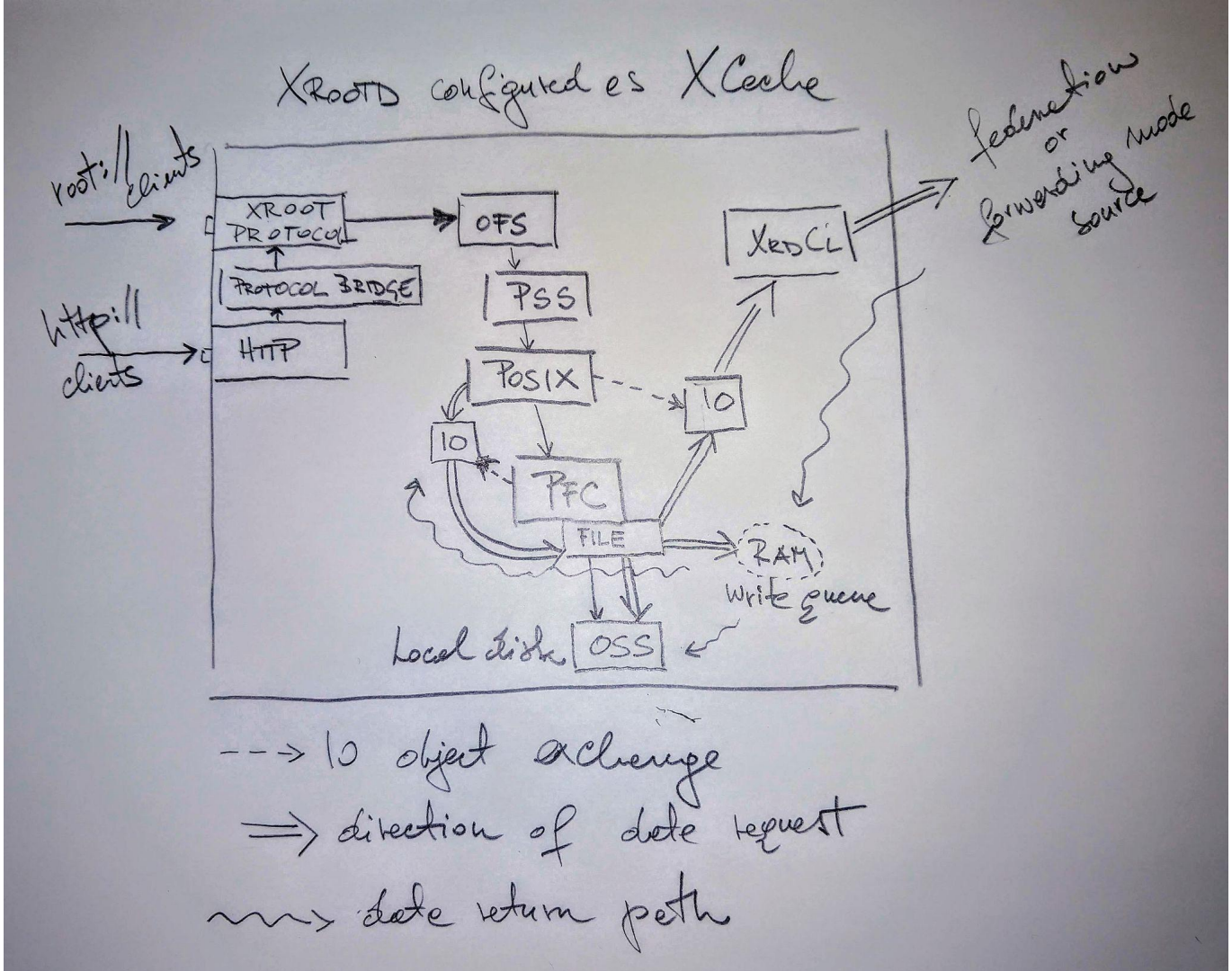
Support pgRead and checksums

```
pfc.cschk { [[no]cache] [[no]net] [off] [[no]tls] [uvkeep {n[d|h|m|s]} | lru}}
```

- A talk by Andy later today – page-read allows parallel transport of 4k crc32c checksums
- Cache corruptions were one of the major motivations
  - network errors → corrupted data in cache → "cache poisoning"
- Default is **net tls** → data is guaranteed to have "come across" correctly
  - automatic retry, within XrdCl
- Checksums also on disk: XrdOssPgi documentation
  - `ofs.osslib ++ /usr/lib64/libXrdOssCsi.so`
- Notes:
  - pgRead uses max 64kB blocks … tie-back to async block size.
  - **uvkeep** – when to purge files with inadequate guarantees

Fully Asynchronous (pg)Read(V), both ways

- Previously, only read requests to the federation were asynchronous
  - Read requests coming from local clients, through XRootd were stalled, waiting for all data to become available
  - This prevents **xrootd.async** from doing what it is supposed to do: enter a "streaming" mode with chunking up of incoming requests into 64 kB blocks
  - Also: Slow origins in OSG / StashCache federation – thread hogging → thread traffic jam
- Now POSIX calls asynchronous read on the cache IO and is called back when data is assembled from the remote, RAM & disk (and direct read)
- This is the reason why **pfc.blocksize** default was reduced from 1M to 128k
  - There is no guarantee that incoming read requests are block-aligned
  - One can increase xrd.async **segsize** – or reduce further the cache block
  - But note: as cache uses pgRead to read from the remote, those will go out in 64 kB blocks anyway.


XROOTD configured as XCache
root:// clients
http:// clients
XROOT PROTOCOL
PROTOCOL BRIDGE
HTTP
OFS
PSS
POSIX
IO
IO
PFC
FILE
RAM
Write queue
Local disk
OSS
XRDCL
federation or forwarding mode source
---> IO object exchange
==> direction of data request
~~~> data return path
~~~

XCache & g-stream

```
xrd.mongstream pfc use flush 10s maxlen 24000 send json insthdr
localhost:9932
```

- Report each file access in g-stream record
  - Data payload is always JSON formatted
  - Various header options to simplify direct consumption
  - remote sources and number of checksum errors are also listed
  - Note: one cache access can mean several client accesses! POSIX hides them in direct mode.

```
{
"event":"file close", -- hardcoded event type for record emitted at file_close time
"lfn": "%s",          -- LFN of the file
"size":%lld,          -- file size [bytes]
"blk size":%d,        -- block size used by cache [bytes]
"n blks":%d,          -- total number of blocks
"n blks done":%d,     -- number of blocks already downloaded / in cache
"access cnt":%lu,     -- number of IO objects that were attached to the file during the time the file was opened
"attach t":%lld,      -- epoch when the first IO object was attached / file was opened at the cache
"detach t":%lld,      -- epoch when the last IO object was detached
"remotes":%s,         -- list of remote origins where the file was read from during the time the file was opened
"b hit":%lld,         -- bytes served from the cache (disk or RAM) [bytes]
"b miss":%lld,        -- bytes served that had to be fetched from the remote [bytes]
"b bypass":%lld,      -- bytes served in bypass mode, read from remote but not written into cache [bytes]
"n cks errs":%d       -- number of checksum errors reported by the XrdCl during remote reads.
```

oss.space: check mount-point & ignore missing disk

oss.space *data* /data1/xcache ***chkmount*** xrd-mnt-check ***nofail***
oss.space *meta* /xcache-meta
pfc.spaces *data meta*

- Really an OFS development … but very relevant for caches
  - chkmount allows one to require a magic file to be present in the mountpoint
    - Otherwise: bad disk – does not get mounted – contents of mount point in / → full /
  - nofail continue without this disk, en error is reported in the startup log
- We recommend using "raw" disks
  - XCache can have O(1000) or more concurrent clients / write streams
    - Slicing each of them onto N disks increases number of iops per disk!
  - Also – it's a cache – if a disk fails we really do not care, we can easily repopulate.
- This allows a cache to be restarted with a missing disk, or with a new disk – everything "just works"

Caching cluster – do not allow multi-source reading

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

- Scenario: caching cluster, greedy multi-stream readers
  - E.g., CMS XrdAdaptor to TFile → does multi source reading to select the best performing one
  - uses tried=hostname opaque parameter to instruct the redirector NOT to give it back an existing source; this works very well for true remote reading
  - But, redirects the client to a new cache server ⇒ multiple copies of the same file in the cache
- Provide additional opaque parameter: triedrc=resel
  - Tells cmsd that the open request is trying to do a source re-selection
    - We disable this for the cache with *nomultisrc*
- Client must be honest!
  - Old CMSSW versions still do not have the *resel* support → *maxretries 0*
    - This fails a job when there is a caching error – not ideal.

Minor changes – a day in the life of an XCache developer

- Use OssAt during purge traversal to reduce pressure on the filesystem.
- Merging of cinfo records
- Report cache stats as part of summary monitoring
- Allow up to 512MB blocksize (was 16 MB, requested 64MB for Ceph)
- Reduce cache-block allocation churn:
  - allocate cache-blocks on page boundaries;
  - keep up to 5% of standard-sized blocks in a pocket for quick reuse
- Multiple write queues – saturate SSDs!
- Winterization
  - Consistency checks between cached and actual file meta-data (size, access log corruption)
  - Resiliency of cinfo files (separate core/access cksums)
  - Purge file on any sign of error, report error to requesting clients
- HDFS mode "large-block" feature is disabled.
  - Was implemented with one cache-file per hdfs-block → only made sense for healing.
  - Can be revived / implemented better if there is interest (through a prefetch policy).

Development plans

Per directory usage, quotas & monitor / purge plugin

- This is already partially implemented
  - pfc.dirstats maxdepth 2 dir /foo/bad/users/*
  - dumps report in a file every purge cycle
- There has been a desire for per-directory quotas around for a while
  - OSG with multi-tenant caches; runaway usage by a VO or even a single user
  - but never strong enough push to actually implement it (it is quite hard)
- Alternative or extension, proposed by Brian / Kingfisher project is:
  - LotManager component that takes over cache monitoring & management
  - can be a plugin … but could also be a service … or both

Prefetching

- Current (well, 10-years old) algorithm is rather stupid, just reads the missing blocks in order.
  - This is fine for in-order access … or for slow-reading clients.
  - Also, results in a full file eventually showing up in the cache.
- A better algorithm would allow one to:
  - specify heuristic: on open, read N-bytes from the head, M-bytes from the tail (ROOT)
  - let the incoming reads drive the actual prefetching … read ahead of the last reads
    - tricky for vector-reads without knowing the branch/basket layout
  - specify how far ahead to read → do not read & cache a full file unless required
    - This is particularly important if / when we move to (extremely) large files.
  - Provide preRead(v) in XrdOucCacheIO interface – an ABI change → not before v6
    - preread vector can be sent as a part of the kXR_read request – send zero size read
      - the interface would need to be added in user code, too, TXNextGenFile & similar

Service work, community, etc.

- Follow up on developments in XRootd
- Keep in touch with the main users:
  - Advise, improve existing features, develop extensions
  - Help with debugging
  - Weekly *xcache-devops* meeting (Thursday 11am Pacific)
    - OSG, ATLAS, CMS + others, as needed
    - *xcache@opensciencegrid.org*
    - slack *OSG#xcache*
- General user / developer support
  - Ask questions: *xrootd-l <xrootd-l@slac.stanford.edu>*
  - Report problems: *https://github.com/xrootd/xrootd/issues*

Questions / Discussion