XCache: The new Resource Monitoring & Purging

XRootD Workshop @ STFC UK, Abingdon

September 12, 2024

Alja & Matevž Tadel, UCSD

Overview

- Introduction: 3-slide overview
  - For history, review of features & config options → see XRootd@JSI 2023 talk
- Review of some relevant stuff: threads & monitoring collection
- Recent developments → planned to go into 5.8
  - Old-style (well, current) implementation of cache purge and directory statistics
  - Resource Monitor & Cache Heartbeat
  - Detailed collection of directory usages and access statistics
  - Purge plugin with a directory quota based minimal implementation
    - Used also by LotMan – the cache space manager for Pelican
- Possible enhancements of the above
- Things that could be worked on

Introduction

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

In code referred to as PFC – proxy-file-cache

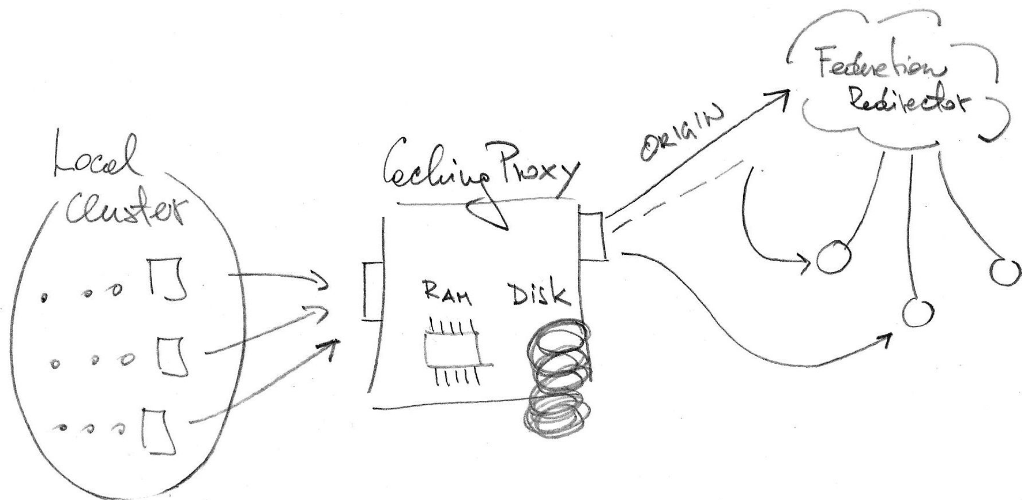
- All classes & structs are in ***namespace XrdPfc***
  - files are in ***src/XrdPfc/***
  - file-names are prefixed by ***XrdPfc***, e.g., ***File*** is in src/XrdPfc/*XrdPfcFile.hh/cc*
- Configuration options prefixed by ***pfc***, e.g. pfc.blocksize

To Cache … or not to Cache?

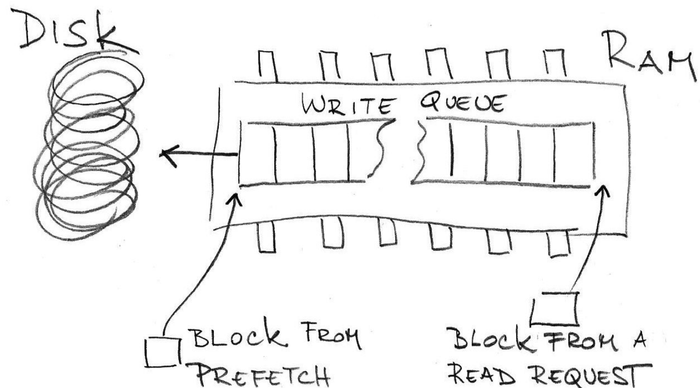
- Benefits of caching
  a. You know you will (or hope to) reuse the data → reduce network use, pressure on data origins
  b. Reduce latency and improve IO efficiency for jobs – especially with prefetching
  c. A way for just-in-time data placement into unmanaged (or semi managed) storage
     - less 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, 100 Gbps)
     - reasonable data reuse → ballance cached namespace against cache cluster volume
- 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
     - For large VOs this runs into conflicts with the desire to have tight control over data placement
  c. Pelican: somewhat automatically managed staging space

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(s) on both ends

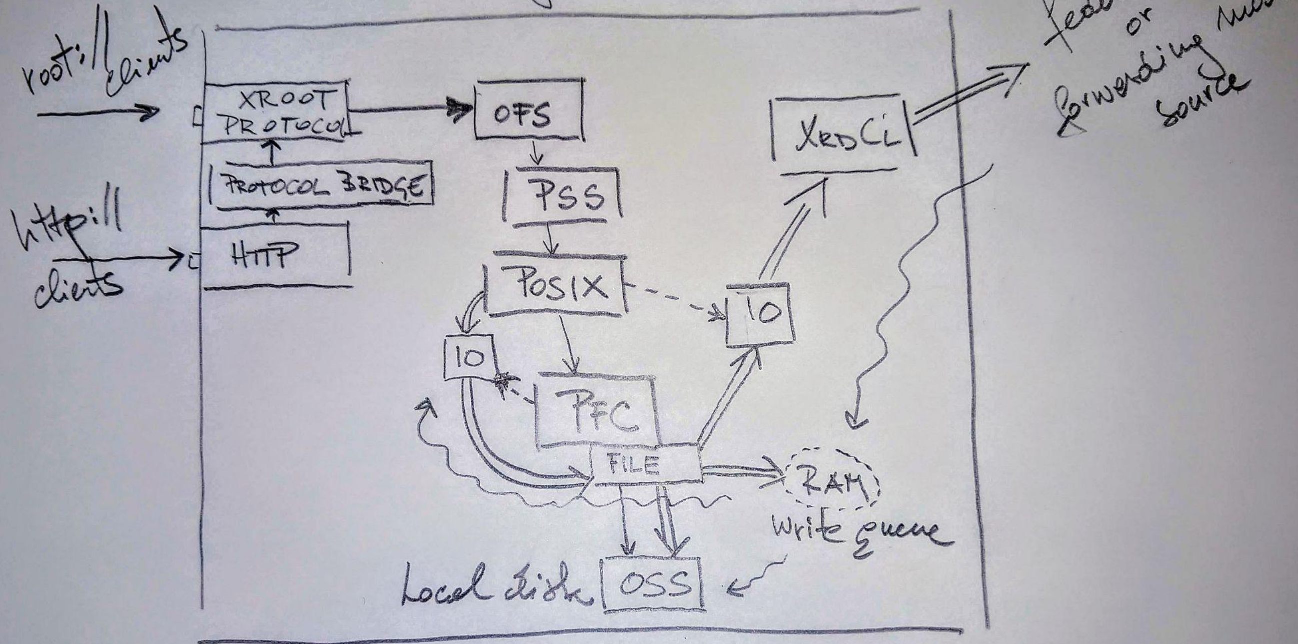


On origins, data sources, squids, and uniqueness

- 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 (?)

Review of relevant stuff

XROOTD configured as XCache



- - -> IO object exchange

⟹ direction of data request

~~~> data return path
~~~

Threads in XRootd & XCache

- Thinly wrapped POSIX threads, see *XrdSysPthread.h/.cc*
  - mutexes, condition-variables, semaphores (in XrdCl)
  - limited number of atomics, mostly without strict ordering
- Main threads in core XRootd
  - accept ("forking"), buffer-reshaper, scheduler for detached tasks, can be delayed (XrdJob)
  - pollers, OssAio, OssCache-scanner, OfsEventFlush, OfsEvRecv
- Threads in XrdCl
  - polling / event handling threads – *pss.setopt ParallelEvtLoop 10*
  - worker threads, max number – *pss.setopt WorkerThreads 64*
- Thread*pfc.flush*s in XCache:
  - Purge, Prefetch, DiskWriters (4 by default)
  - Data-flow through cache is **completely asynchronous, both ways**:
    - in-between Ofs/Pss/Posix and XrdCl (via IO objects)
  - XrdJob - file sync. when conditions are met, see

Current per-file statistics & purge implementation

- Per file access statistics, *XrdPfc::Stats* – collected for each open file
  - *{ N_ios, duration, N_bytes hit/missed/bypassed/written, N_cksum_errors }*
  - summary put into .cinfo file as an access record (with timestamps)
  - if so configured, also sent out at close time via the pfc g-stream
- Purge thread
  - if (purge_needed) { scan_namespace(); purge(); } else { coffee_break(); }
- Purge decision – initiate purge?:
  - low / high water mark – based on total disk usage (potentially other users of the space)
  - file usage limits (total N_bytes written accumulated to avoid re-scanning)
- Purge scan, purge execution
  - prepare an ordered map (by last access time in cinfo) of candidates
  - loop over this … but skip files that are now open or had purge protection set upon them

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

Directory statistics, recent history

- There was the desire for per-directory monitoring & 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
- Proto directory statistics from summer 2019 – collected during purge scan
  - *DirStats* constructed from the file statistics of files that were closed since the previous purge
  - Aimed to include per directory quota, initial idea of Heart-Beat and continuous DirStats
    - Turned out to be a rather hard problem – fast incoming changes, slow processing
  - Nobody pushed for quotas / configurable purge again until Kingfisher in early 2023
    - LotMan (Lot Manager) component that takes over cache monitoring & management
    - a plugin + local daemon + a management service
- Then, with Pelican starting ~ a year ago, this all became rather urgent …

The new Resource Monitoring & Purging

Recap of issues / complexity involved

- Requirements / Goals:
  - Provide sufficiently detailed and accurate snapshots of XCache state and usage
  - Optimize information collection and storage – reduce number of required cache namespace traversals
  - Design a good-enough purge-plugin API, provide a directory quota based prototype

- Directory state
  - changes come on per-file basis … from a super-multi-threaded environment
    - and, esp. for Reads(), this comes from critical sections of XCache
    - ⇒ avoid too aggressive / frequent collection, provide a simple way for data to "come out"
  - aggregation over child directories → how to avoid redoing this too often
  - creation of new (sub-)directories & their removal
  - How to best avoid parsing file-names and traversing the cached namespace
  - The collection / reporting needs to be implemented so that it makes sense from both:
    - file / low-level / high-frequency event handling side; and
    - top-level consumer, like purge plugin or monitoring information reporter

Solution overview – *XrdPfc::ResourceMonitor*

- Owns and manages data structures representing:
  - filesystem usage
  - per directory usage & "traffic"
- Runs a dedicated *heart_beat()* thread
- Collects changes from other threads in a couple of queues
- Directory state: Usages vs. Stats (changes / deltas)
  - Usages get updated from DirStats periodically (last open/close timestamps are current)
- Tree representation for continuous updates
- Export snapshots in vector form for reporting and for purge
- Reduce reliance on cinfo information – use stat!
  - Measure sizes in stat-blocks, 512 bytes – more accurate anyway (holes in files)
    - get it for free from stat, the old way was calculating it from cinfo block bit-vector.
  - Use mtime of cinfo file to determine last usage time, "touch" on open

Data types for usages and stats

- struct Stats
  - { N_ios, duration, N_bytes hit/missed/bypassed/written, N_cksum_errors }
- struct DirStats : public Stats
  - { StBlocksRemoved, NFilesOpened/Closed/Created/Removed, NDirectoriesCreated/Removed}

- struct DirUsage
  - { LastOpen/CloseTime, StBlocks, NFilesOpen, NFiles, NDirectories }

- struct DirState
  - { DirStats here_stats, recursive_subdir_stats;
DirUsage here_usage, recursive_subdir_usage;
DirState *parent;
std::map<dir-name, DirState*> subdirs; }

Structs in C++-form

```
class Stats
{
   int       m_NumIos = 0;          //!< number of IO objects attached during this access
   int       m_Duration = 0;        //!< total duration of all IOs attached
   long long m_BytesHit = 0;        //!< number of bytes served from disk
   long long m_BytesMissed = 0;     //!< number of bytes served from remote and cached
   long long m_BytesBypassed = 0;   //!< number of bytes served directly through XrdCl
   long long m_BytesWritten = 0;    //!< number of bytes written to disk
   long long m_StBlocksAdded = 0;   //!< number of 512-byte blocks the file has grown by
   int       m_NCksumErrors = 0;    //!< number of checksum errors while getting remote data
   …
};

class DirStats : public Stats
{
   long long m_StBlocksRemoved = 0; // number of 512-byte blocks removed from the directory
   int       m_NFilesOpened = 0;
   int       m_NFilesClosed = 0;
   int       m_NFilesCreated = 0;
   int       m_NFilesRemoved = 0; // purged or otherwise (error, direct requests)
   int       m_NDirectoriesCreated = 0;
   int       m_NDirectoriesRemoved = 0;
   …
};

struct DirUsage
{
   time_t    m_LastOpenTime  = 0;
   time_t    m_LastCloseTime = 0;
   long long m_StBlocks      = 0;
   int       m_NFilesOpen    = 0;
   int       m_NFiles        = 0;
   int       m_NDirectories  = 0;
   …
};
```

```
DirStateBase
{
   std::string  m_dir_name;

   DirStateBase() {}
   DirStateBase(const std::string &dname) : m_dir_name(dname) {}
};

struct DirState : public DirStateBase
{
   typedef std::map<std::string, DirState> DsMap_t;
   typedef DsMap_t::iterator               DsMap_i;

   DirStats     m_here_stats;
   DirStats     m_recursive_subdir_stats;

   DirUsage     m_here_usage;
   DirUsage     m_recursive_subdir_usage;

   DirState    *m_parent = nullptr;
   DsMap_t      m_subdirs;
   int          m_depth;
   …
};

struct DataFsStateBase
{
   time_t    m_usage_update_time = 0;
   time_t    m_stats_reset_time = 0;

   long long m_disk_total = 0; // In bytes, from Oss::StatVS() on space data
   long long m_disk_used  = 0; // ""
   long long m_file_usage = 0; // Calculate usage by data files in the cache
   long long m_meta_total = 0; // In bytes, from Oss::StatVS() on space meta
   long long m_meta_used  = 0; // ""
};

struct DataFsState : public DataFsStateBase
{
   DirState        m_root;
   …
};
```

Event queues for file events

The queues:

```
Queue<int, OpenRecord>          m_file_open_q;
Queue<int, Stats>               m_file_update_stats_q;
Queue<int, CloseRecord>         m_file_close_q;
```

Registration functions:

```
int register_file_open(const std::string& filename, time_t open_timestamp, bool existing_file);
void register_file_update_stats(int token_id, const Stats& stats);
void register_file_close(int token_id, time_t close_timestamp, const Stats& full_stats);
```

- open → returns *int token_id* – allows quick lookup of the *DirState** later on
- update →register delta-Stats, multiple calls for the same *token_id* get added up together
- All queues have separate write / read buffers – periodically swapped.
- The heart_beat() thread is the owner of the DirState stuff → no locks

Event queues for purging

Queues for file removal events (can come from evict / *Cache::UnlinkFile()*):

```
Queue<DirState*,    PurgeRecord> m_file_purge_q1;
Queue<std::string, PurgeRecord> m_file_purge_q2;
Queue<std::string, long long>    m_file_purge_q3;
```

Registration functions:

```
void register_file_purge(DirState* target, long long size_in_st_blocks);
void register_multi_file_purge(DirState* target, long long size_in_st_blocks, int n_files);
void register_multi_file_purge(const std::string& target, long long size_in_st_blocks, int n_files);
void register_file_purge(const std::string& filename, long long size_in_st_blocks);
```

- Funcs with *DirState** argument used when known – avoid tokenization.
  - In evict case this simply can not be known.
- Optimization for purging of multiple files.

ResourceMonitor::main_thread_funciton()

- perform_initial_scan()
  - populate the basic usages
  - some trickery needed to facilitate opening of files during the traversal
- heart_beat() loop
  - Every 10 s: Apply events from the queues. Order: open / update / close / purge
  - Every NN s: [NN = 60, to be increased & made a little configurable]

    1. Apply Stats to Usages, upward propagate stats & usages

    2. If needed, prepare *DataFsSnapshot* – (limited-depth) vector-form of DirState tree.

       - Export to file-system as binary blob or JSON
       - [Potentially pass it to purge plugin, in a dedicated task / thread.

    3. Clear-out DirStats – reset to zero.

       Remove leaf empty directories (unless purge is currently in progress).
  - Every MM s: [MM = 120,

    Perform purge check. If needed, prepare *DataFsPurgeShot* → pass it to purge thread task.

    [PurgeShot is limited-depth vector-form export of DirUsages only.]

Purge & Purge plugin API – Preliminary

- When purge is needed, purge plugin is contacted first
  - It can return a list of *PurgePin::DirInfo { dir-path; bytes-to-remove }* structs
  - For each entry, the sub-tree is traversed and files are collected in the LRU order, to satisfy the requested removal volume.
  - The plugin can request removal beyond the volume actually required by the configuration.
- If data still needs to rbe emoved to satisfy the configured space constraints, the regular purge-scan and remove is ran to complete the purge cycle.

Internal reference implementation *XrdPfcPurgeQuota : public XrdPfc::PurgePin*

pfc.purgelib libXrdPfcPurgeQuota /etc/xrootd/quota.cfg

```
/store/group/visualization 500G
# /store/user/* 100G DOES NOT WORK (YET)
/atlas                     0
```

To be released like this in 5.8 – and potentially extended for 6.0

Development plans

Final testing & Release of the above

- Release as:
  - a) keep a branch synchronized on top of master + OSG build; or
  - b) dedicated 5.8 with just these changes
- Limited scope, detailed testing at UCSD caches + Pelican
  - I'd be nervous to unleash this upon the world
- If interested in testing – please do get in touch
  - this could influence mode of the release
- We've just started the a) option …

To explore: Building upon usage snapshots

- Right now, we don't do anything (well, write the latest stats on disk as json)
- Note – what heart_beat() produces is a change over the last (fixed) period
  - It is not easy to:
    - aggregate beyond that (collection stats cleared); or
    - average over arbitrary time intervals (requires floating point representation of DirStats)
- However – Cache can dump every Snapshot on disk → *usage-<time>.bin*
  - Summing those up is mostly trivial (+-, directories can be created and removed)
  - pfc-usage-last.bin →add them up into *usage-15min.json, usage-60min.json, usage-180.json*
  - How to configure this?
- Monitoring collection – via xrootd itself – export them through */pfc-stats/*
  - shortcut XrdPfc::File to serve the file wo/ cinfo (or create a dummy .cinfo, or use http handler)
  - special purge
- No shovelers and collectors – monitoring sinks/consumers pull monitoring data
  - auth – allow access to specific roles / uids

Possible Purge plugin extensions

- The new infrastructure is able to support a more general purge algorithm
  - Purge of specific, individual files.
  - Execution of sub-tree traversal, along the PurgeShot vector, from within the plugin.
    - Can perform additional checks / calculations.
    - The traversal & directory scanning code is general enough and easily reusable.
      - Already used in the initial-scan and for purge sub-tree traversal.
  - All that is needed is that purge event gets reported back to ResourceMonitor queues.
    - Might need to provide UnlinkAt.

Getting in touch with the XCache crew

- XRootd developers + community of main users:
  - Weekly *xcache-devops* meeting (Thursday 11am Pacific)
    - OSG, Pelican, ATLAS, CMS + others, as needed – or desired
    - *xcache@opensciencegrid.org*
    - slack *OSG#xcache*
  - Advise, improve existing features, develop extensions
  - Help with debugging, analysing issues
- General user / developer support
  - Ask questions: *xrootd-l* <xrootd-l@slac.stanford.edu>
  - Report problems: https://github.com/xrootd/xrootd/issues

Should improve 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
- Use the new resource monitor + heart_beat() to drive / influence prefetch.