Pelican: Building out XRootD functionality via plugins

XRootD Workshop ‘26

Brian Bockelman

MORGRIDGE
INSTITUTE FOR RESEARCH

Pelican – Building a CDN for Science

- The vision behind the **Pelican Platform** is to create software for global data distribution.
- Goal is to bring together the distributed data repositories (object stores, filesystems, Globus, HTTP endpoints) and the compute capacity (clusters, notebooks, browsers, laptop) around the planet.
- The Open Science Data Federation (**OSDF**) is the largest running instance of Pelican.

CLOUDFLARE

+

"Cloudflare for Science"

Infrastructure for scaling access to your storage

Center for High Throughput Computing (**CHTC**):

Founded in 2006 by Miron Livny; Bockelman joined in 2019.

Research center, split across the **UW-Madison** (part of the new College of Computing and AI) and **Morgridge** Institute for Research

Goal is to advance the state of Throughput Computing.

Produces the HTCondor and Pelican Software.

OSG Consortium

Founded in 2005.

Community organization led by the OSG Council.

Advances science using distributed High Throughput Computing (dHTC).

Runs a broad Fabric of Services, used by both individual researchers and large collaborations / experiments.

CHTC - Software

HTCondor Software Suite

Manages computing workloads and capacity.

Project started in 1985.

Pelican Platform

Manages & delivers data, connecting it to compute.

Project started in 2023.

CHTC's Impact – HTCondor Installations

HTCondor Quick Facts:

- **>200** known installs.
- **>350k** cores in largest publicly-known pool.
- **>100** institutions contributing to the single OSPool.

From industry (SpaceX, Dreamworks) to government (DOE, USGS) to academia (CERN, U. Nebraska), over 200 organizations have publicly identified as having an HTCondor installation.

OSDF: A National Object Delivery Service

Value Proposition

Shares your data (Objects) with authorized consumers.

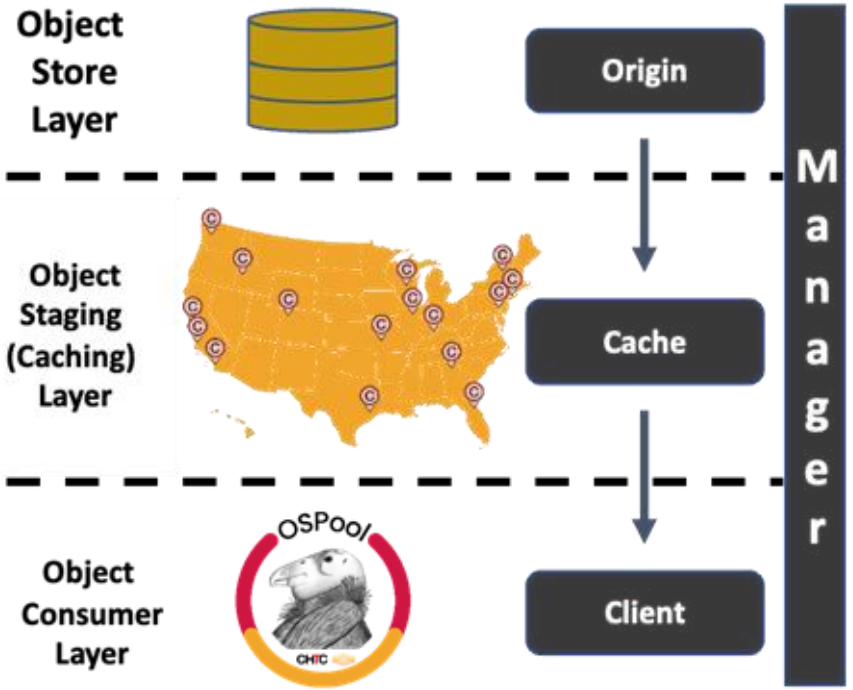
Manages the load of Object reuse on your Object Store.

Fuses Objects from diverse Object Stores through a common namespace and API.

Stages Objects in network-embedded storage.

Who can benefit?

Open to any US researcher, data (object) provider, or computing facility.



>280 PB moved

>6.5B Objects delivered

In the last 12 months.

34 caches throughout the world, at points of presence within major compute centers and the global R&E networks (ESNet, Internet2).

Who is using the OSDF?

Workloads that have significant **reuse and delivery** of objects at remote autonomous Object Stores.

Object Stores with limited capacity or egress costs.

NOAA: Via AWS OpenData

NSF: NCAR, NRAO, LIGO, EHT

DOE: FNAL (LHC, DUNE), JLab, Fusion

Key concepts

Object: both data & non-data objects (containers, models, libraries).

Object Store: a S3-compatible endpoint, filesystem, or HTTP webserver.

Origin: connects an Object Store with national infrastructure. Objects are moved from (GET) and to (PUT) the store through Origin.

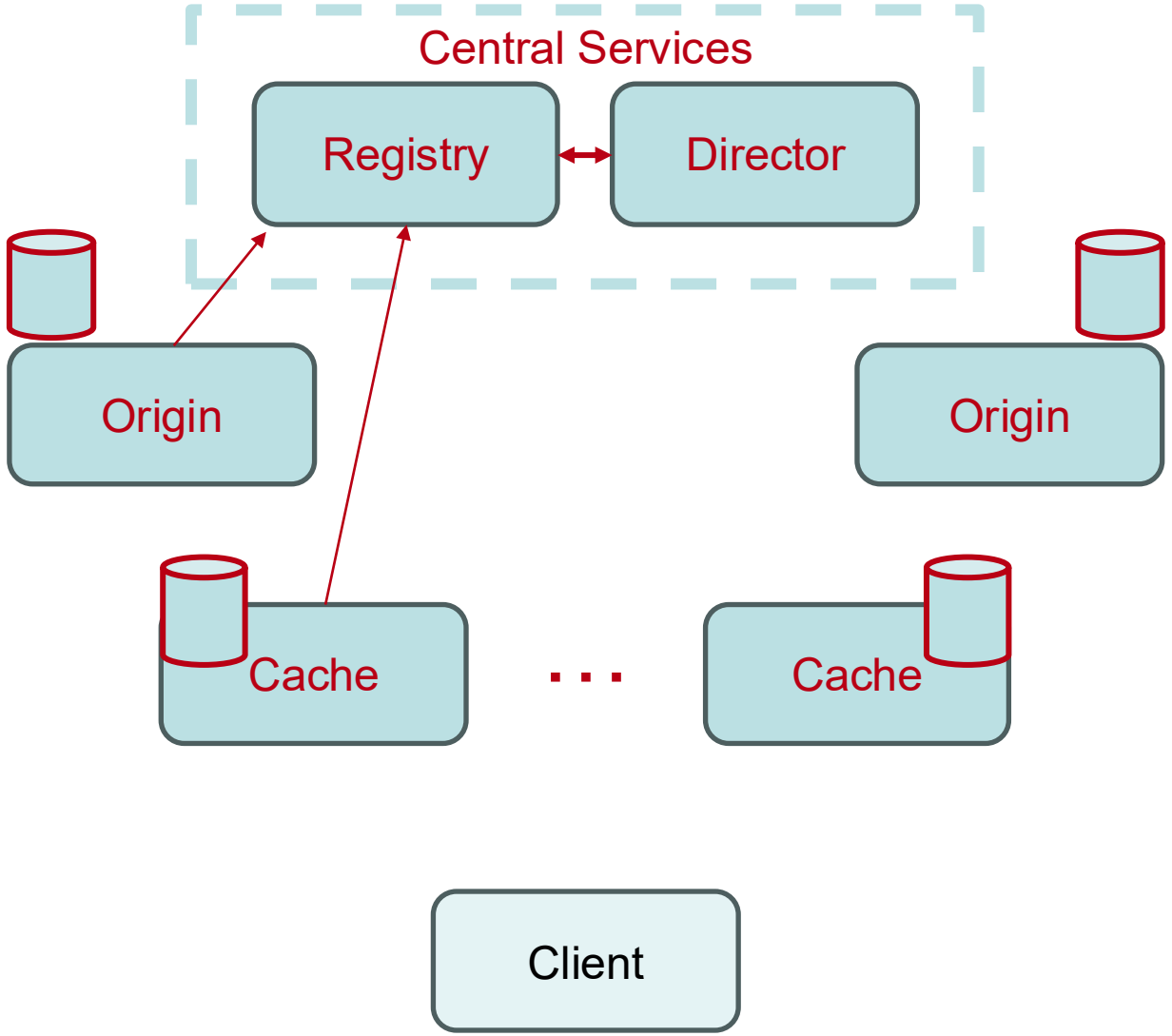
Pelican Data Federation - Architecture

The central services in Pelican:

Registry: stateful database of all known namespaces, services, and corresponding public keys.

Only registered endpoints can offer services to the federation!

Director: takes request for an object, selects a service (e.g., a cache) to perform the request.



Pelican Data Federation - Architecture

The distributed services:

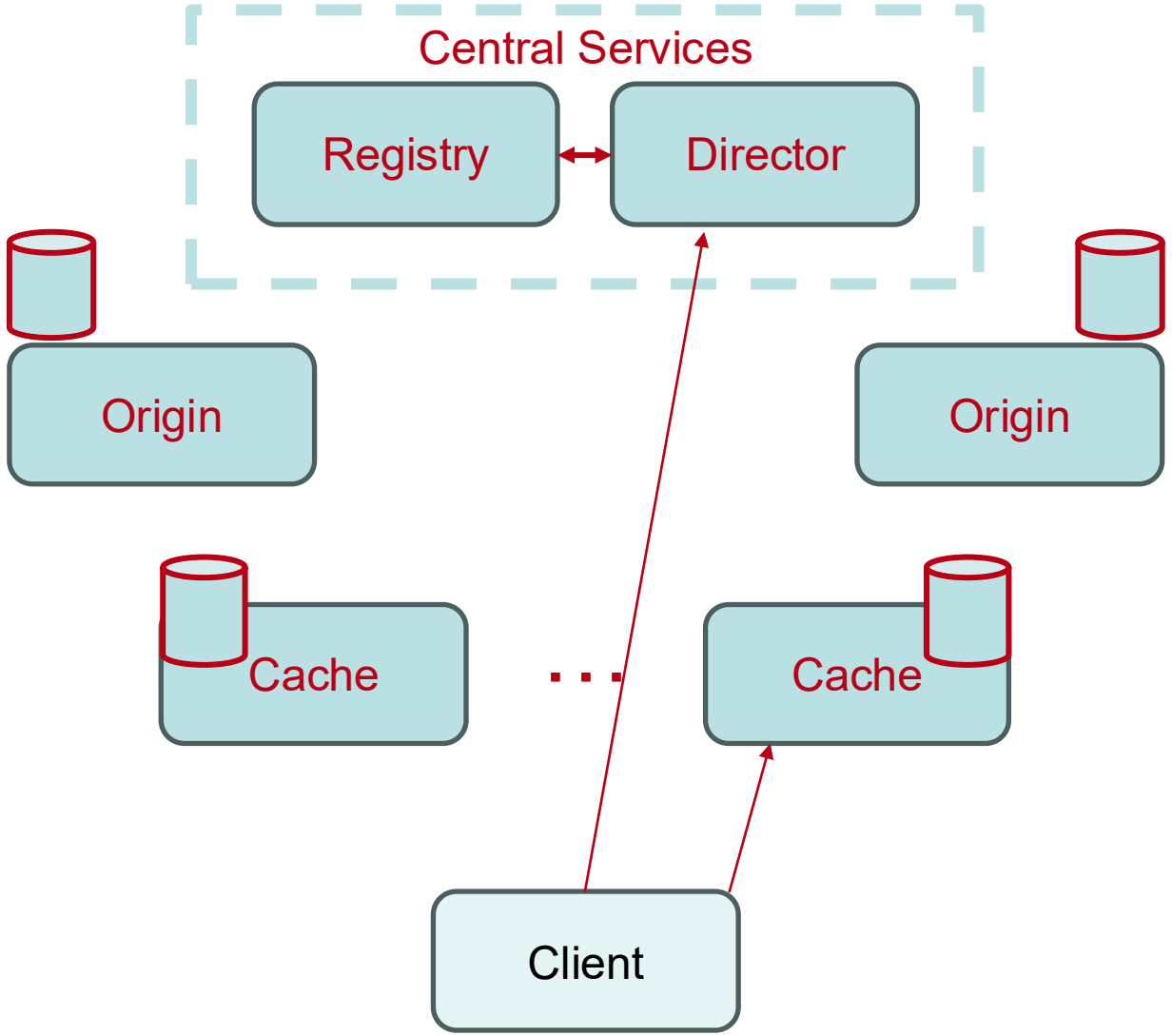
Cache: Manages storage, ideally "close" by the compute infrastructure.

Origin: Connects the data repository / object store to the OSDF.

When a client wants an object, it'll contact the director with the object name and the desired operation (GET/PUT).

The director selects a service (cache) to perform the work and sends hints about the credential (token) needed.

The client will subsequently contact the cache for the object.



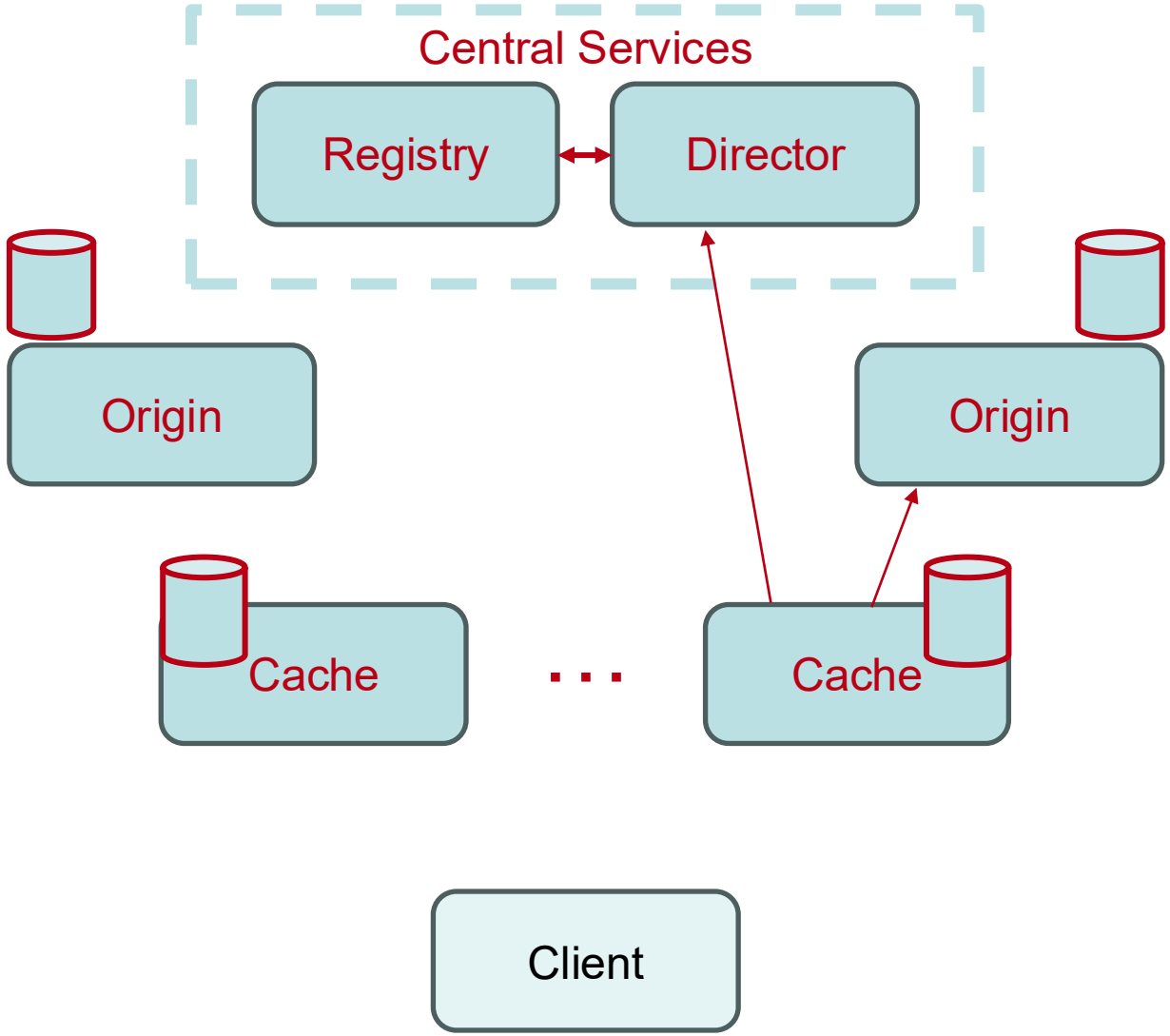
Pelican Data Federation - Architecture

If the cache has the object, great!

It'll transfer it to the client.

Otherwise, the cache will ask the director for an origin.

The cache gets the origin service which can subsequently serve the object.



Pelican uses XRootD for data delivery

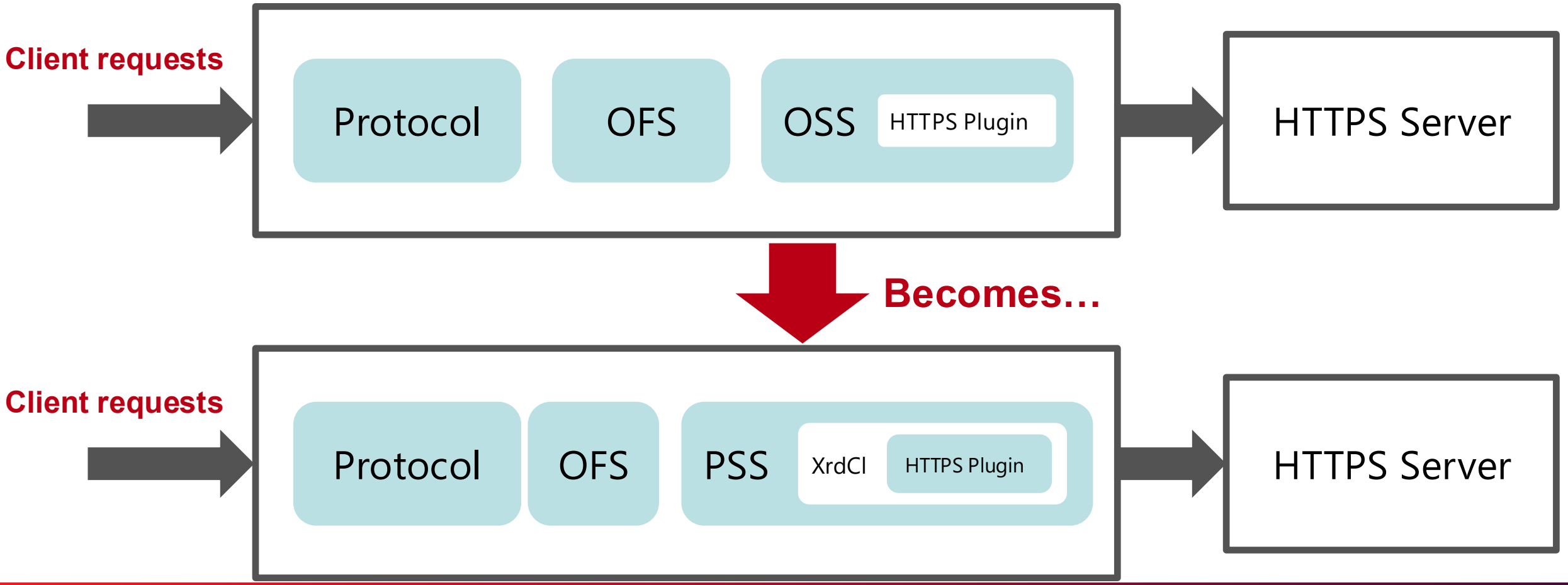
- Each distributed Pelican service (origin, cache) consists of two processes:
  - The `pelican` process performs control plane actions: connects to the federation, registers the service, performs monitoring, issues tokens, handles web UI. At startup, it generates a new XRootD configuration.
  - The `xrootd` process moves bytes over the HTTP protocol (GET/PUT of objects).
- We leverage the battle-tested community code in XRootD and several of the plugins.
- However, we also customize the service's functionality through the use of the plugin layers.
- What layers do we interact with?
  - **Storage API** (OSS): Different storage backends (S3, HTTP, Globus, multiuser)
    - And stacking to alter behavior: POSCv2, deadlock detection, statistics.
  - **Name translation** (N2N): Non-trivial namespace mapping.
  - **HTTP plugins**: Providing simple REST APIs to extend functionality.
  - **Client** (XrdCl): providing a C++ client library to end-users and ability for caches to use Pelican.
- Let's explore!

Storage: XRootD's 'OSS' API

- ~~OSS = Office of Strategic Services, the espionage agency of the US Government during WW2~~.
- OSS = Open Storage System, the API for providing storage functionality for the XRootD framework.
  - The OSS is the lowest-level API, invoked by the OFS (Open File System) layer. The OFS manages higher-level functionality, such as authorization and redirection.
- Example of OSS implementations:
  - **(Default)**: Invokes equivalent POSIX APIs.
  - **PSS**: Proxy storage services, proxying to another server using the XrdCl (client). This is how XCache works.
  - **Multiuser**: Passes through to the POSIX API *after* setting the filesystem UID; allows XRootD to write as a different Unix user based on the security credential.
  - **Globus**: Use a Globus collection / API as storage backend.
  - **S3, HTTP**: Presented last 2 years; utilizes libcurl / has a C++ implementation of S3 on top.
- Last year I presented our new approach to the of the OSS API:
  - **Replace the OSS plugins with XrdCl plugins**.

Vision – XrdCl plugins replace similar OSS plugins

- Given the server's OSS proxy implementation invokes the client, if you implement the XrdCl API, you get the server functionality "for free" *and* it works with the client.



Speedbump: Wrong names!

- Well… not quite:
  - For **XrdOss** plugins, we have a robust way to use the configuration file(s) to bootstrap plugin config.
  - For **XrdCl** plugins, a completely separate configuration approach!
    - Since Pelican auto-generates the XRootD configuration, adding environment variables wasn't *the worst* but it sure made it tough to "view" configuration (half in file, half in env vars).
- Pinch point: naming!
  - The XrdClS3 plugin tried to closely mirror the DAVIX style of URL/path construction for S3.
  - Needed a way to "mount" the XrdClS3 namespace under the existing paths.
  - Instead of embedding name mapping strangely inside the client plugin … used the "Name 2 Name" layer to translate!

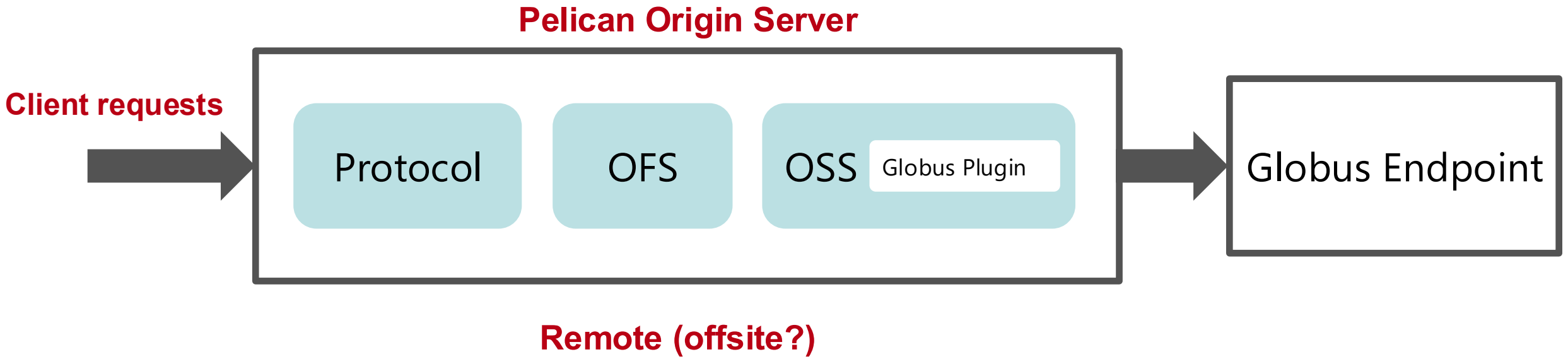
Example Config:

```
oss.namelib libXrdN2NPrefix.so

# Map /store/* to /data/cms/*
prefixn2n.rule /store /data/cms

# Map /cache/* to /tmp/cache/*
# with strict slash handling
# (preserves //)
prefixn2n.rule -strict /cache \
  /tmp/cache
```

See: https://github.com/PelicanPlatform/xrootd-s3-http#configure-the-n2n-prefix-plugin

Globus Support in XRootD

- One reason XRootD is used is that it supports all the peculiarities of the WLCG.
  - Makes storage remotely available that does not support HTTP, xrootd protocol, X.509/VOMS authorization, token authorization.
- Globus, OTOH, is quite popular at US insitutions.
- How do we have “best of both worlds?”: compatible with Globus, compatible with WLCG?
- Answer: treat it as a Pelican storage backend!

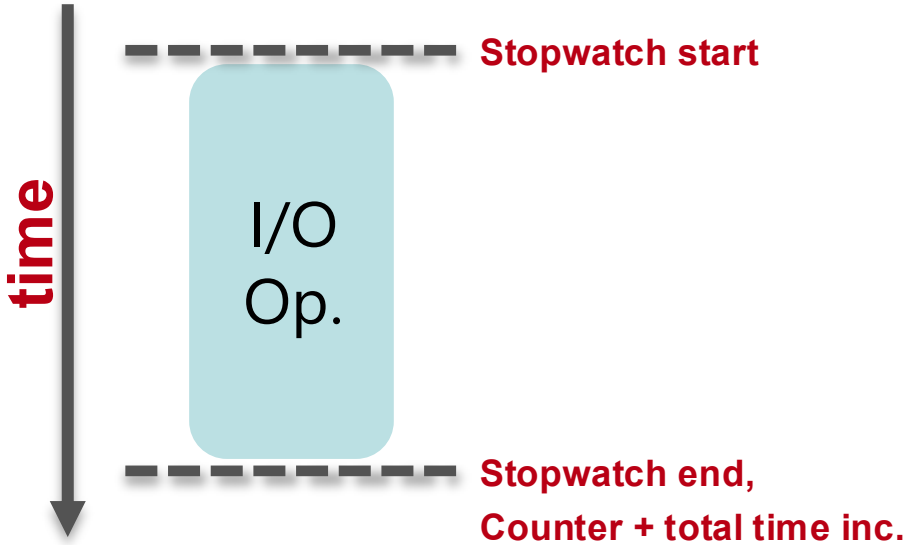


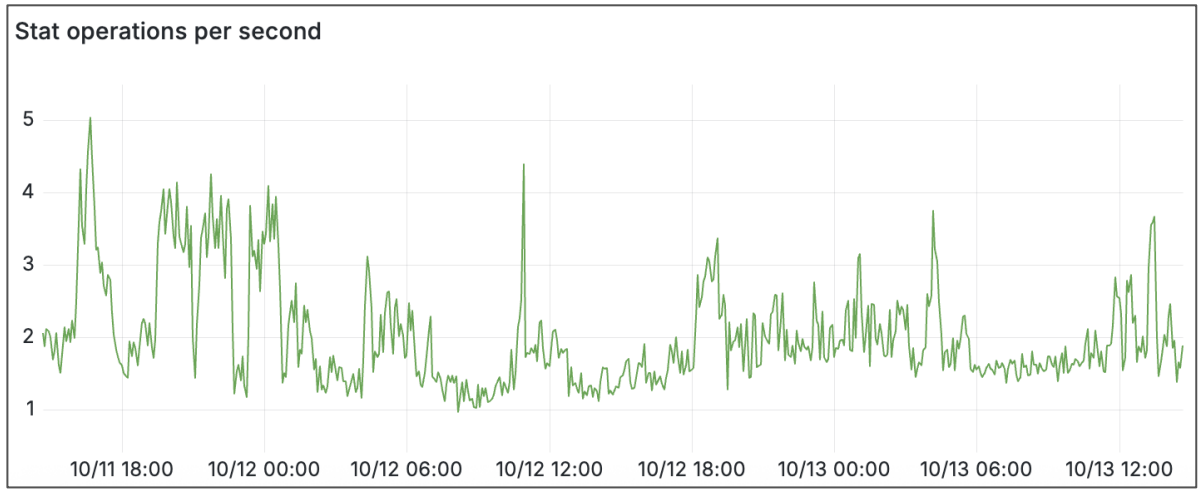
OSS: Better Behavior via Stacking

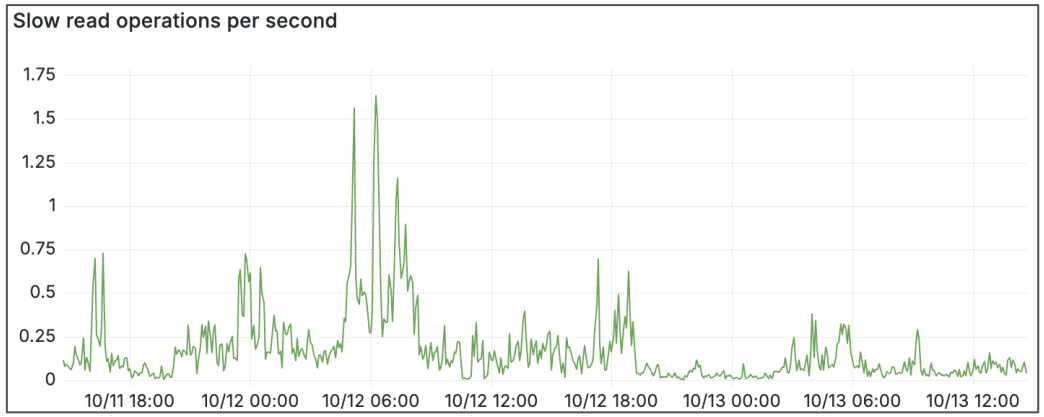
- Not all OSS plugins are about a unique storage backend!
- Some OSS plugins are *stackable*:
  - They provide additional functionality to existing plugins, not complete solutions in themselves.
- Example:
  - (2025) The XrdOssStats plugin: it adds timing / count information, not a storage system.
  - (2025) The XrdOssThrottle plugin: slows down requests (refuses to pass them to the next in the stack) to penalize overly-aggressive individual users. Goal is “fairshare” of I/O time.

XrdOssStats

- Next – how much time is being spent on specific filesystem operations?
  - How often do users experience “slowness” from the filesystem?
- The (v5.9.0) XrdOssStats plugin wraps around all I/O operations.
  - Maintains a counter of # of operations and total I/O time.
  - If the operation lasts longer than an admin-provided threshold (2.0s by default), a separate set of counters is incremented.





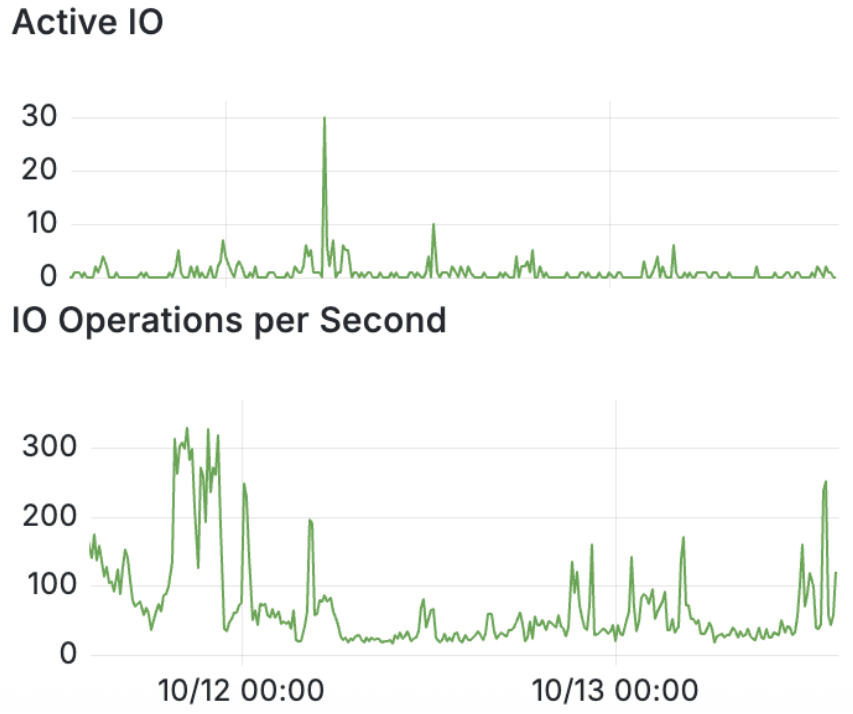


(About 1% of total for this host)

Xrd(Oss)Throttle

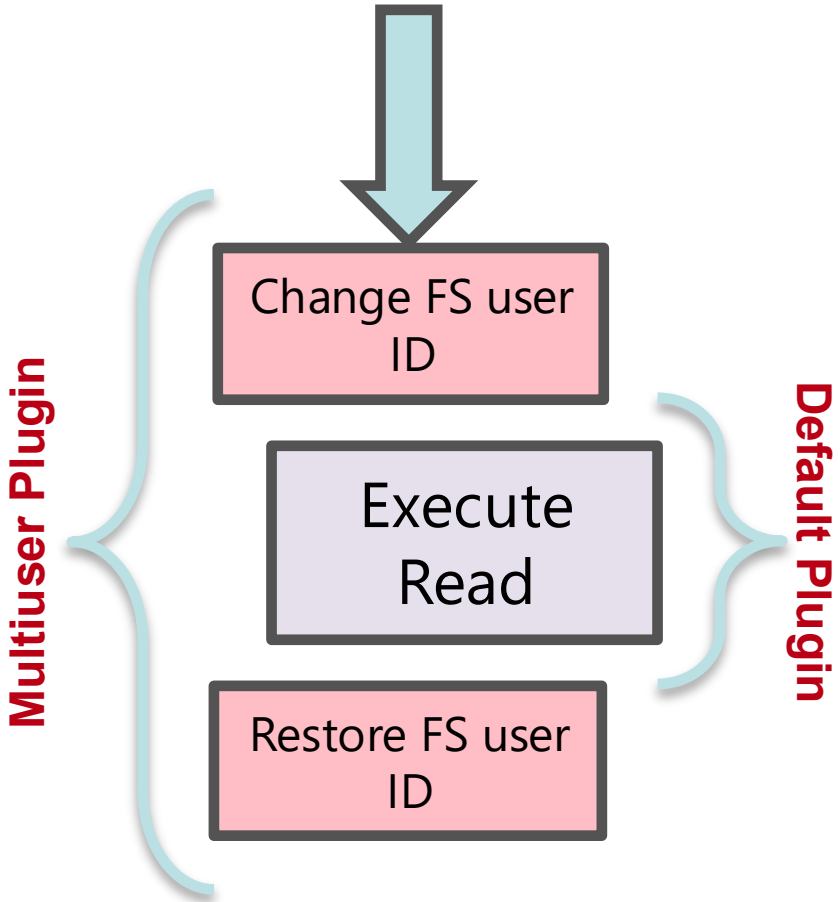
- The XrdThrottle library tracks and limits:
  - **Bandwidth**: bytes moved in/out of the server. (Somewhat useful if you know the server has network capacity limits)
    - Better: Use the host's network layer to manage the network.
  - **I/O Operations per second**: how many read/write operations complete (Mostly useless due to page cache)
  - **I/O load**: how much time the server is spending in the I/O layer (very useful, tracks well with degraded performance).
- Highlights in 2025 (XRootD 5.9.0):
  - Servers over-limit of I/O load will throttle according to fairshare: users are delayed in proportion to the load they are causing. Users that "aren't the problem" see no delay due to throttles.
    - Must be switched to OSS mode: only OSS layer knows the "username" for a request.
  - I/O statistics (IOPS, bandwidth, load) are reported via the g-stream.
    - Does **not** require the operation to finish. An operation taking 2-hours to complete will be reported while it is active.

Example Throttle Data; (Denver Xcache)



Stackable plugins - Multiuser

- Another example is the multiuser plugin:
  - The multiuser plugin extracts a username from the filesystem request, maps it to a Unix UID, and switches the active filesystem UID.
  - It then calls the underlying plugin to handle the I/O.
  - Once complete, it'll restore the FS UID to the xrootd user.
- Additionally, the plugin computes checksums on the fly when writing a new file.
  - Must do similar UID switch for accessing xattr's.
  - This functionality is an OSS-specific version of the new streaming checksum creation.



Stackable plugins - Filter Plugin

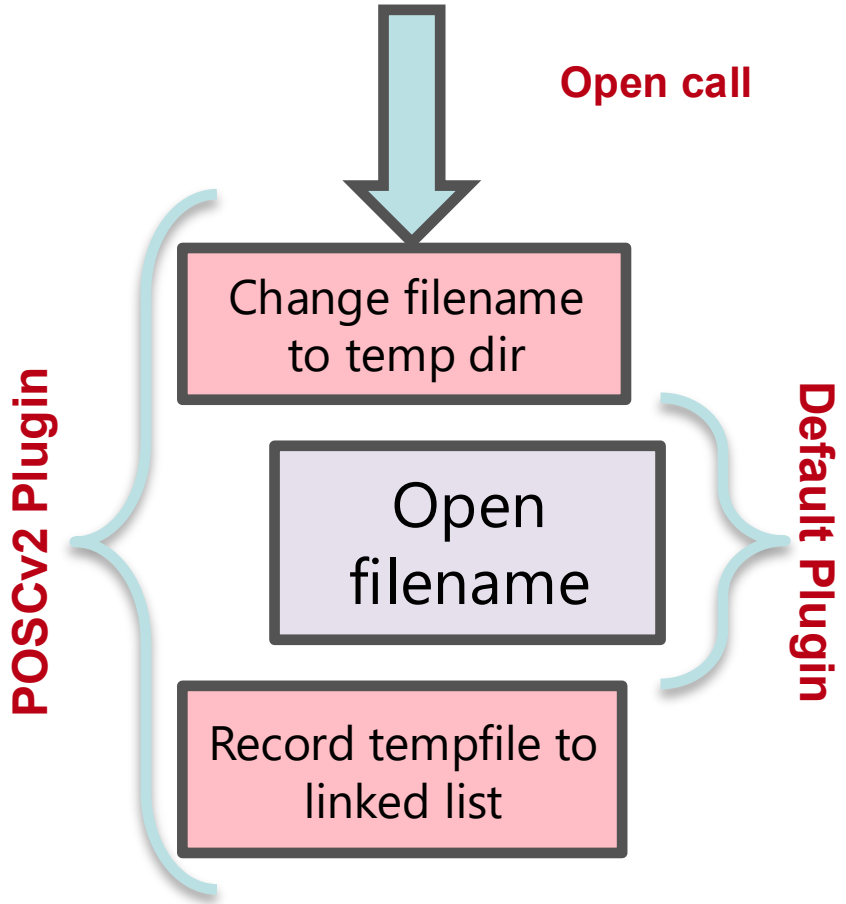
- Deceptively simple case:
  - "I want to export /store/**/dataset* and only make matching files visible"
  - It's not "I want to deny read access" but really "**make invisible**".
- The "Filter" plugin provides more fine-grained control over what's exported beyond just prefix as in all.export.
- To use:

```
ofs.osslib ++ libXrdOssFilter.so
filter.glob /store/**/dataset*
```

- Follows all the expected "Unix glob" rules; uses glibc's implementation plus custom code for bash-style globstar / ** semantics.
- Useful to ensure no "dot files" (.*) are accidentally exported to the world.
- Works at the XRootD layer: if a user has POSIX access and can create a symlink, then they can make anything visible!

Stackable plugins - POSC Plugin

- Playing with a new twist on an old friend – “persist on successful close”.
- **Challenge**: If a file is visible in the origin, then it can be accessed by XCache.
  - The second it’s visible to XCache, the cache may have a permanent copy of the incomplete file.
  - In general, **can we make partial files “invisible” until the upload is complete?**
- **POSCv2** plugin: Using OSS, *only make visible* once the file is fully uploaded.
- **General idea**: have dedicated temporary directory; store files there; remove directory from storage listings (Listdir) or stats.



Stackable plugins – POSCv2 Plugin

- The POSCv2 plugin:
  - Writes all new files to a designated temporary directory (makes said directory invisible to XRootD).
  - On Close(), atomically renames file to the destination.
  - **Upshot**: Only files successfully uploaded are visible in the namespace.
- Can be combined with the multiuser plugin; temporary files are stored in /$(posc.prefix)/$(username)/in-progress.$(timestamp).$(random).
- Periodically scans directory to discover + cleanup old files that have been abandoned.
  - Periodically "touches" this temp file to keepalive.
  - This prevents us from needing a separate journal to persist what is in-progress across crashes.
  - Similarly, allows multiple XRootD servers to share the same filesystem with POSCv2.

```
ofs.osslib ++ libXrdOssPosc.so
posc.prefix /in-progress-dir
```

Stackable plugins – Deadlock Detection

- **A bug in the OSS plugin can be catastrophic**.
- If an Open request deadlocks, **then all requests for that file deadlock**.
  - Incredibly hard to detect: shows up as a specific file that stops working and a slow-but-steady increment of threads. Humans can spot the thread-growth pattern, difficult to see programmatically.
- For the cache, an OSS request passes through: XrdPss, XrdPosix, XrdPfc, XrdOucCache IO, XrdCl, XrdClPelican, XrdClHttp, and libcurl.
  - Every layer must be 100% bug / deadlock free or user-visible timeouts occur.
- Idea:
  - Every Open request is noted on a linked list.
  - **Speed matters**, hotpath: list entry is on the stack, avoiding an allocation; per-CPU list/mutex, avoiding contention.
  - Periodically, a thread will walk each CPU's list.
  - If a file takes longer than N seconds to open, assume it is a bug – shutdown immediately!
    - **Example**: if the HTTP HEAD timeout is 90 seconds, assume a deadlock has occurred for any open request that is at 180 seconds.
    - More difficult to use for "real" POSIX filesystems – storage can be arbitrarily slow!

```
ofs.osslib ++ libXrdOssDeadlock.so
deadlock.timeout <seconds>
deadlock.logfile <path>
```

XrdHttp Plugin: Prestage & Eviction API

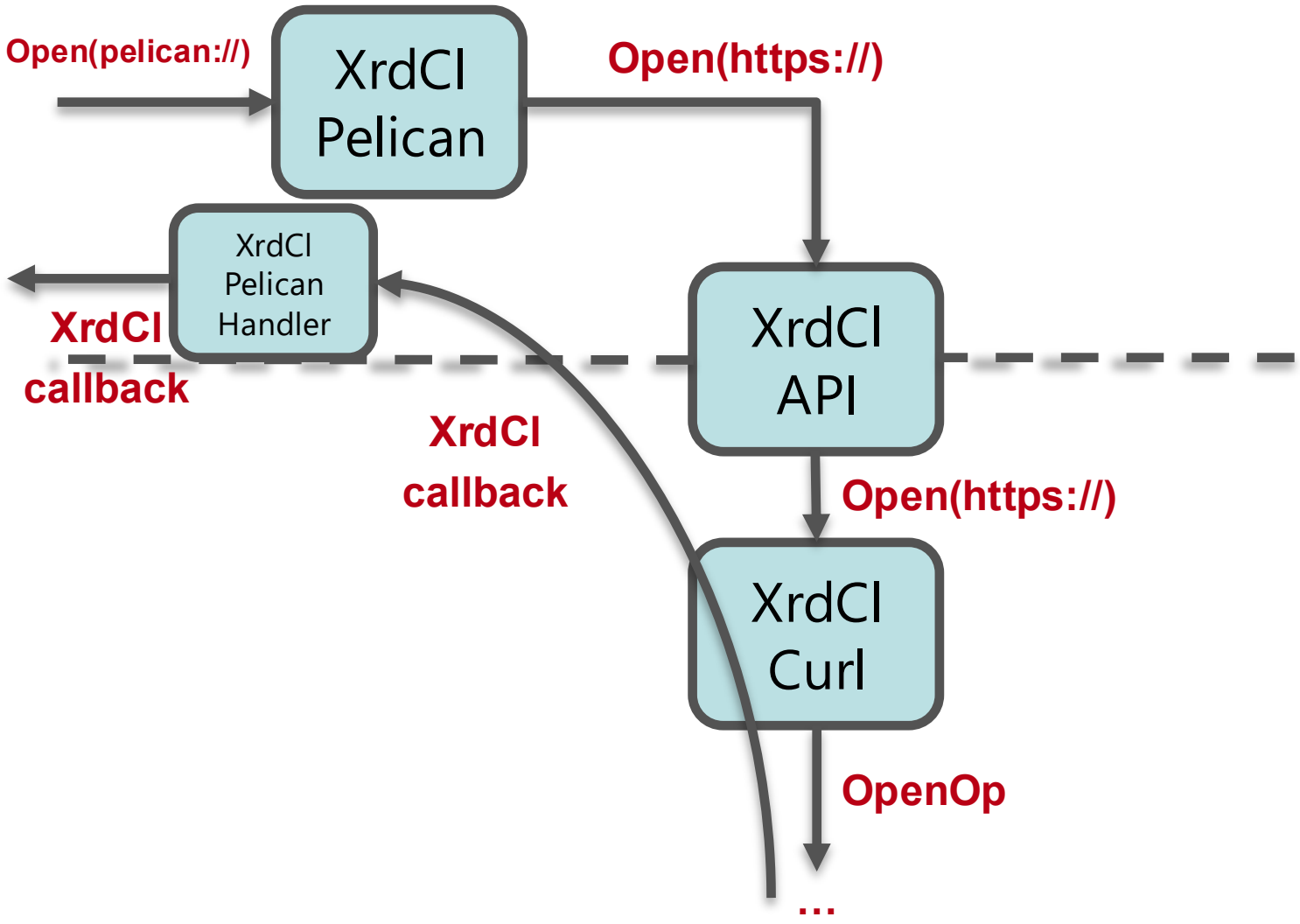
- Setting “http.exthandler xrdpelican libXrdHttpPelican.so” enables our prestage/eviction REST API for the cache.
- Prestage: GET /pelican/api/v1.0/prestage?path=<absolute_path>
  - Reads the file into the cache but doesn’t deliver it to the client. Forces a local copy.
  - **Per-User Queueing**: Prestage requests are queued per user/VO identity to ensure fair resource allocation
  - **Progress Tracking**: Prestage operations report progress every 200ms or when offset changes significantly
  - **Backpressure**: When queues are full, new requests receive a 429 status code
- Eviction: GET /pelican/api/v1.0/evict?path=<absolute_path>.
  - Removes the file from the cache via the eviction SFS_FSCTL_PLUGXC.
- Both integrate with the XRootD authorization subsystem:
  - Prestage requires READ access to the file.
  - Eviction requires DELETE access to the file.

XrdHttp Plugin: Various and Sundry

- The Pelican XrdHttp plugin has become our “abuse the plugin to start an arbitrary worker thread”.
  - Whenever we need functionality that doesn’t fit anywhere else and doesn’t interact with the rest of XRootD, it gets shunted to a maintenance thread in this plugin.
- Examples:
  - **Drop-privilege help**: In drop priv mode, XRootD and Pelican run as Unix user xrootd and pelican, respectively.
    - `pelican` does not have permission to send signals to `xrootd`. `xrootd` does not have permission to read the X.509 private key of `pelican`.
    - **Solution**: pass file descriptors (and other commands like signals) through a Unix domain socket read by worker thread in XRootD. Allows Pelican process full control, even when it cannot do it via Unix perms.
  - **Stack trace signal handler**: Corefiles / coredumps are *unfortunately rare* in containers. Just to have something, we install a signal handler that will create a signal-safe stack trace of the thread that caused the crash.
    - Does addr2line and friends to print out actual symbol names, not just memory locations.

XrdCIPelican v2: A Layered Approach

- As presented last year, XrdCIPelican has been **completely refactored** to work on top of the https:// handler for XrdCl.
  - Hurray!
- For v5.9, we still use the forked XrdCIHttp library.
  - Periodically, review commits and backport bugfixes.
- For v6.0, we should jump onto upstream!
- Biggest out-of-tree changes:
  - Canceling a prefetch GET no longer holds onto a transfer slot for 30s.
  - Add ability to retry specific idempotent requests for known error codes.
  - Fix redirect loop.
  - Pass Cache-Control / Etag info from response to cache.
  - Use new XRootD 6 error message API to pass libcurl error message to HTTP error message.
  - Prevent single host from consuming all possible transfer slots (slow hosts can no longer bring down a cache).



XRootD Plugins & Pelican - Summary

- XRootD has pluggable policies across its service – and some can even stack!
- Pelican heavily uses plugins to add storage backends, provide Pelican-specific APIs, and to modify behavior.
  - Perhaps second only to EOS in level of customization?
- Plugin capability allows us to develop/work on ideas outside XRootD proper, then submit them upstream, **without having to fork XRootD**.
- Plugin capability allows us to have Pelican-specific functionality without bothering upstream, **without having to fork XRootD**.