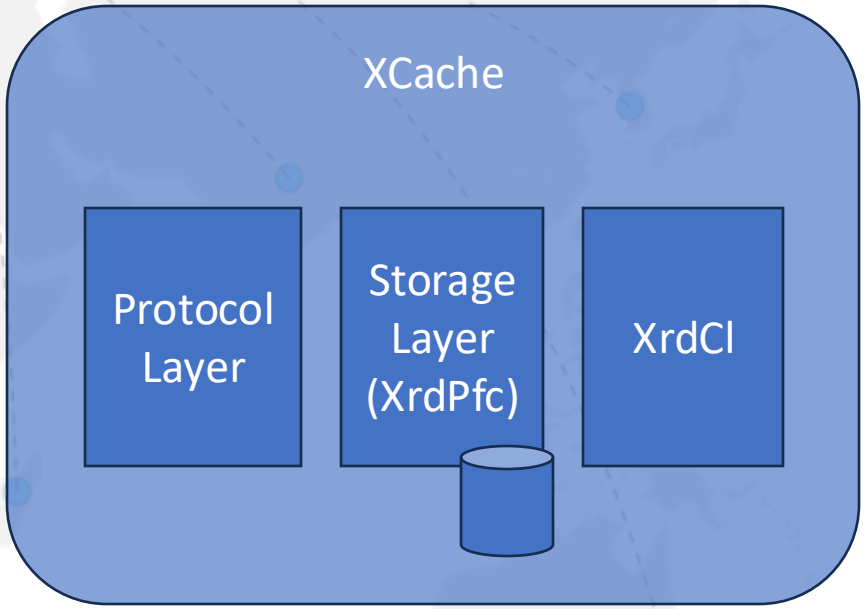
Making XCache Do What you Want

Writing XrdCl Plugins for fun and profit

XCache and XrdCl

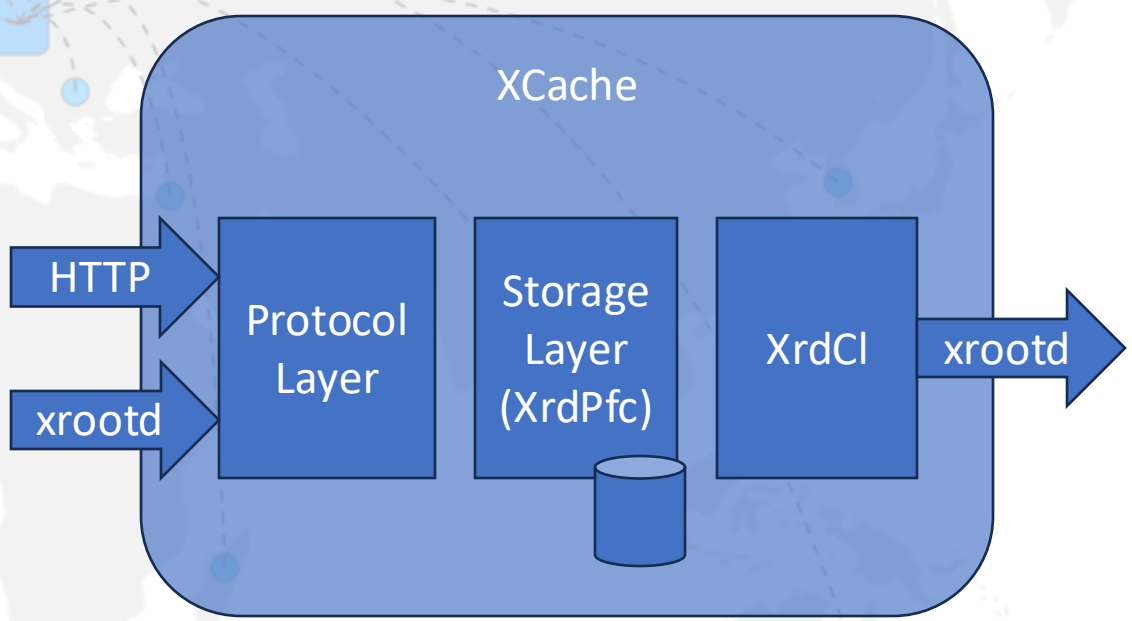
- XCache is the ‘caching proxy’ mode for XRootD.
- The cache itself is implemented as the “XrdPfc” (Proxy File Cache), a plugin for the XRootD storage layer.
  - Really is “just” a storage plugin like POSIX, S3, or HTTP backends.
  - Additional plugin layers expose the cache functionality.



A highly abstracted view of what’s inside an xrootd daemon when configured in XCache mode.

XCache and XrdCl

- When a request comes in, XrdPfc determines if the requested data is available locally.
  - If local, it's served from storage.
- On a cache miss, XrdPfc needs to download the missing data.
- How? Use the XrdCl client!
  - Great synergy: XCache uses the exact same client library underlying the venerable xrdcp.



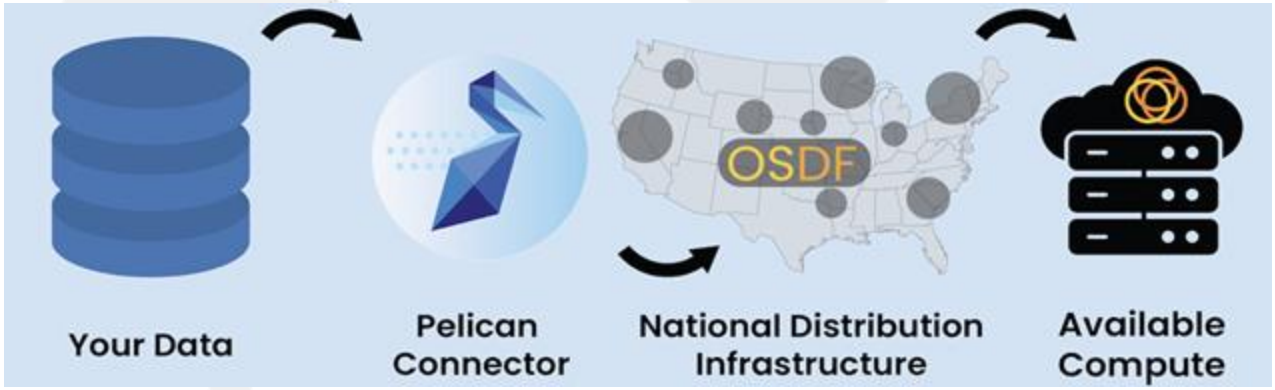
A highly abstracted view of what's inside an xrootd daemon when configured in XCache mode.

Pelican Reminder

The Pelican software powers the OSDF, a shared scientific data distribution infrastructure. A “CDN for science”.

Pelican relies on XCache to do the actual data movement.

- Hence, we need to ensure XCache fits well into the Pelican architecture.



Using XCache in Pelican

Pelican has some challenging aspects for XCache:

- All components speak HTTP, not necessarily XRootD.
  - (Particularly the director)
- Some origins are behind a firewall – no incoming connectivity.
- Pass through client-provided timeouts.

Q: How do we make XrdCl do all this “stuff”?
A: Use the XrdCl plugin mechanism!

The XrdCl Plugin

- One can provide a shared library that re-implements the XrdCl C++ API.
- For registered URLs, the shared library object is called instead of the default XrdCl code.
- Provided the implementation obeys the API “contract”, the shared library can have any desired behavior.
  - Let’s write a plugin for Pelican!

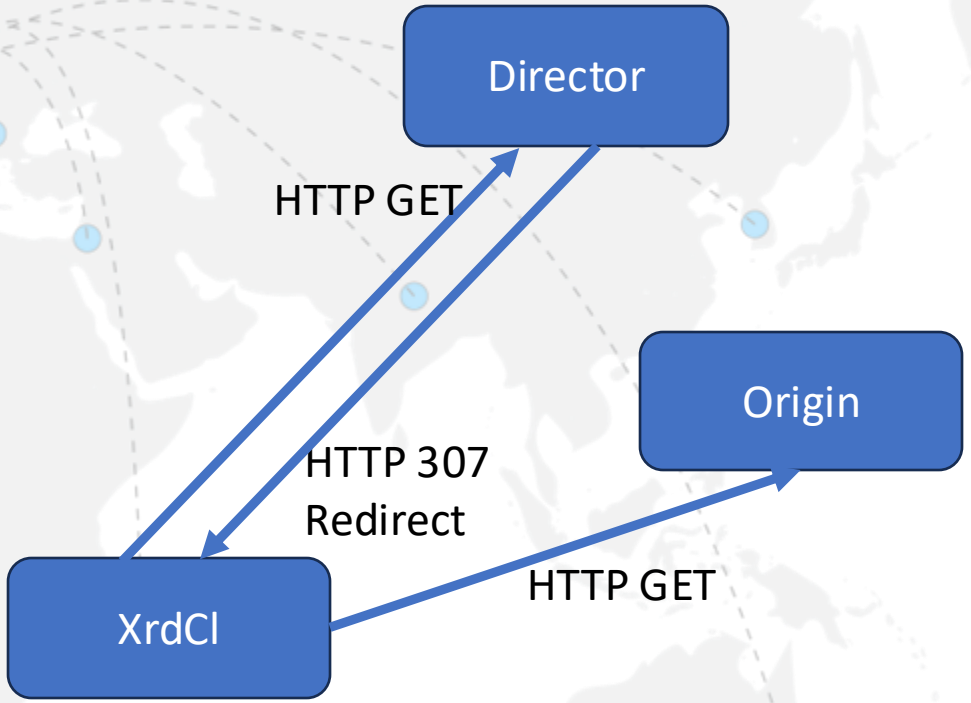
Example plugin configuration file contents:

```
url = pelican://*
lib = libXrdClPelican.dylib
enable = true
```

The above has libXrdClPelican.dylib used for any pelican:// URL.

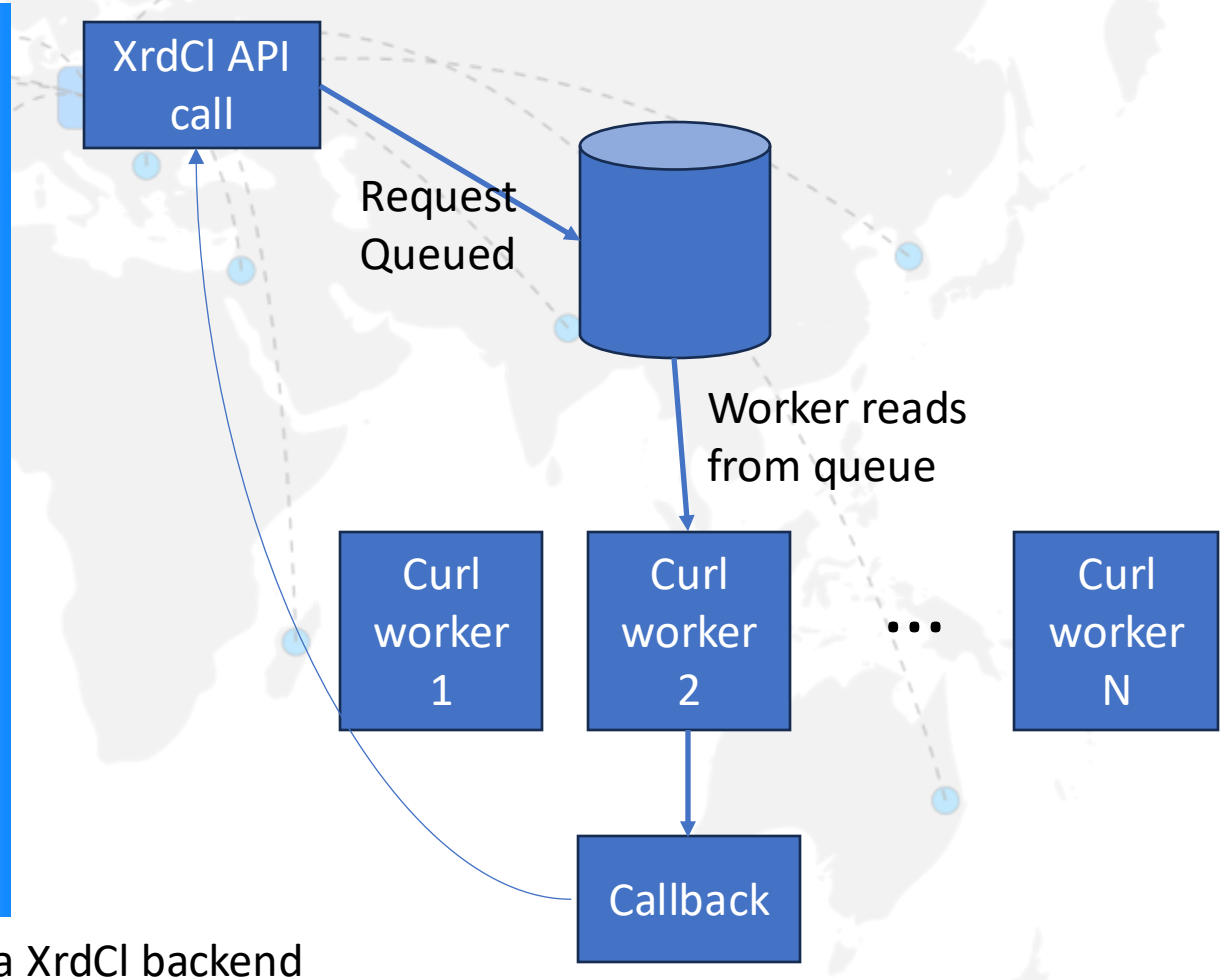
Pelican-izing XCache “Phase 1”

- Started by having the plugin invoke libcurl for file operations:
  - Open => HTTP GET sent to director, gets redirect to the origin.
  - Read => GET sent to origin
  - Stat => HEAD
  - Write => PUT (but not currently implemented)
- There’s a similar plugin, libXrdClHttp, that invokes Davix.



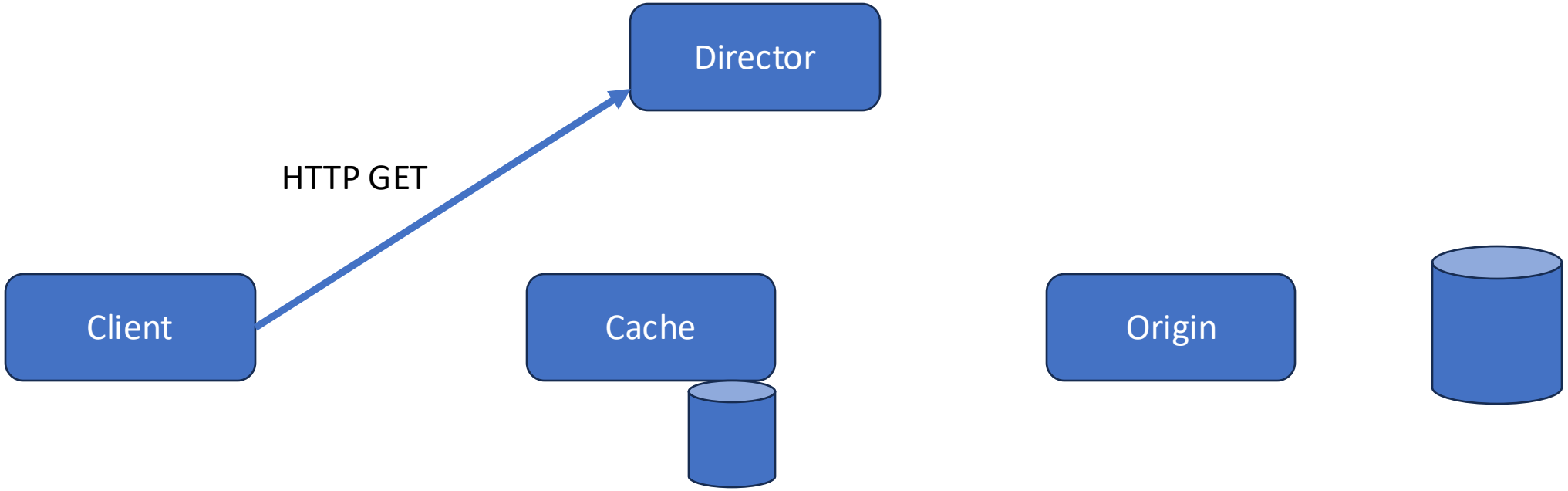
Pelican-izing XCache “Phase 1”

- Invoking libcurl directly reduces dependencies.
- We use the libcurl “multi” interface to have a single thread handle many transfers.
- By default, 5 worker threads
  - Concurrency is configurable
- When completed, the worker thread invokes the appropriate callback.
  - Hugely benefits from the fact XrdCl is asynchronous/callback-based.

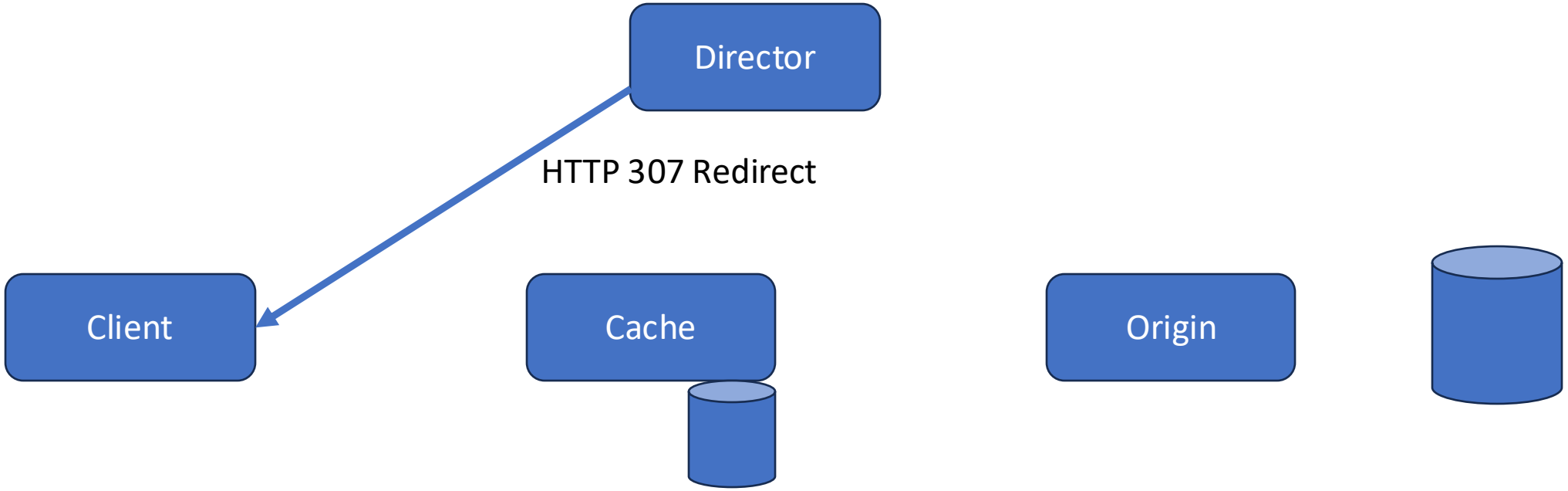


Great, we have a XrdCl backend for HTTP. Now what?

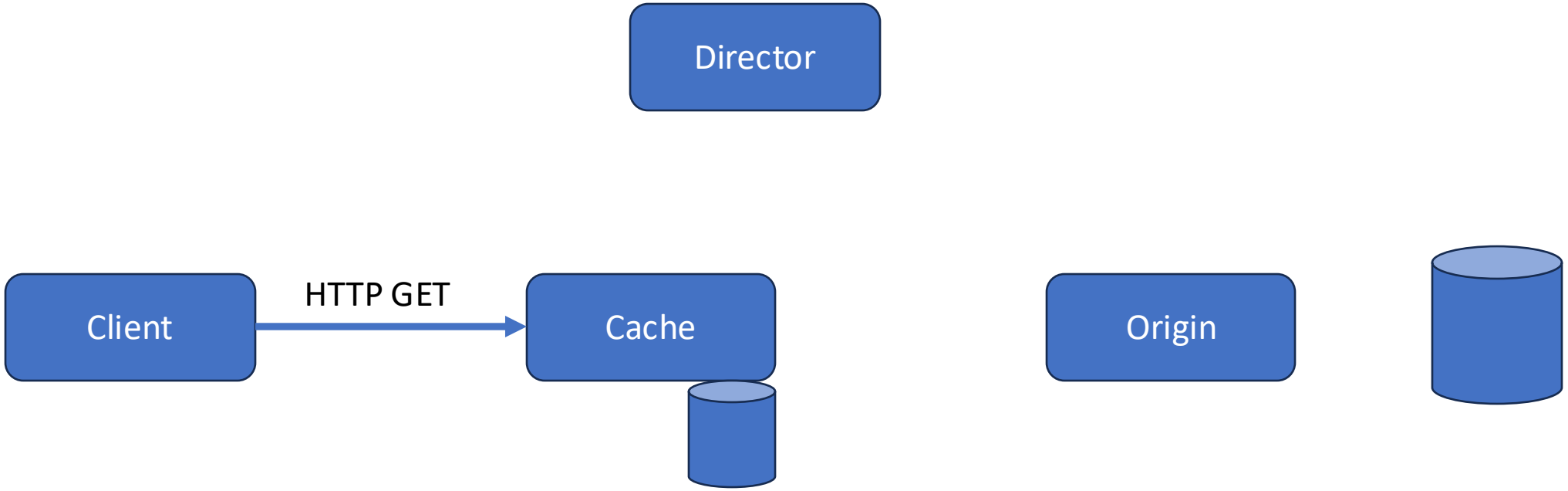
Plugin data flow



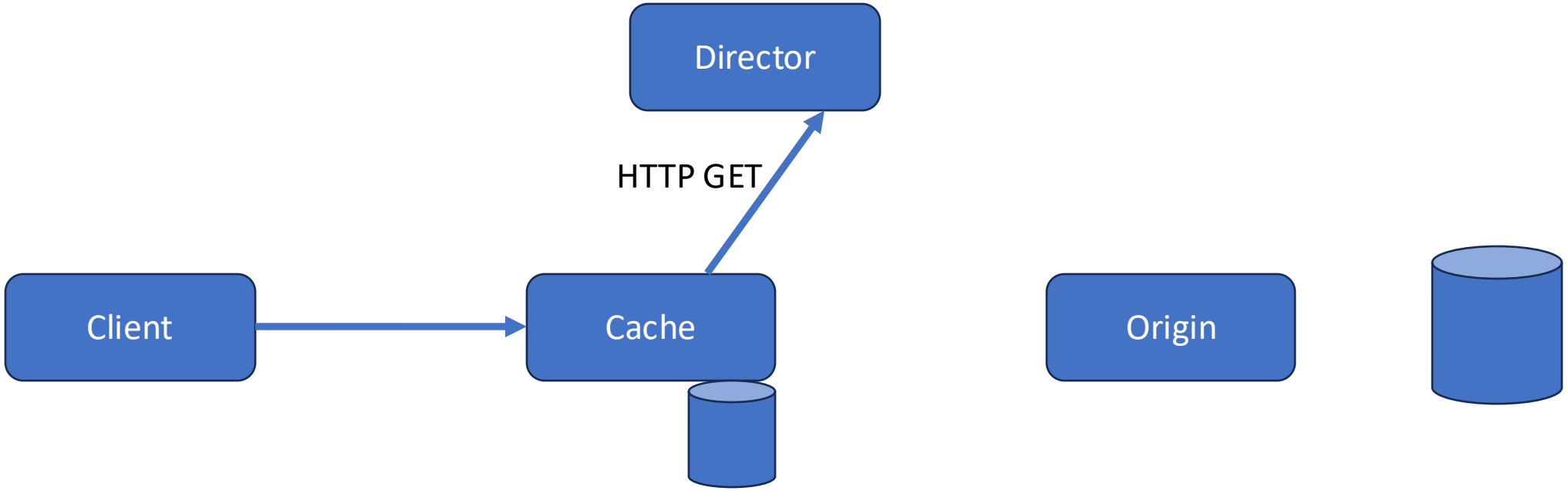
Plugin data flow



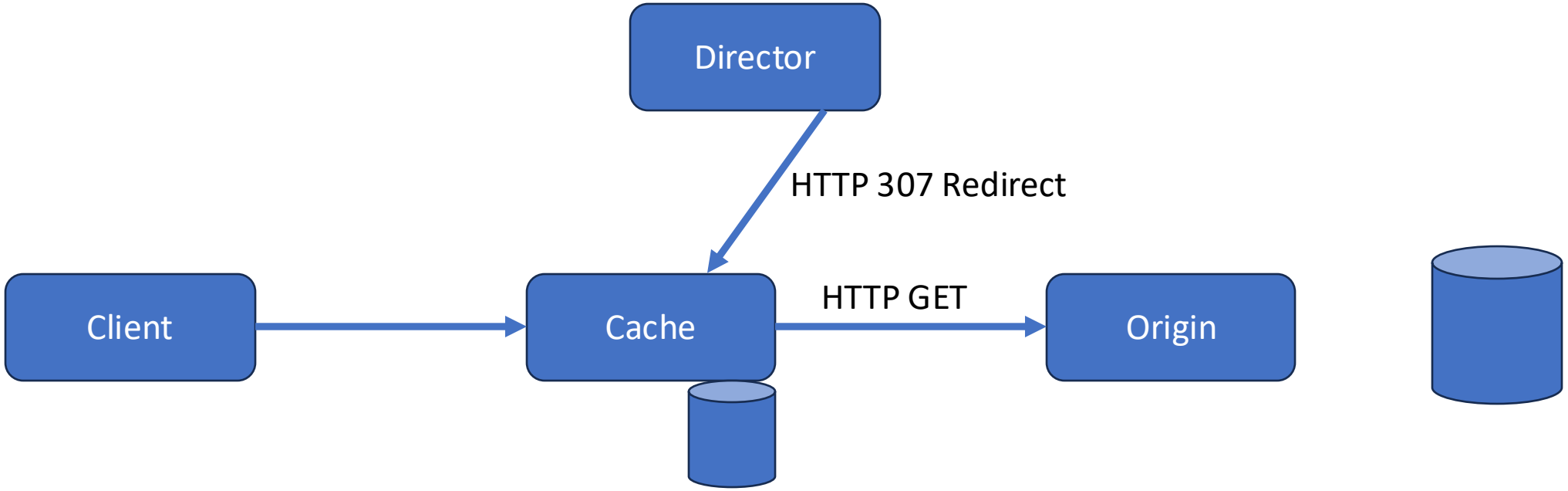
Plugin data flow



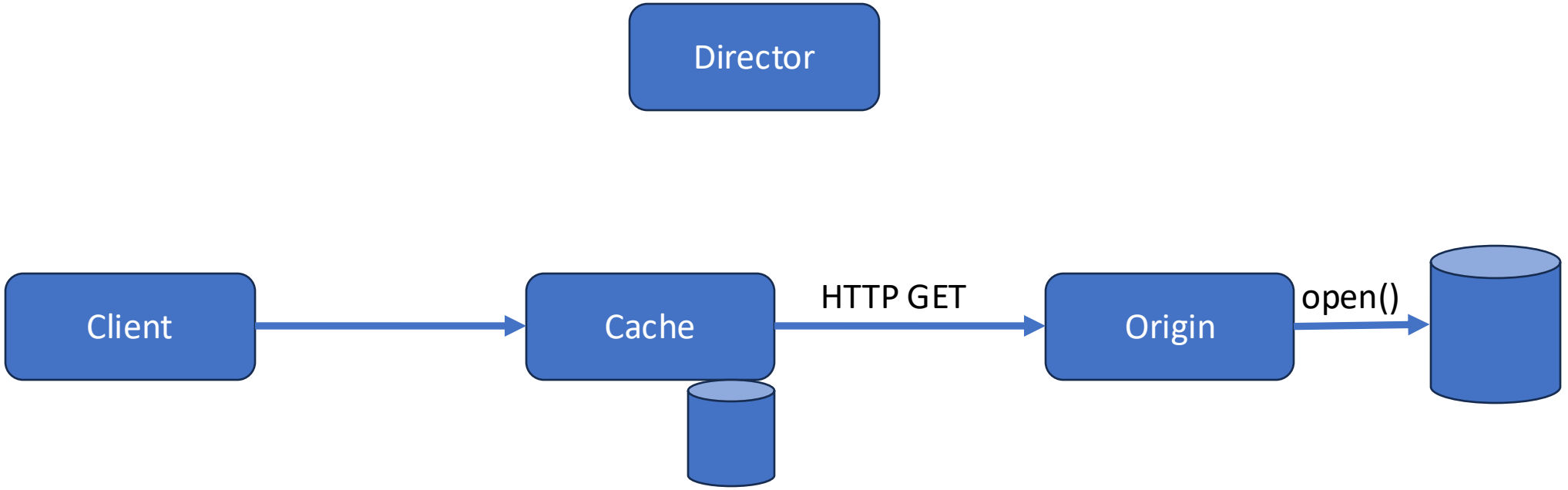
Plugin data flow



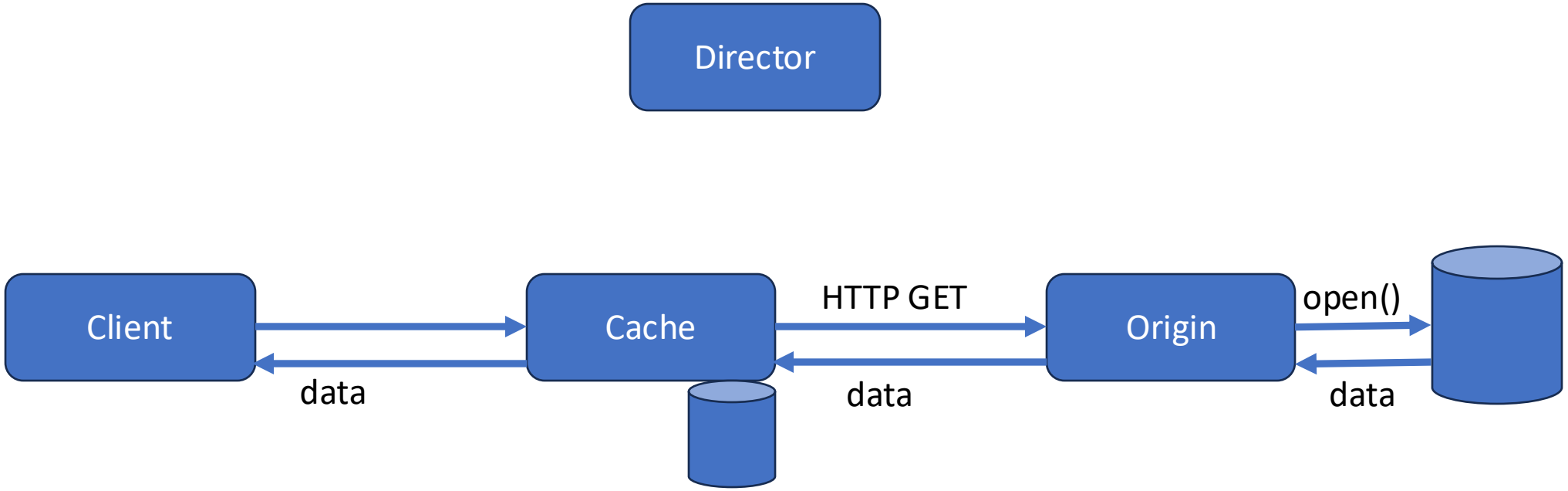
Plugin data flow



Plugin data flow



Plugin data flow



Note that the protocol between the client, cache, director, and origin is based on HTTP.

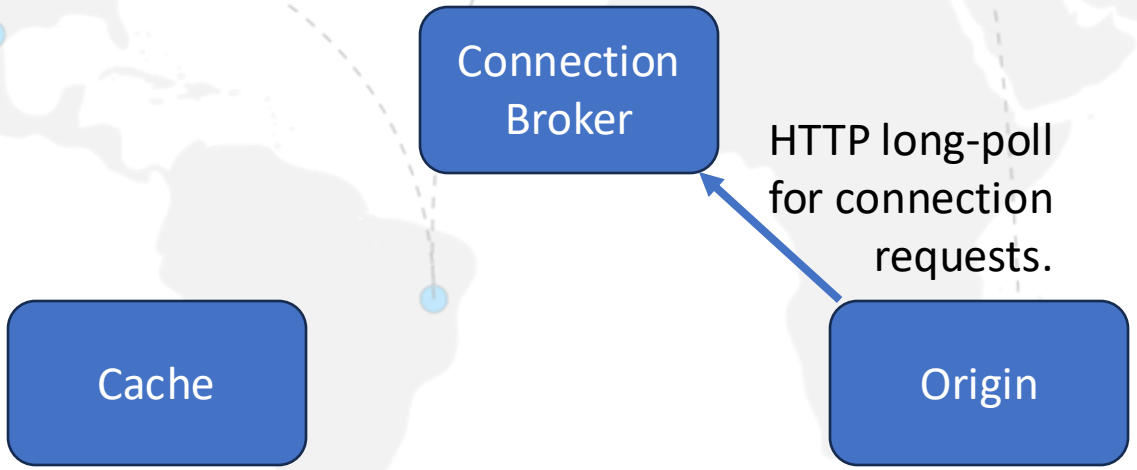
Next step – origins behind firewalls

- We want scientists to easily setup their own origin.
- Aspirational Goal:
  “I have a server in my lab and I want to share the data with my collaborators via OSDF”
- Aspirational Goal:
  Anyone with enough technical knowledge to setup a WiFi router can setup a Pelican origin.
- The average scientist doesn’t know how to:
  - Get a public IPv4 address for their lab.
  - Open an incoming port in the university firewall.
  - Setup a DNS entry.
  - Get a host certificate for their endpoint.

Get rid of these requirements!

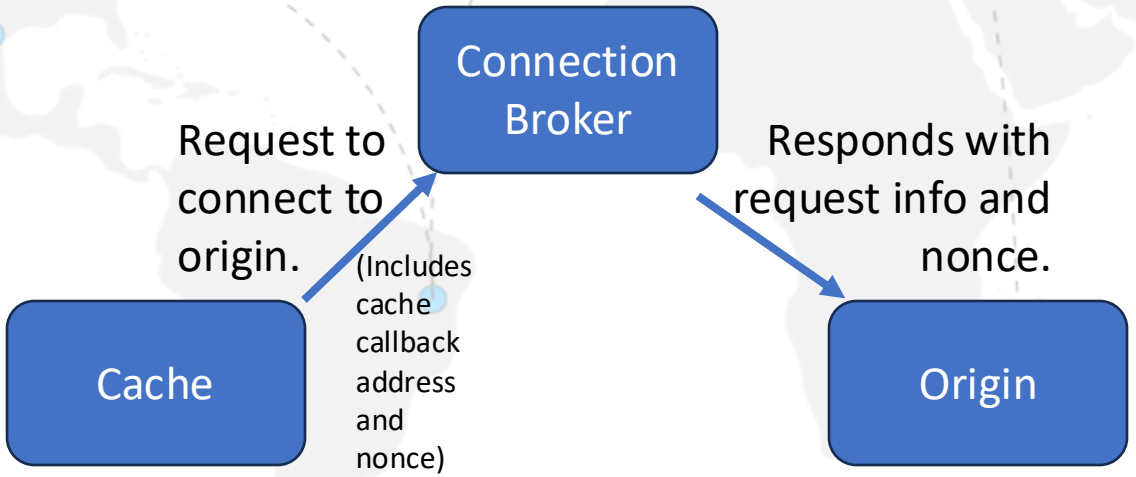
Connection Reversing

- Q: How do you host an origin without incoming connectivity?
  A: Connection Reversing!



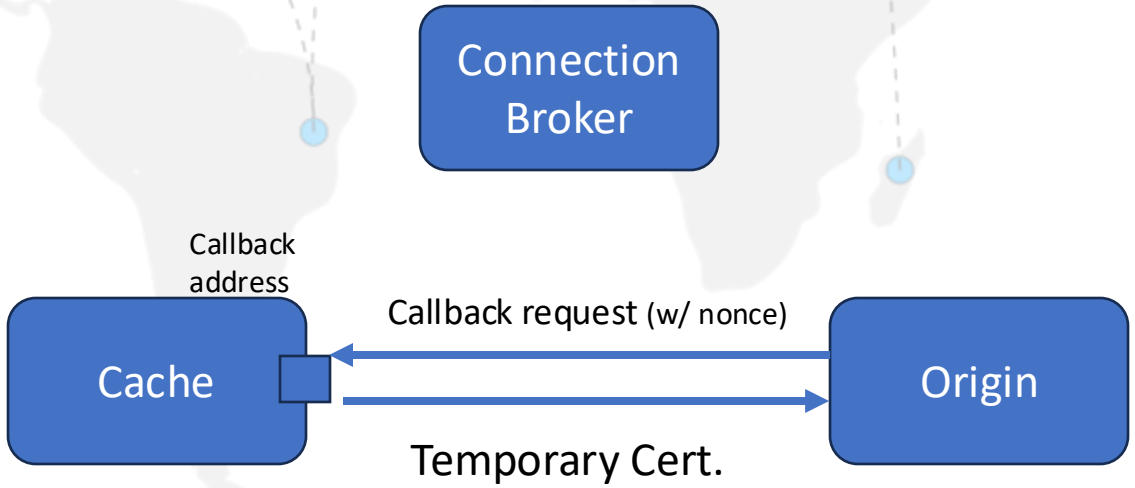
Connection Reversing

- Broker is the central agent to turn around requests.
  - Origin's outgoing poll either times out or retrieves a request.



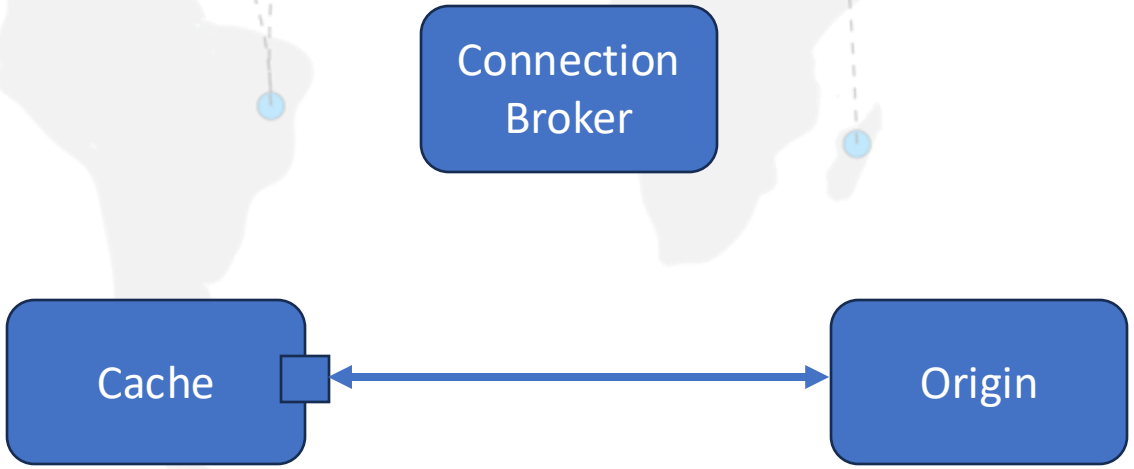
Connection Reversing

- The origin opens an outgoing connection to the cache.
  - To authenticate the request, sends the nonce it received from the broker.
  - The cache will generate a temporary host certificate and send it to the origin.



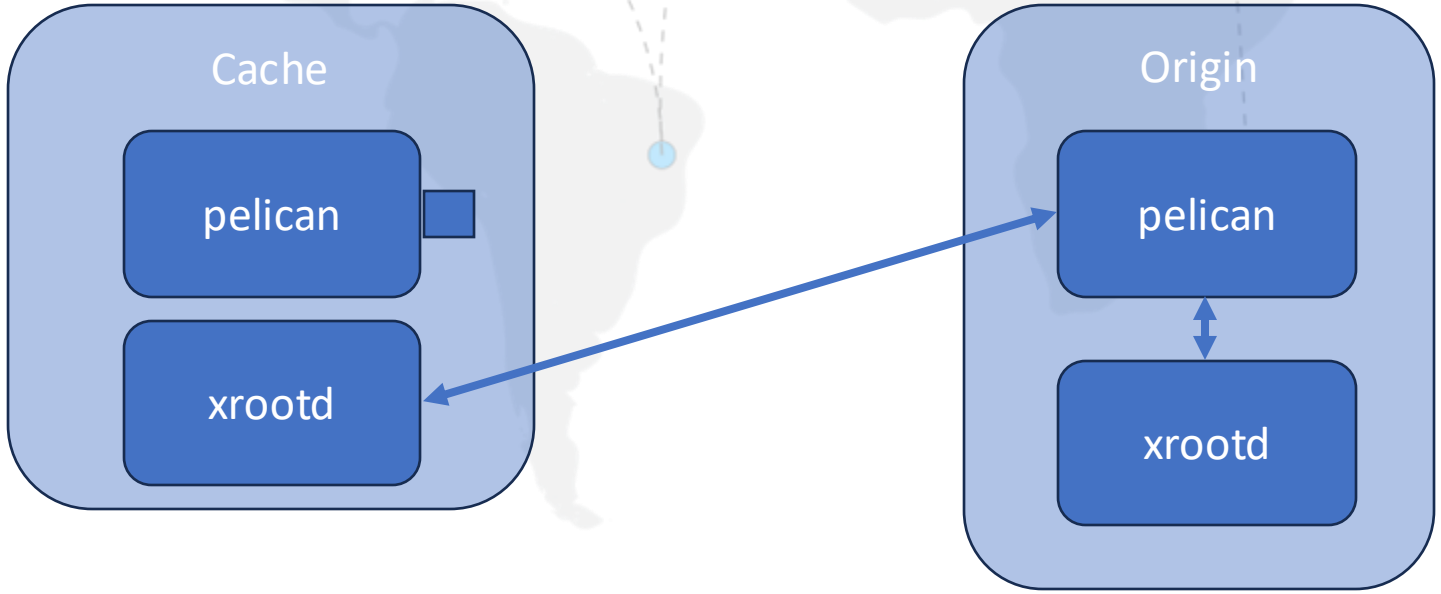
Connection Reversing

- Once the callback HTTP request has succeeded, we take advantage of the fact that an established TCP connection is bidirectional.
  - No difference between client and server!
  - Pretend it's a fresh connection and just treat it in reverse.
- The origin will use the temporary certificate it received (signed by the cache's self-signed CA) for future communication from the cache.



Connection Reversing

- Cache: Use Unix file descriptor passing to send the TCP connection from pelican to xrootd.
- Origin: Proxy requests from Pelican to xrootd.
- XrdCl plugin uses the TCP connection “normally” for HTTPS.



Connection Reversing – Putting it Together

- The XrdCl plugin gets a request for data.
- XrdCl plugin contacts the director, told the origin needs a connection reverse.
- XrdCl prepares and invokes a libcurl request.
- If libcurl doesn't have an existing TCP connection, it'll callback to the plugin to provide one.
  - The plugin will request a TCP connection from the pelican process.
  - Pelican will do the connection reversing and return a file descriptor.
  - The plugin returns the connected TCP socket to libcurl.
- Libcurl continues the request with the connected socket.

Handling client timeouts

- Q: Who is to blame for an unresponsive cache?
  - Is the cache overloaded or deadlocked?
  - Or is the cache fine and the origin unresponsive?
- Understanding if it's an "origin problem" or a "cache problem" greatly influences debugging and retry policies.
  - If it's an origin problem, then retrying at a different cache will likely only contribute to origin overload!
- **Idea**: Client should send, in its request, when it'll give up on the service.
  - If the cache is responsive but waiting on the origin, timeout the origin request and respond with "bad origin".

Request timeouts

- Client sets “X-Pelican-Timeout” header, e.g.
  - X-Pelican-Timeout: 10s
- Timeout is copied from header to the XRootD “cgi”:
  - /foo/bar?X-Pelican-Timeout=10s
- The XrdCl plugin gets the XRootD string and parses out the timeout
  - /foo/bar
  - Timeout: 10s
- The plugin will cancel the libcurl request prior to the timeout.
  - When timed out, the cache will return HTTP’s “502 Bad Gateway”.

Conclusions

- Pelican wants lots of custom behavior, different from the default XRootD / XCache experience.
  - But don't want to re-implement XCache! Many, many reasons to keep this component.
  - XrdCl plugins allow us to do keep XCache but also mix in our desired behavior.
- The existing plugin allows use of HTTP origins and connection-reversing.
- **Food for thought**: what's the difference between a XrdCl plugin that speaks HTTP and an OSS plugin that speaks HTTP?

See https://github.com/PelicanPlatform/xrdcl-pelican - have fun!

Questions?

This project is supported by the National Science Foundation under Cooperative Agreements OAC-2331480. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.