Pelican under the hood: how the data federation works

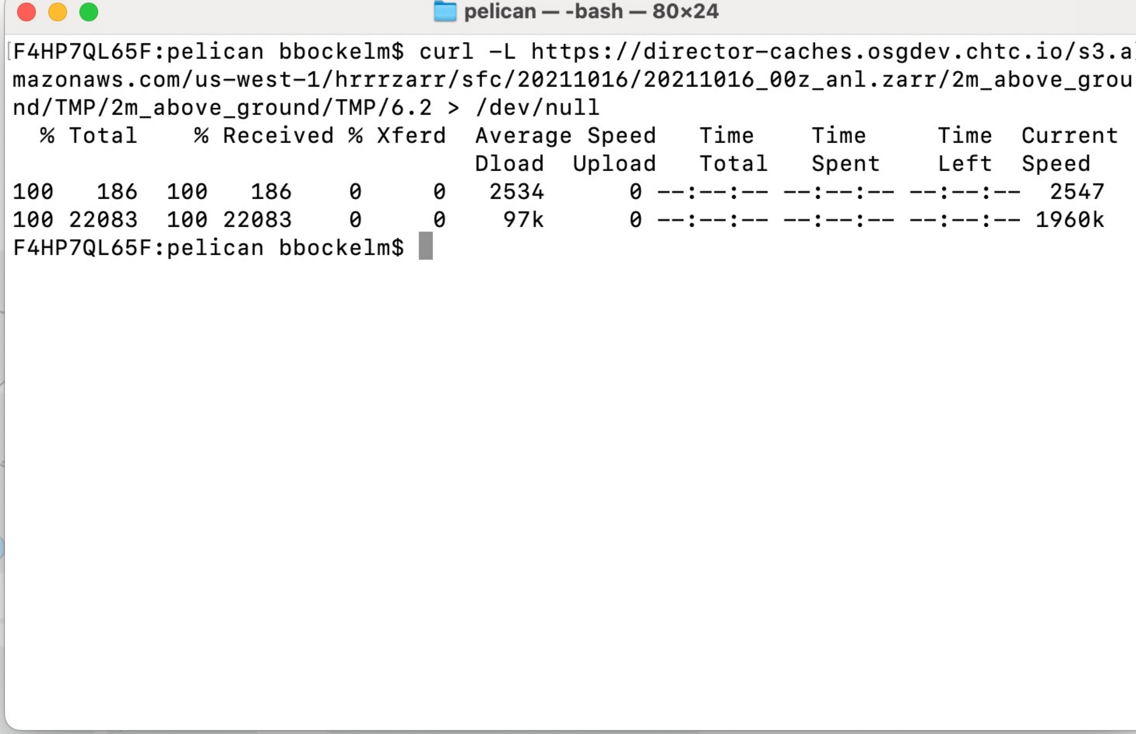
If only I had a whiteboard…

- … I could talk and draw for hours about how the system works.
- So today I'll pick three topics:
  - How we use HTTP under the hood in the client, cache, and origin.
  - How we "authorize" an origin to the director.
  - Authorizing the origin to the object store.

HTTP, HTTP Everywhere

Pelican uses HTTP

- Pelican uses HTTP to move bytes*.
- We hew to using standard HTTP where possible. While we *prefer* you use the Pelican client, any HTTP client suffices.
  - Downloading an object? => GET
  - Uploading an object? => PUT
  - Want to know if the object exists? => HEAD



```
F4HP7QL65F:pelican bbockelm$ curl -L https://director-caches.osgdev.chtc.io/s3.a
mazonaws.com/us-west-1/hrrrzarr/sfc/20211016/20211016_00z_anl.zarr/2m_above_grou
nd/TMP/2m_above_ground/TMP/6.2 > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   186  100   186    0     0   2534      0 --:--:-- --:--:-- --:--:--  2547
100 22083  100 22083    0     0    97k      0 --:--:-- --:--:-- --:--:-- 1960k
F4HP7QL65F:pelican bbockelm$ 
```

* Except it where it doesn’t: legacy services still transitioning.

Example request from client to director

```
> GET /chtc/staging/bbockelm/testfile HTTP/2
> Host: osdf-director.osg-htc.org
> User-Agent: curl/8.4.0
> Accept: */*
```

Example director response

```
< HTTP/2 307
< content-type: text/html; charset=utf-8
< date: Mon, 08 Jul 2024 17:17:17 GMT
< link: <https://osdf-uw-cache.svc.osg-
htc.org:8443/chtc/staging/bbockelm/testfile>; rel="duplicate"; pri=1;
depth=3, <https://stash-
cache.osg.chtc.io:8443/chtc/staging/bbockelm/testfile>; rel="duplicate";
pri=2; depth=3,...
< location: https://osdf-uw-cache.svc.osg-
htc.org:8443/chtc/staging/bbockelm/testfile
< x-pelican-authorization: issuer=https://chtc.cs.wisc.edu
< x-pelican-namespace: namespace=/chtc, require-token=true, collections-
url=https://origin-auth2000.chtc.wisc.edu:1095
< x-pelican-token-generation: issuer=https://chtc.cs.wisc.edu, max-scope-
depth=3, strategy=OAuth2
< content-length: 109
```

Example director response

```
< HTTP/2 307
< content-type: text/html; charset=utf-8
< date: Mon, 08 Jul 2024 17:17:17 GMT
< link: <https://osdf-uw-cache.svc.osg-htc.org:8443/chtc/staging/bbockelm/testfile>; rel="duplicate"; pri=1; depth=3, <https://stash-cache.osg.chtc.io:8443/chtc/staging/bbockelm/testfile>; rel="duplicate"; pri=2; depth=3,...
< location: https://osdf-uw-cache.svc.osg-htc.org:8443/chtc/staging/bbockelm/testfile
< x-pelican-authorization: issuer=https://chtc.cs.wisc.edu
< x-pelican-namespace: namespace=/chtc, require-token=true, collections-url=https://origin-auth2000.chtc.wisc.edu:1095
< x-pelican-token-generation: issuer=https://chtc.cs.wisc.edu, max-scope-depth=3, strategy=OAuth2
< content-length: 109
```

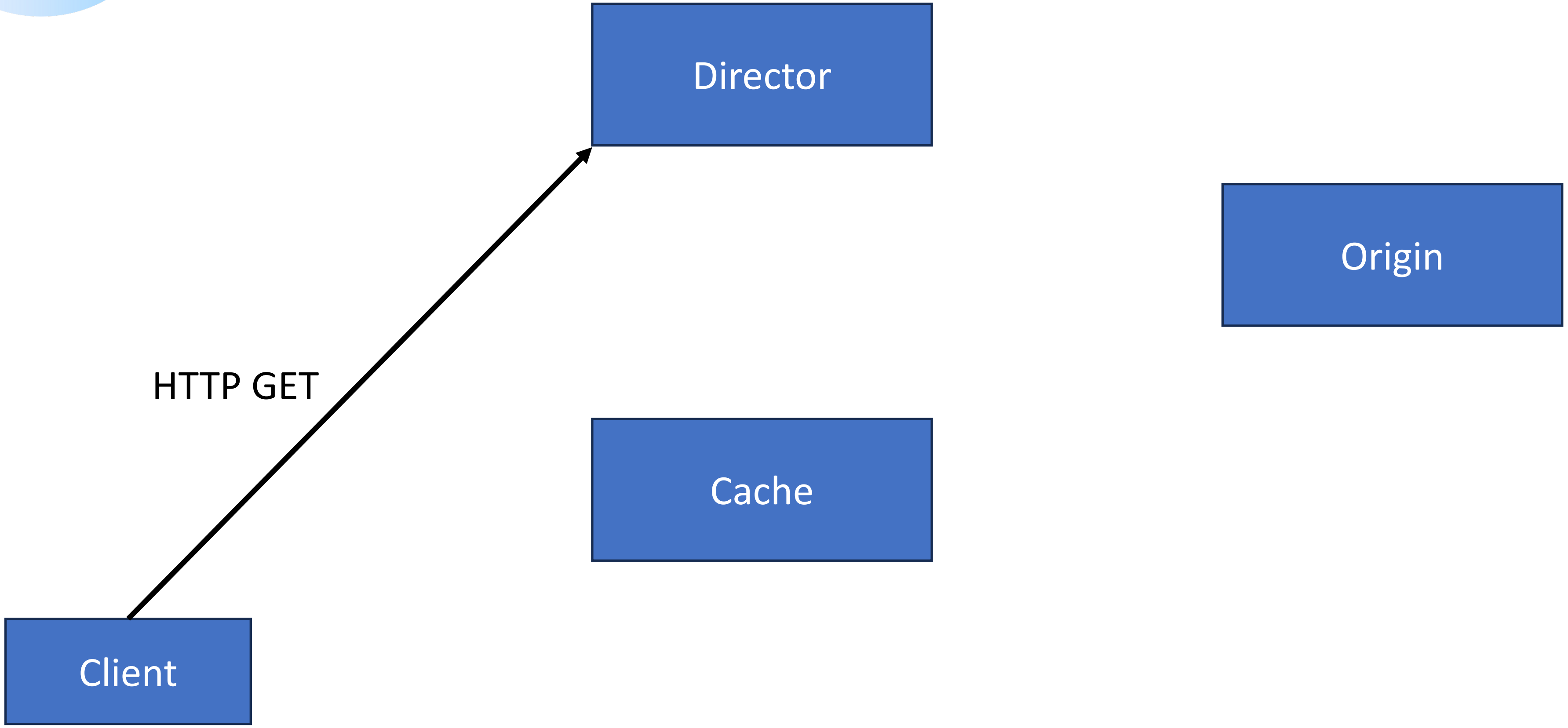
Example director response

```
< HTTP/2 307
< content-type: text/html; charset=utf-8
< date: Mon, 08 Jul 2024 17:17:17 GMT
< link: <https://osdf-uw-cache.svc.osg-
htc.org:8443/chtc/staging/bbockelm/testfile>; rel="duplicate"; pri=1;
depth=3, <https://stash-
cache.osg.chtc.io:8443/chtc/staging/bbockelm/testfile>; rel="duplicate";
pri=2; depth=3,...
< location: https://osdf-uw-cache.svc.osg-
htc.org:8443/chtc/staging/bbockelm/testfile
< x-pelican-authorization: issuer=https://chtc.cs.wisc.edu
< x-pelican-namespace: namespace=/chtc, require-token=true, collections-
url=https://origin-auth2000.chtc.wisc.edu:1095
< x-pelican-token-generation: issuer=https://chtc.cs.wisc.edu, max-scope-
depth=3, strategy=OAuth2
< content-length: 109
```

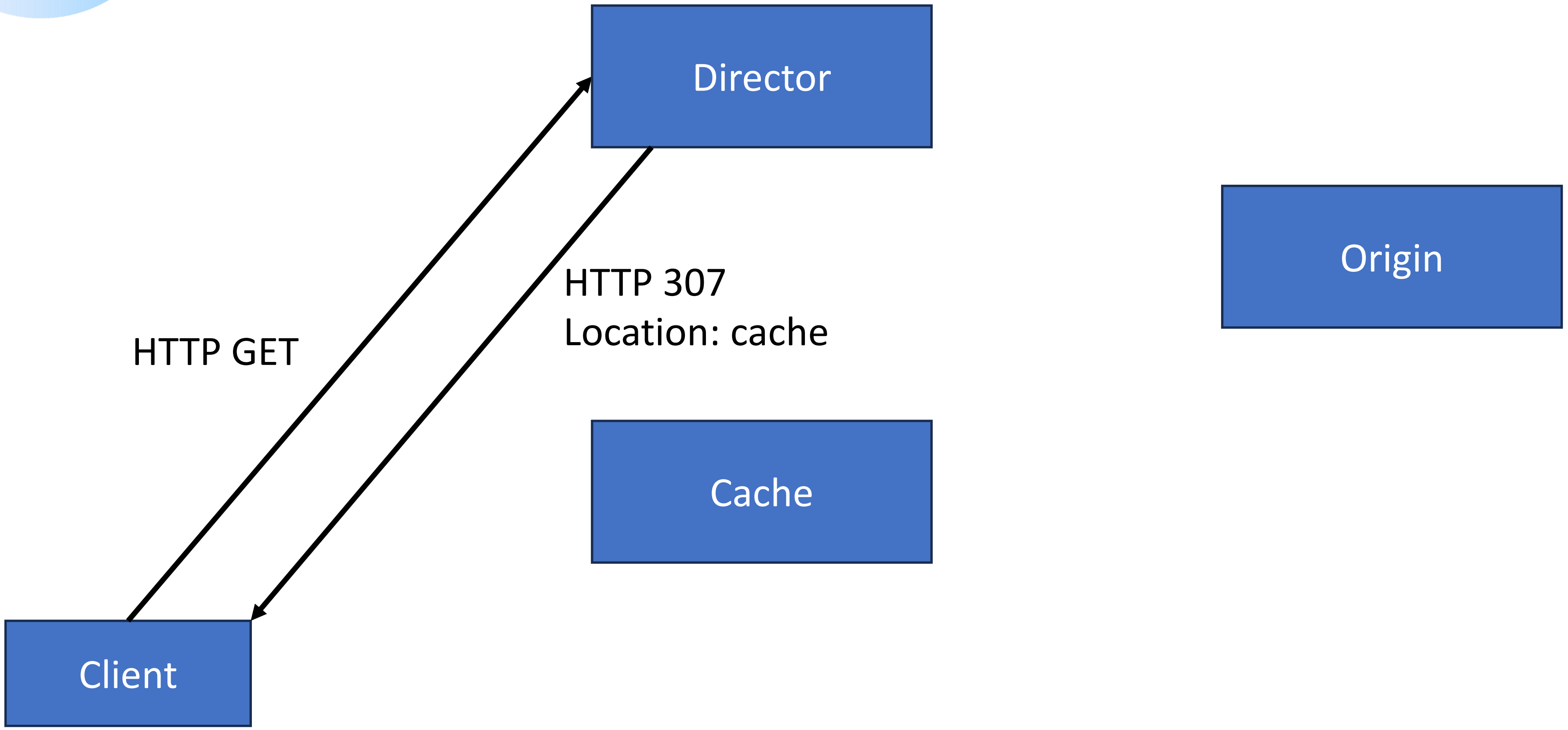
Director Response

- If you speak "plain HTTP", you only understand the "blue" headers and will successfully access the data.
- If you are the "Pelican client", you can interpret the "red" headers:
  - X-pelican-authorization: What token the client needs to successfully access the data.
  - X-pelican-namespace: What namespace the object is in. Informs client how to reuse the director response; no need to return to director for each object.
  - X-pelican-token-generation: If the client doesn't have a usable token, how to receive one.
  - Link: An ordered list of potential endpoints (caches) that can serve the requests. Actually, a standard RFC header (RFC 6249).

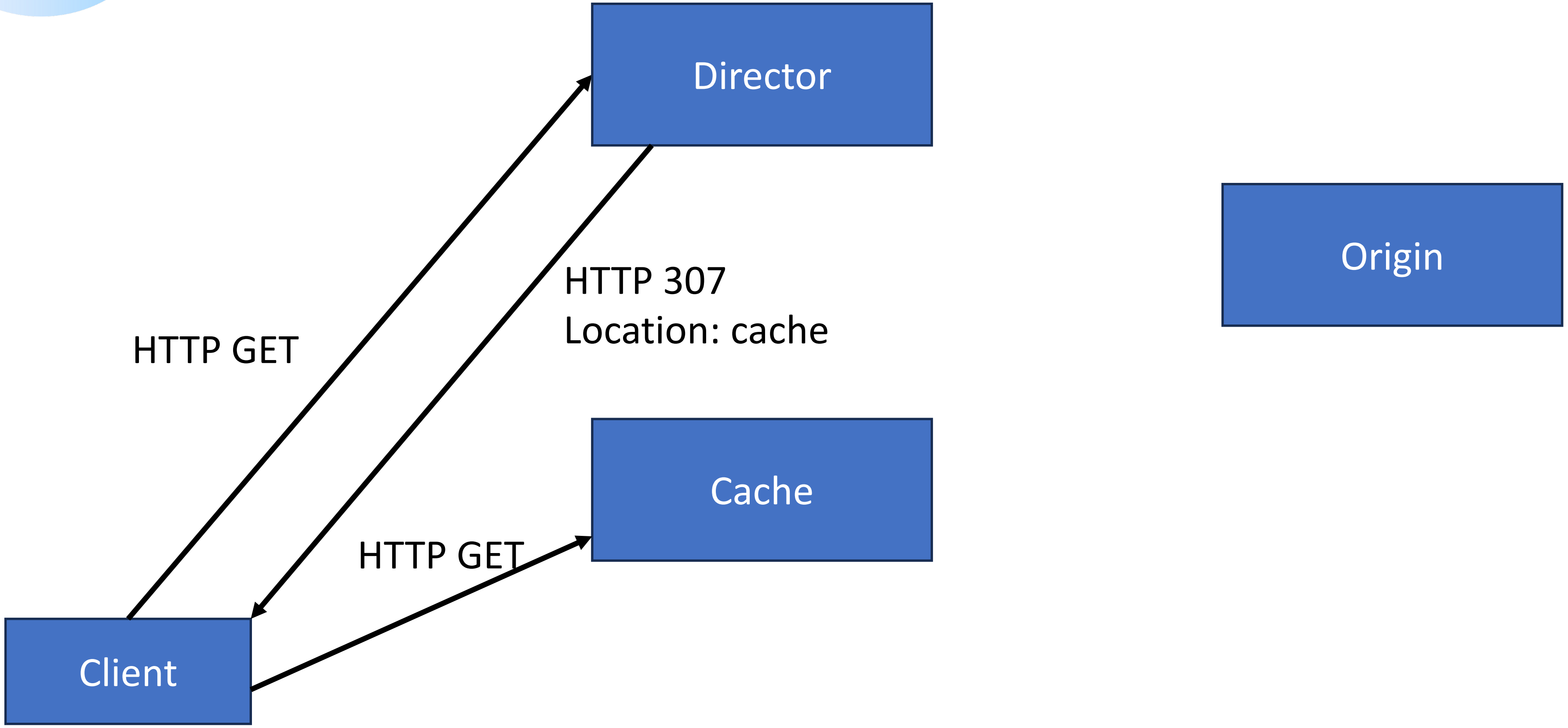
Cache access



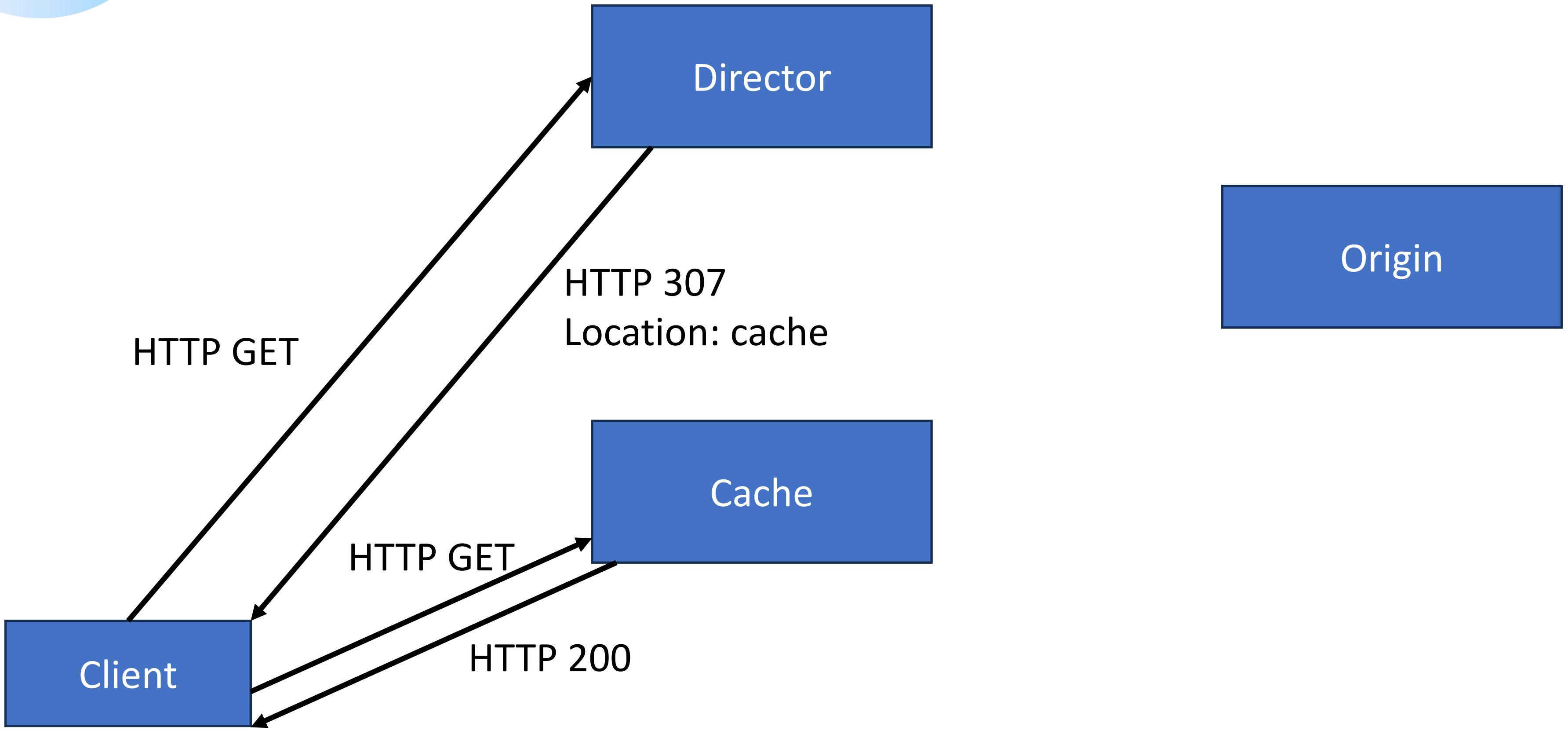
Cache access



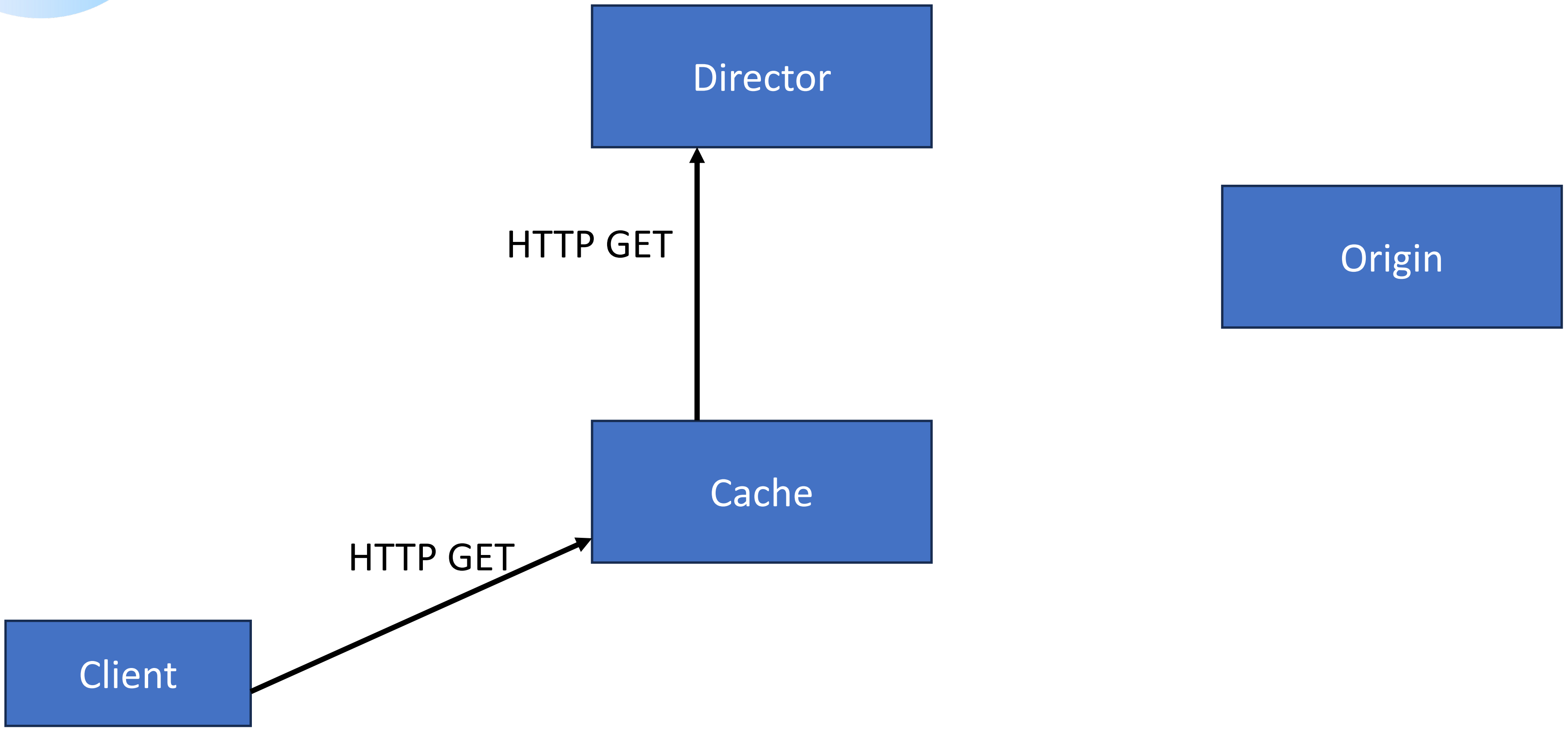
Cache access



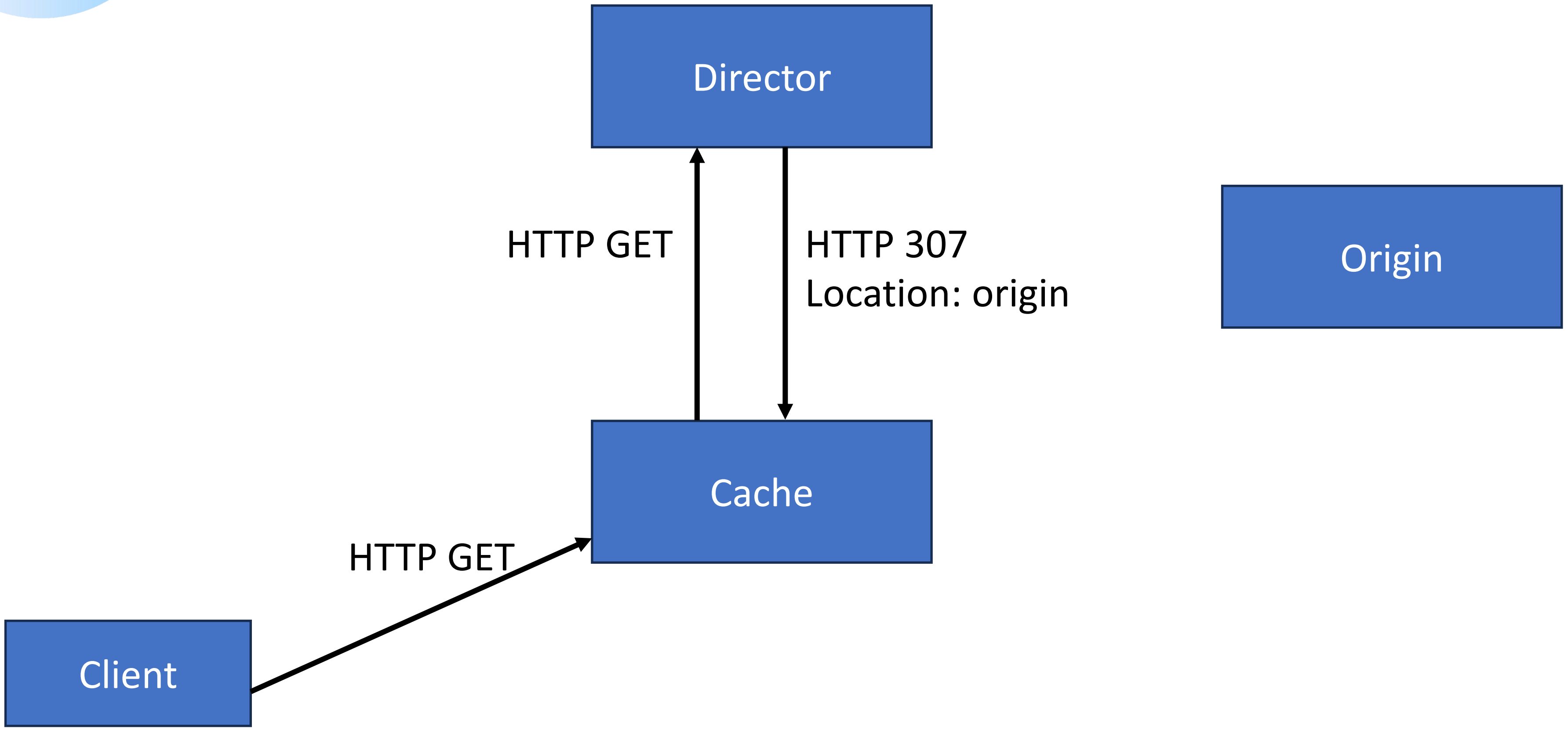
Cache access – hit!



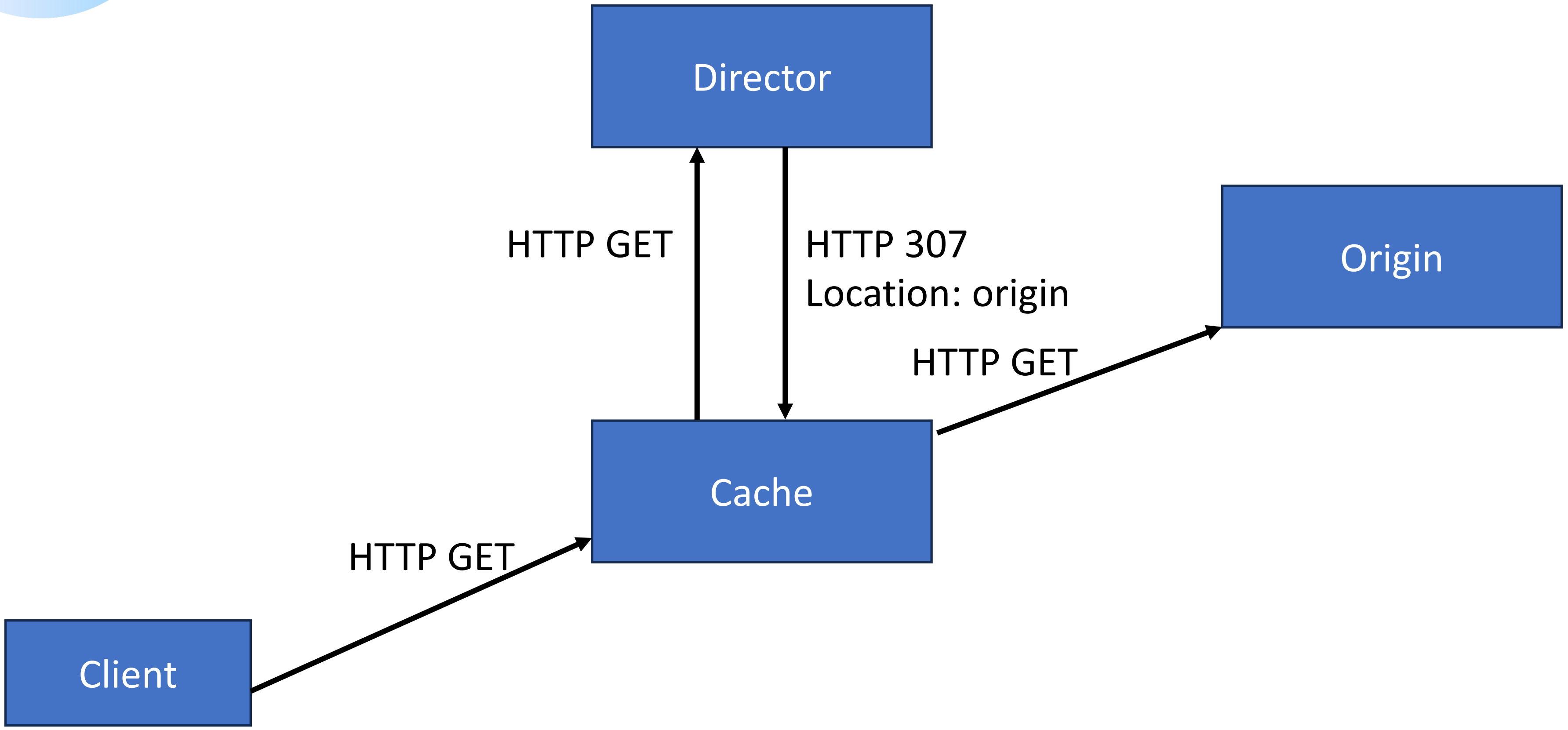
Cache access – miss



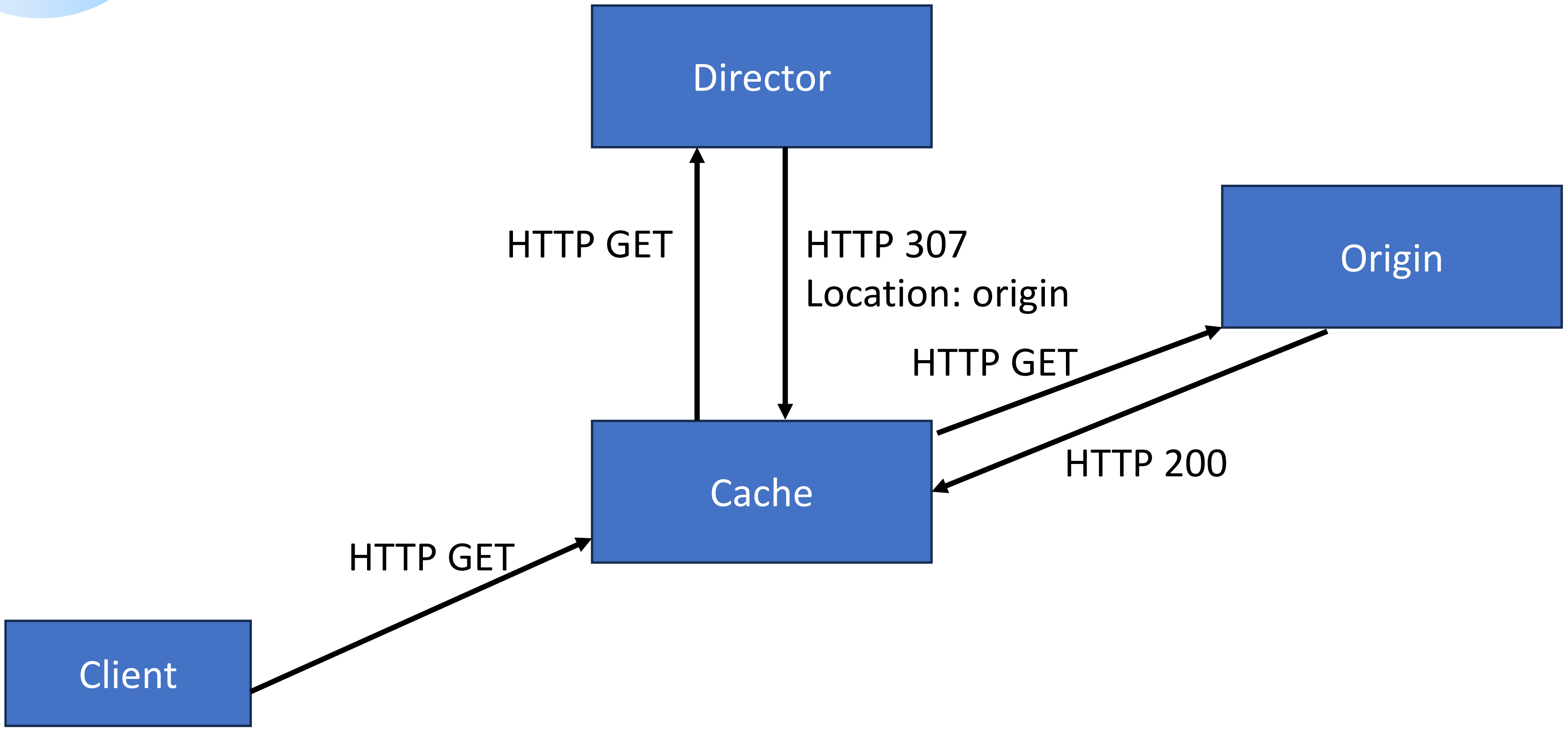
Cache access – miss



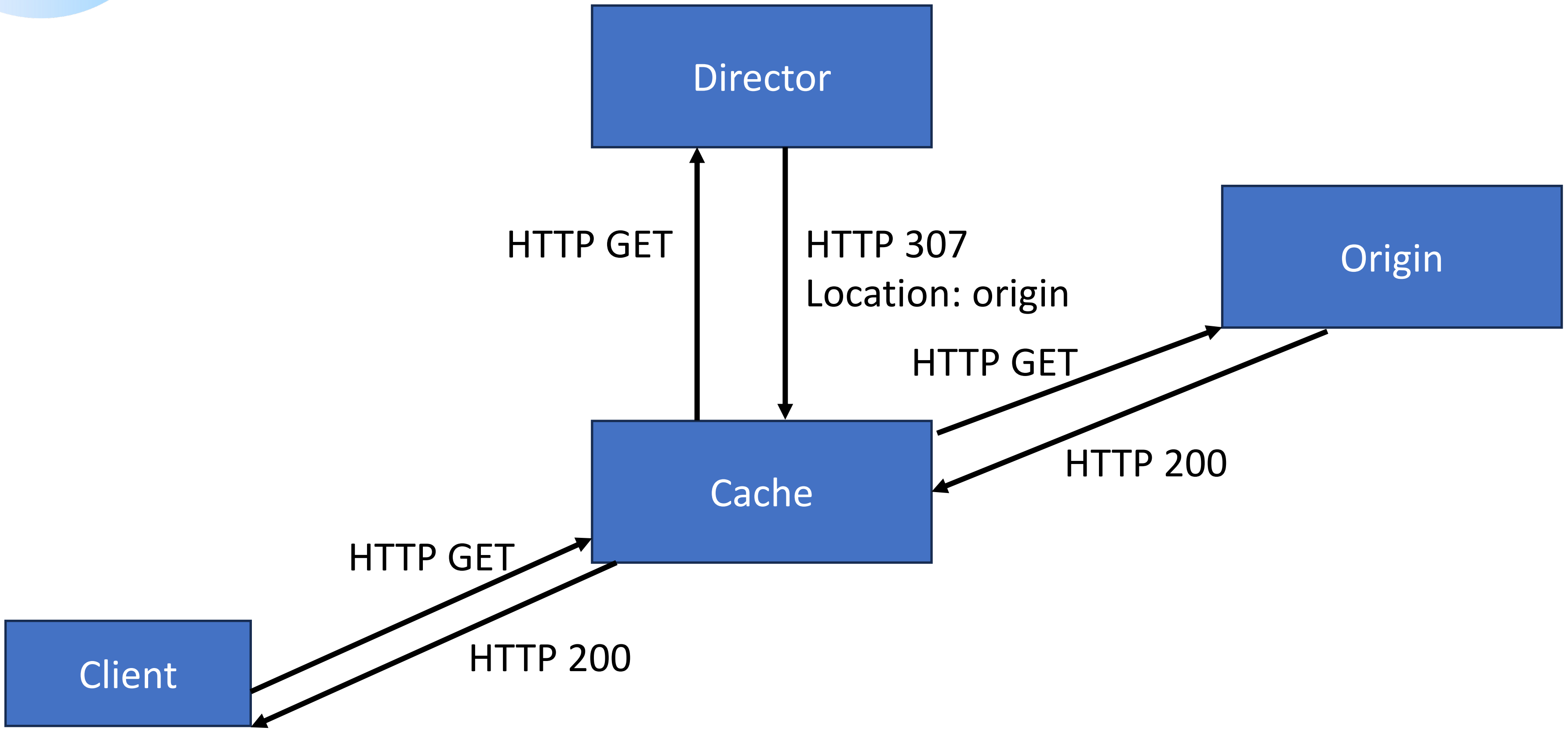
Cache access – miss



Cache access – miss



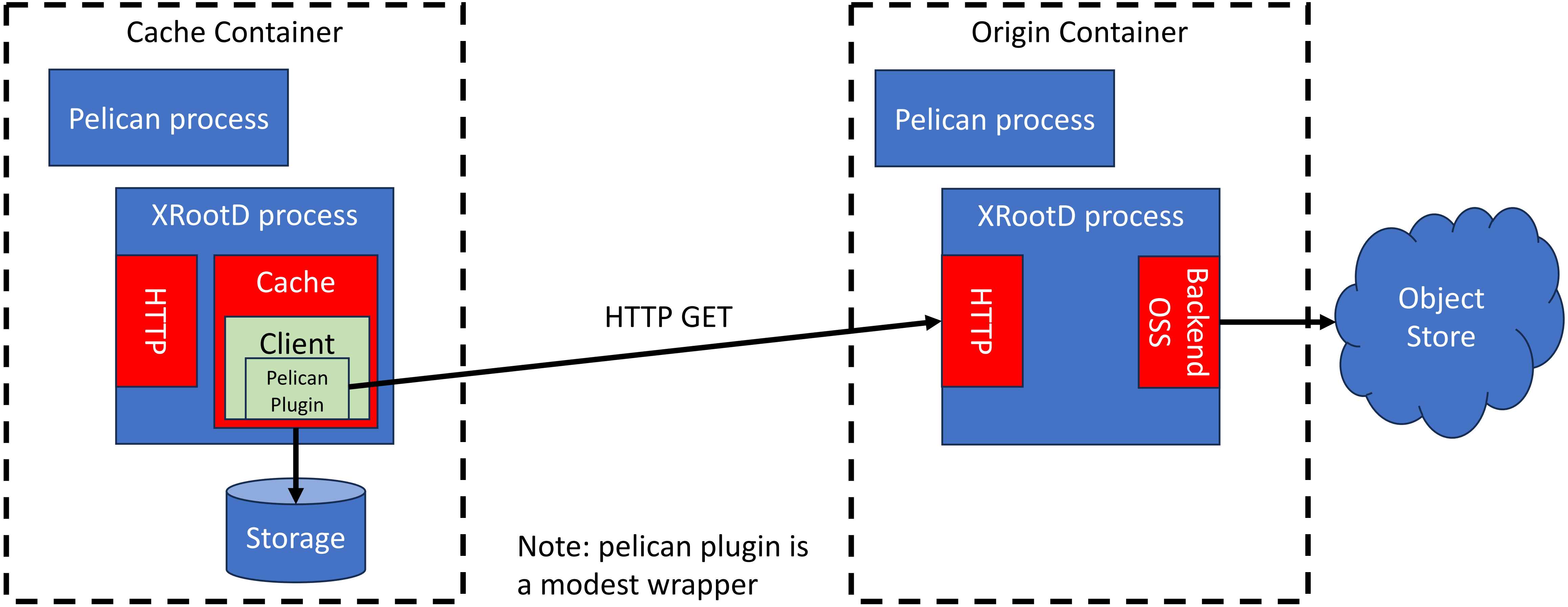
Cache access – miss



Some notes

- I drew the pictures as if the director blindly redirects the cache to an origin.
    - In reality, there may be multiple origins for the namespace. The director may perform a HEAD request to each potential origin and decide the "best" one for a request based on the response.
- What happens if the object is 1PB?
    - We don't want a client request to wait until 1PB is moved to the cache.
    - The cache requests smaller, 64KB chunks in parallel.
    - The response to the "client" starts as soon as the first chunk is returned.

A slide for the XRootD people out there…

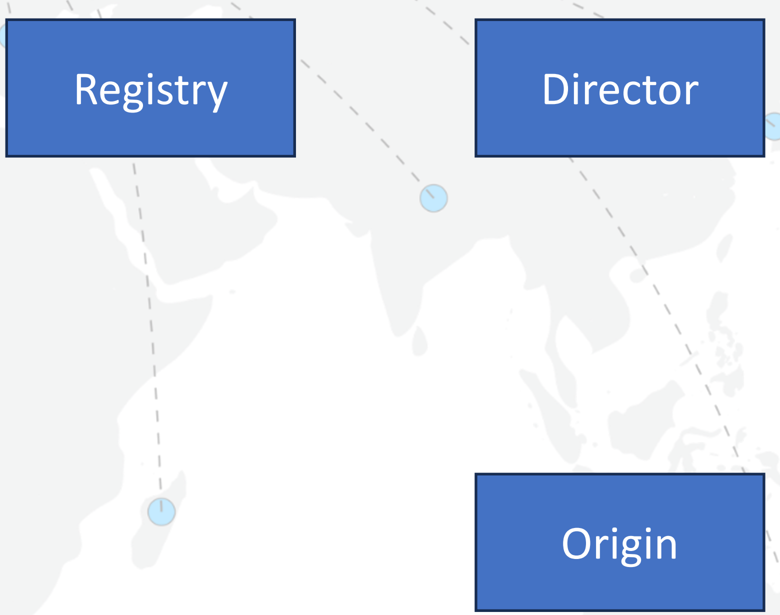


Note: pelican plugin is a modest wrapper around libcurl.

How do you trust an origin?

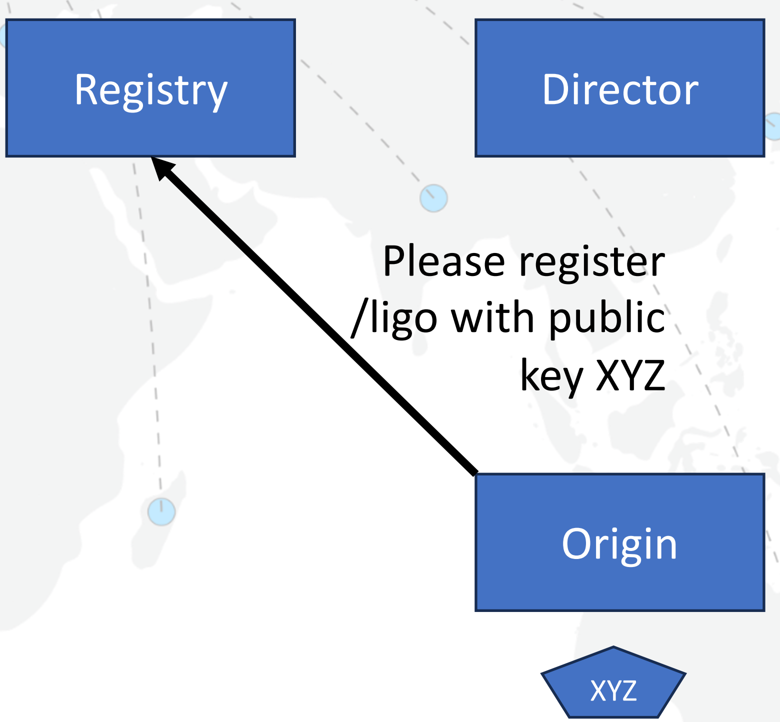
Don’t let just anyone connect to OSDF!

- If an origin connects to OSDF advertising it serves the /ligo namespace, how do we know that’s an OK origin to redirect users to?
  - I.e., how do you weed out “fake” origins?
- Answer: The Registry!



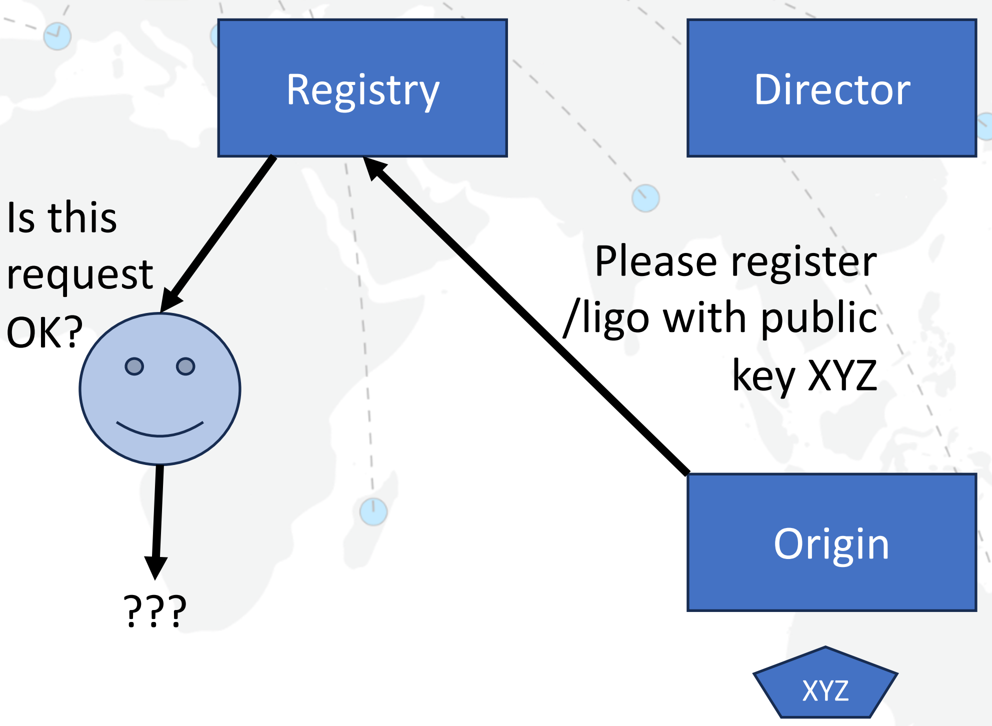
Don’t let just anyone connect to OSDF!

- If an origin connects to OSDF advertising it serves the /ligo namespace, how do we know that’s an OK origin to redirect users to?
    - I.e., how do you weed out “fake” origins?
- Answer: The Registry!



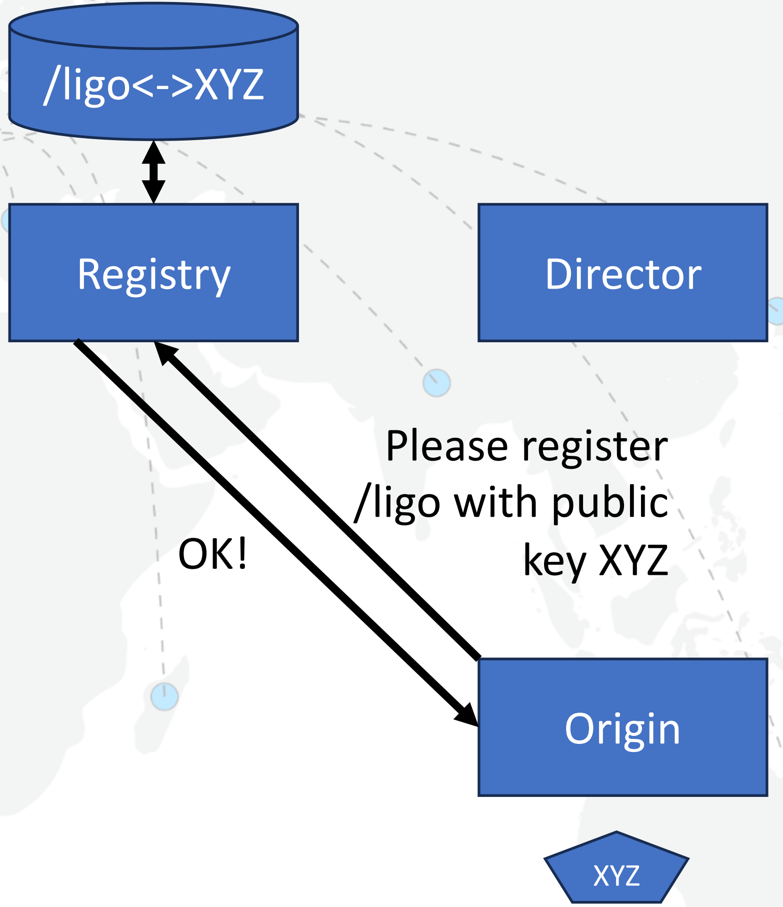
Don’t let just anyone connect to OSDF!

- If an origin connects to OSDF advertising it serves the /ligo namespace, how do we know that’s an OK origin to redirect users to?
  - I.e., how do you weed out “fake” origins?
- Answer: The Registry!



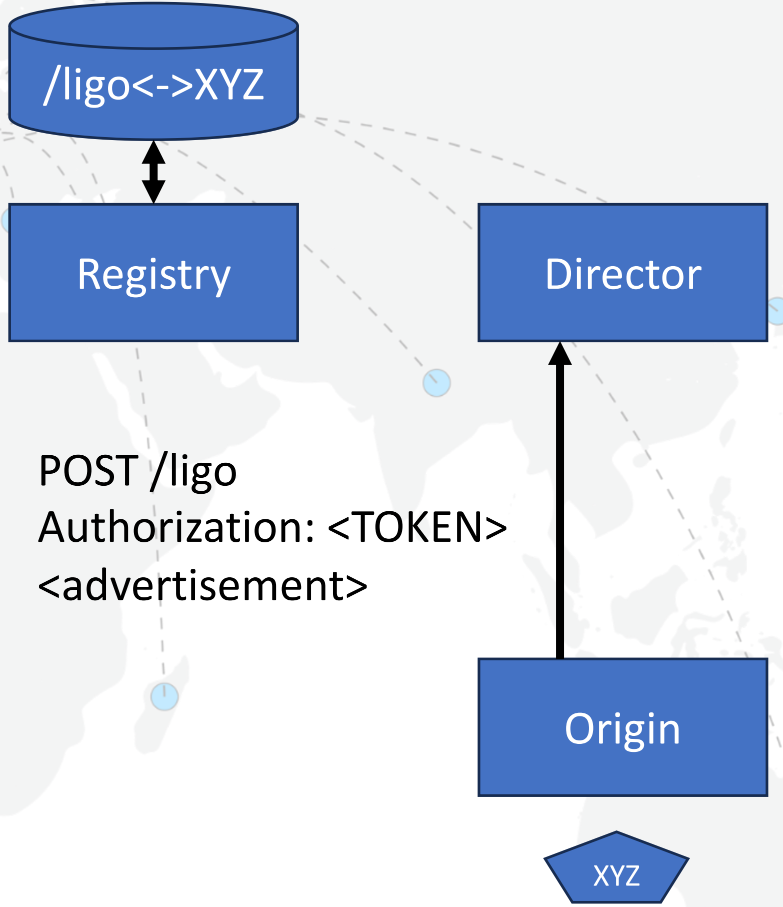
Don’t let just anyone connect to OSDF!

- If an origin connects to OSDF advertising it serves the /ligo namespace, how do we know that’s an OK origin to redirect users to?
  - I.e., how do you weed out “fake” origins?
- Answer: The Registry!



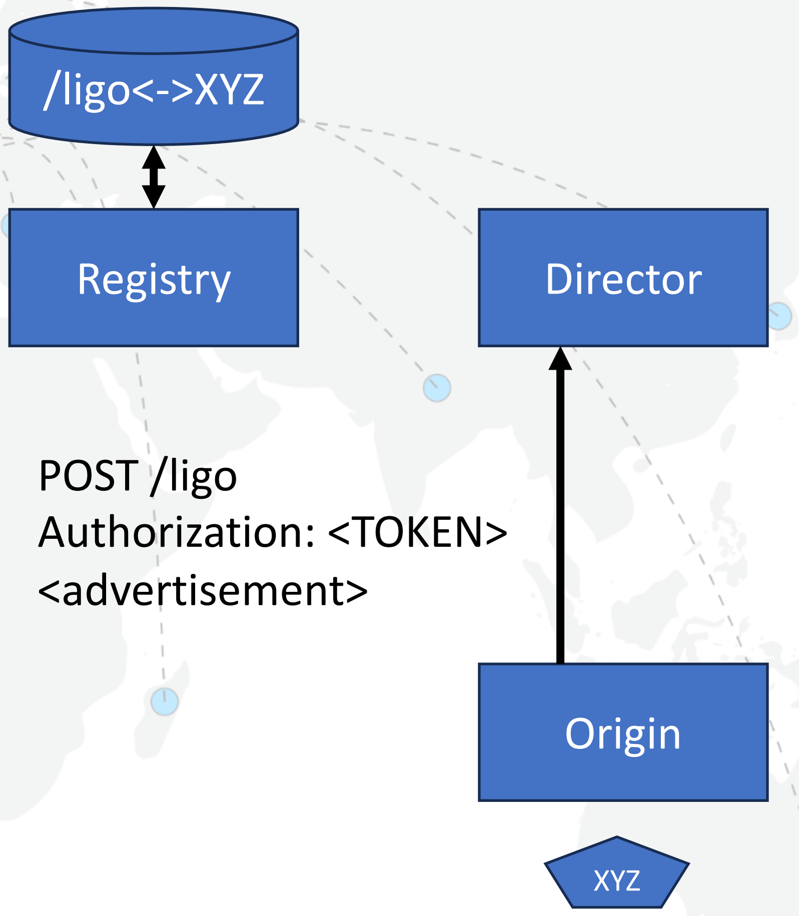
Don’t let just anyone connect to OSDF!

- The origin will advertise its services to the Director.
  - This advertisement contains information about how to contact the origin, what namespaces it supports, what token issuers it supports, the operations it is willing to perform (read/write).
- Sounds like a HTCondor collector, no?



Don’t let just anyone connect to OSDF!

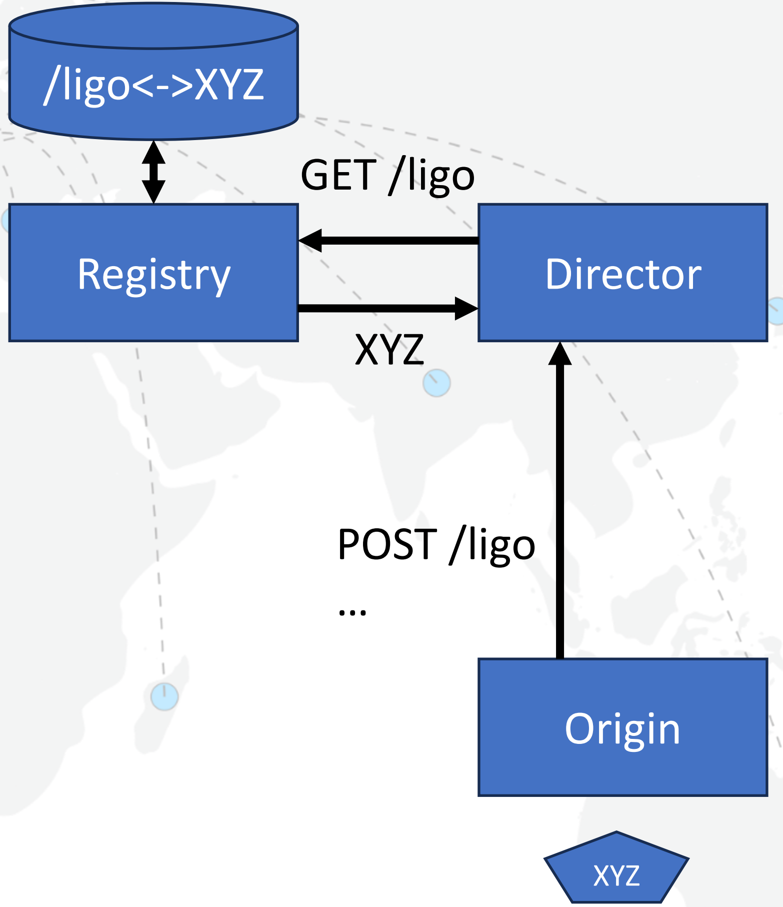
- What’s in the token?
  - Standard JWT headers
  - Capability for “advertise”
  - Issuer name
  - Public key name (“XYZ”)



POST /ligo
Authorization: <TOKEN>
<advertisement>

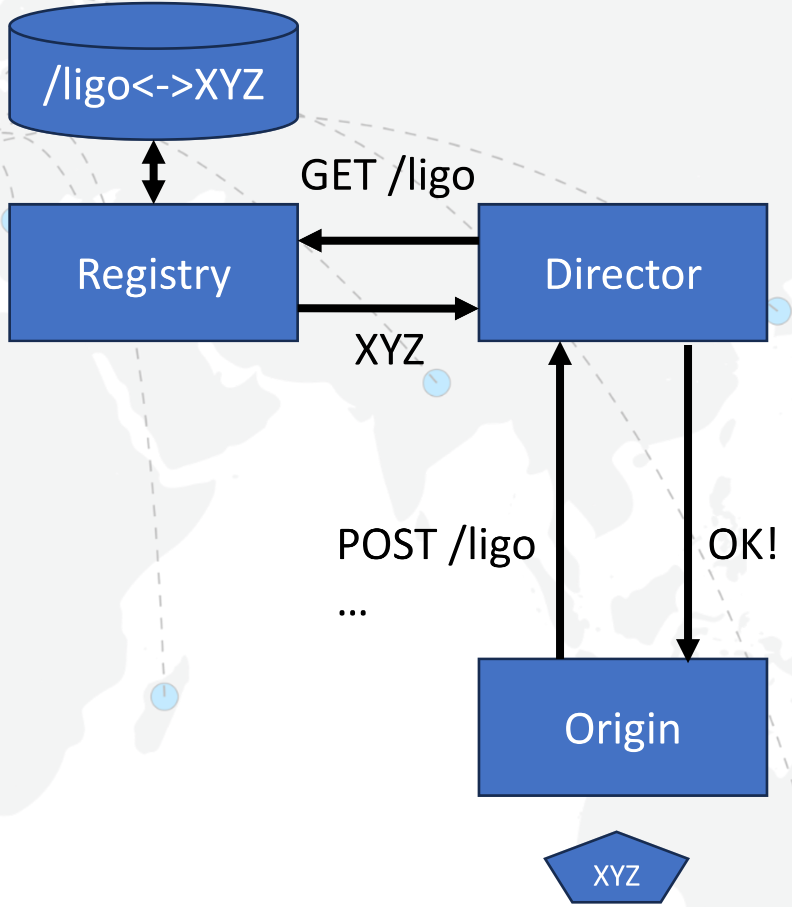
Don’t let just anyone connect to OSDF!

- What’s in the token?
  - Standard JWT headers
  - Capability for “advertise”
  - Issuer name
  - Public key name (“XYZ”)
- Director looks up the public keys allowed for the /ligo namespace.



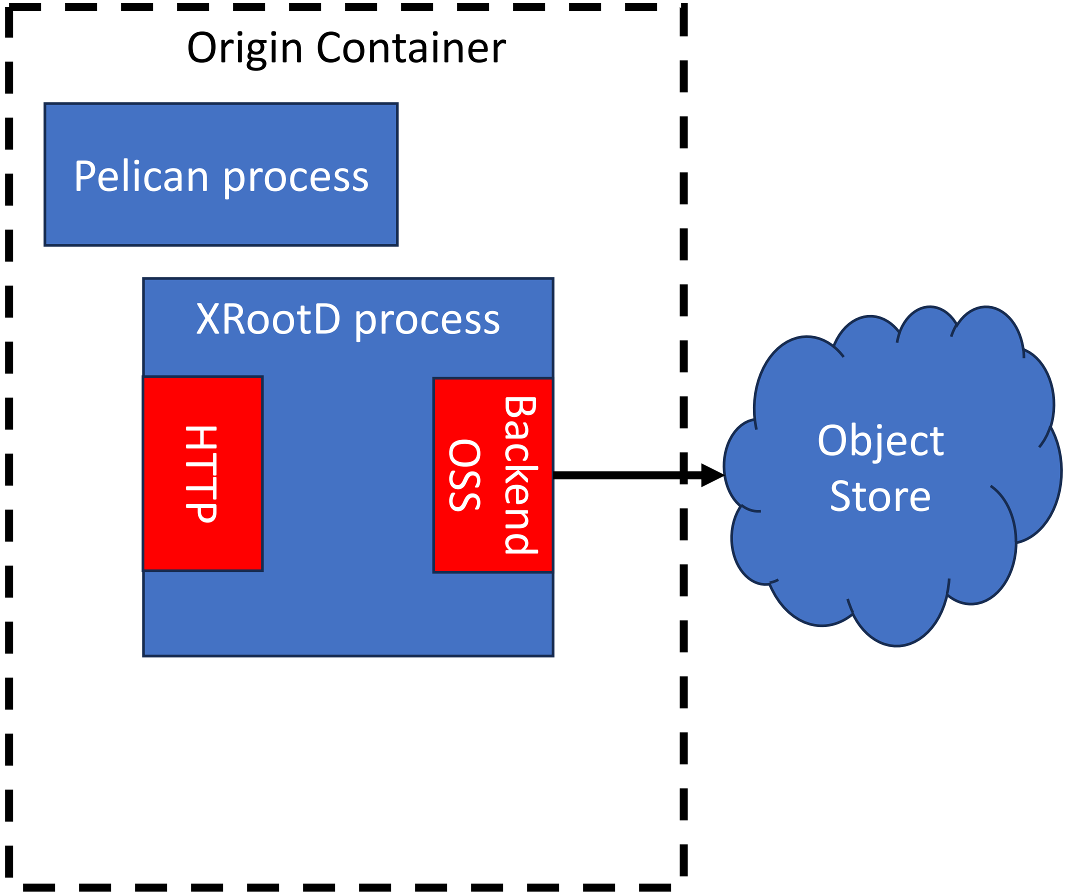
Don’t let just anyone connect to OSDF!

- What’s in the token?
  - Standard JWT headers
  - Capability for “advertise”
  - Issuer name
  - Public key name (“XYZ”)
- Director looks up the public keys allowed for the /ligo namespace.
  - Registry responds with the information in the DB.



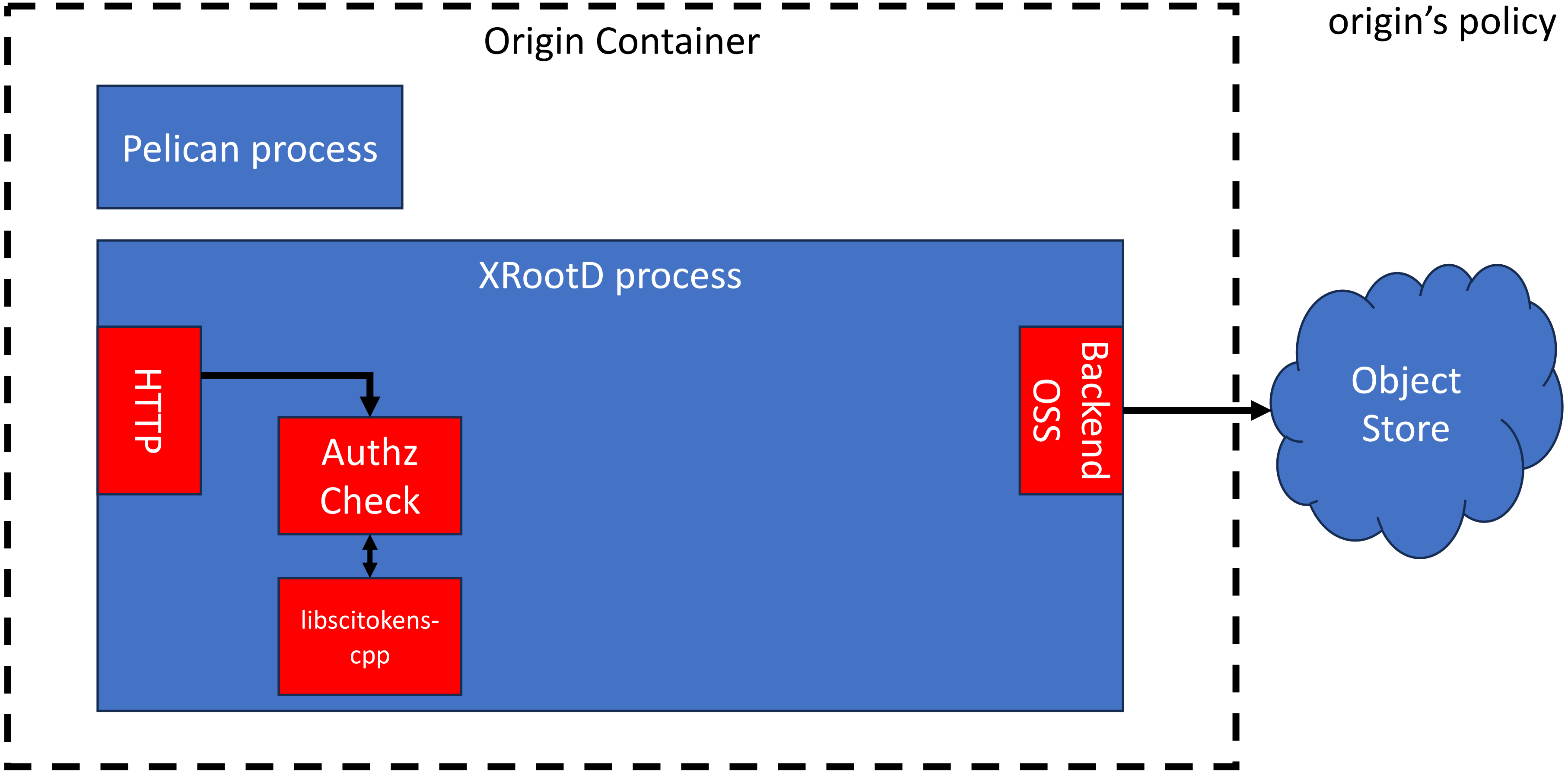
Origin to the Object Store

Authorization and proxying

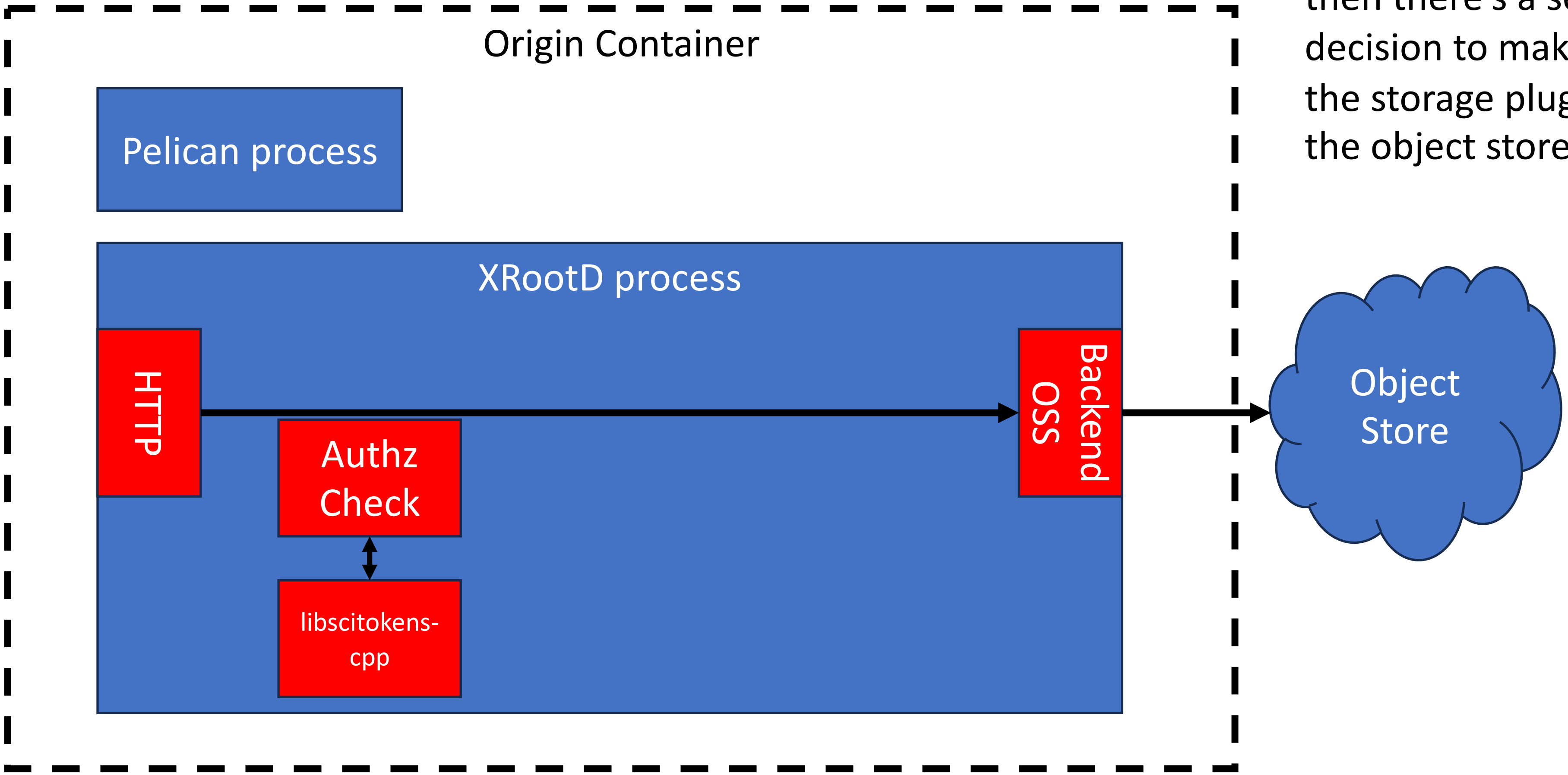


Authorization and proxying

All requests are explicitly authorized using the origin's policy configuration

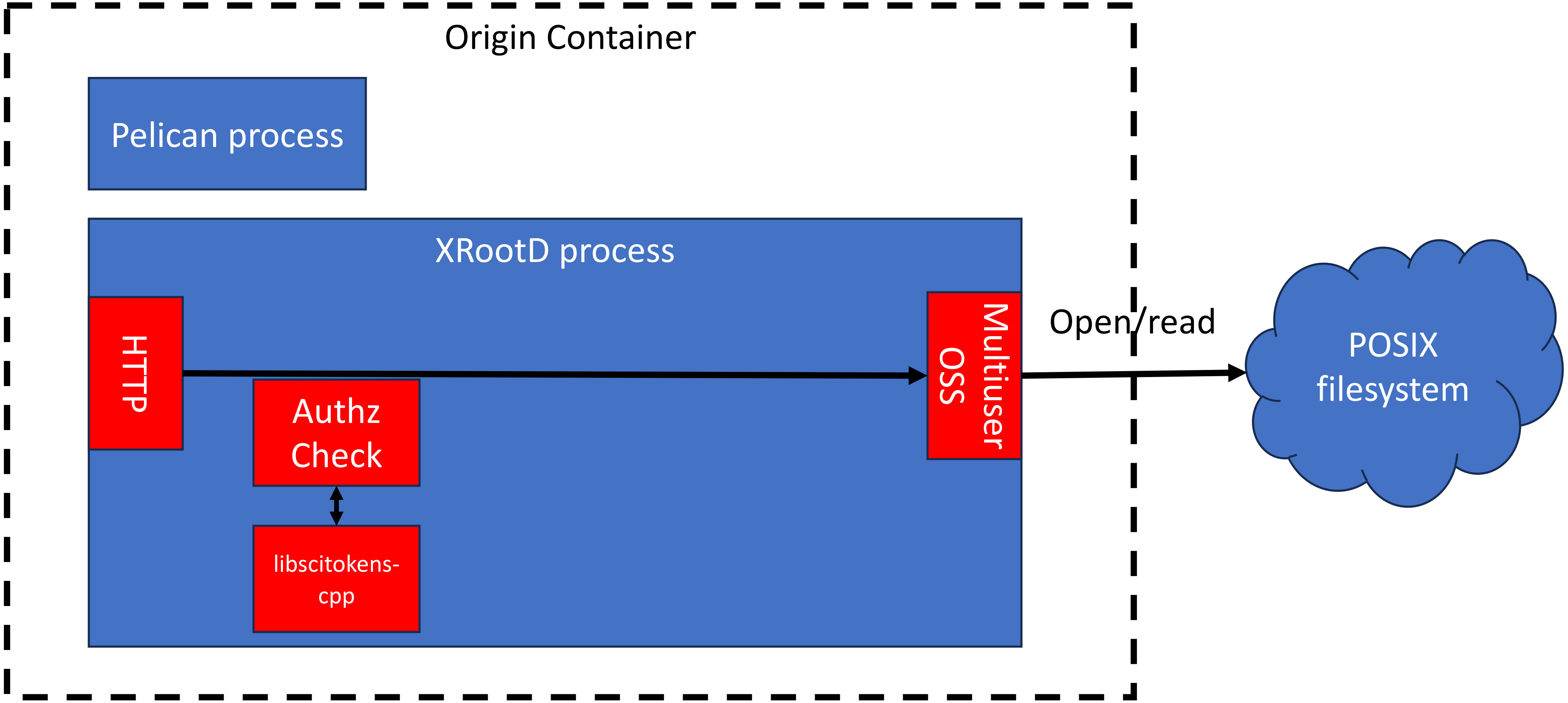


Authorization and proxying

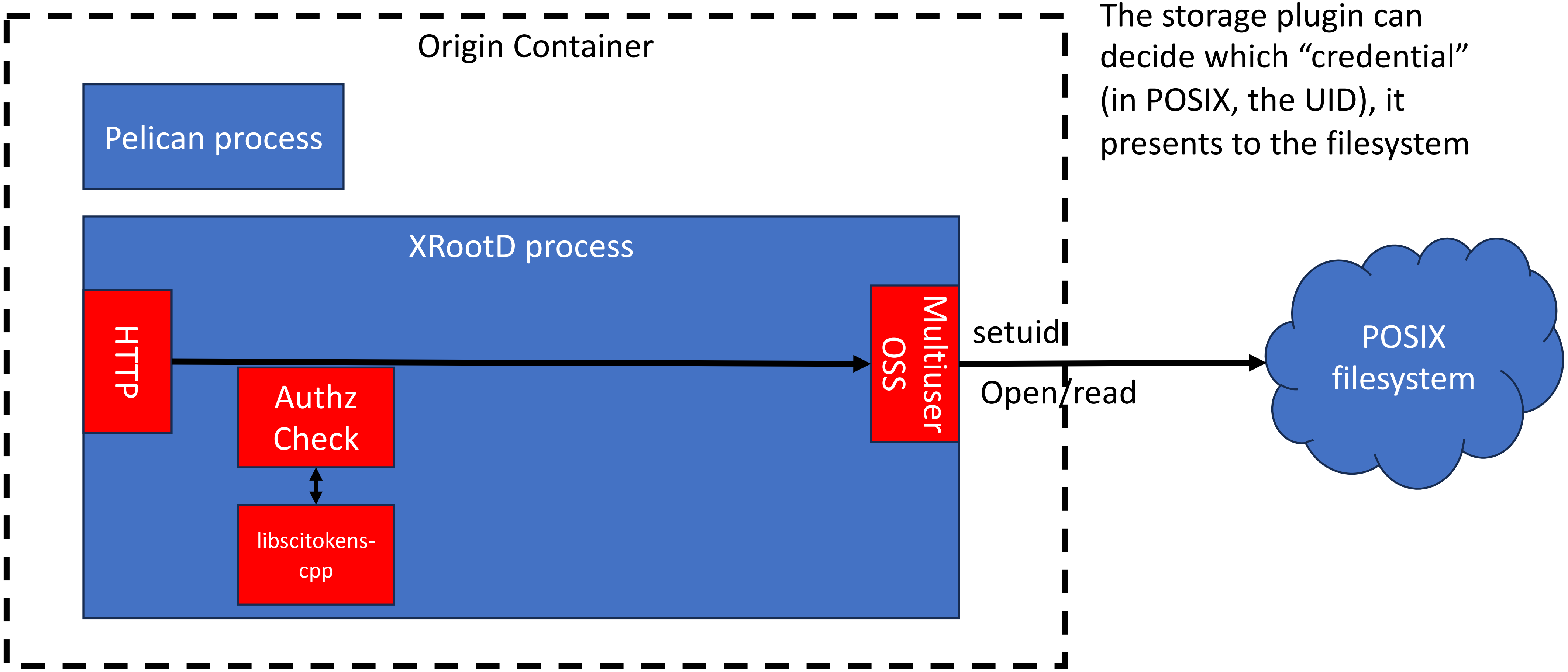


Once the request is authorized, then there’s a separate decision to make – how should the storage plugin interact with the object store?

Authorization: POSIX (simple)



Authorization: POSIX ("multiuser")

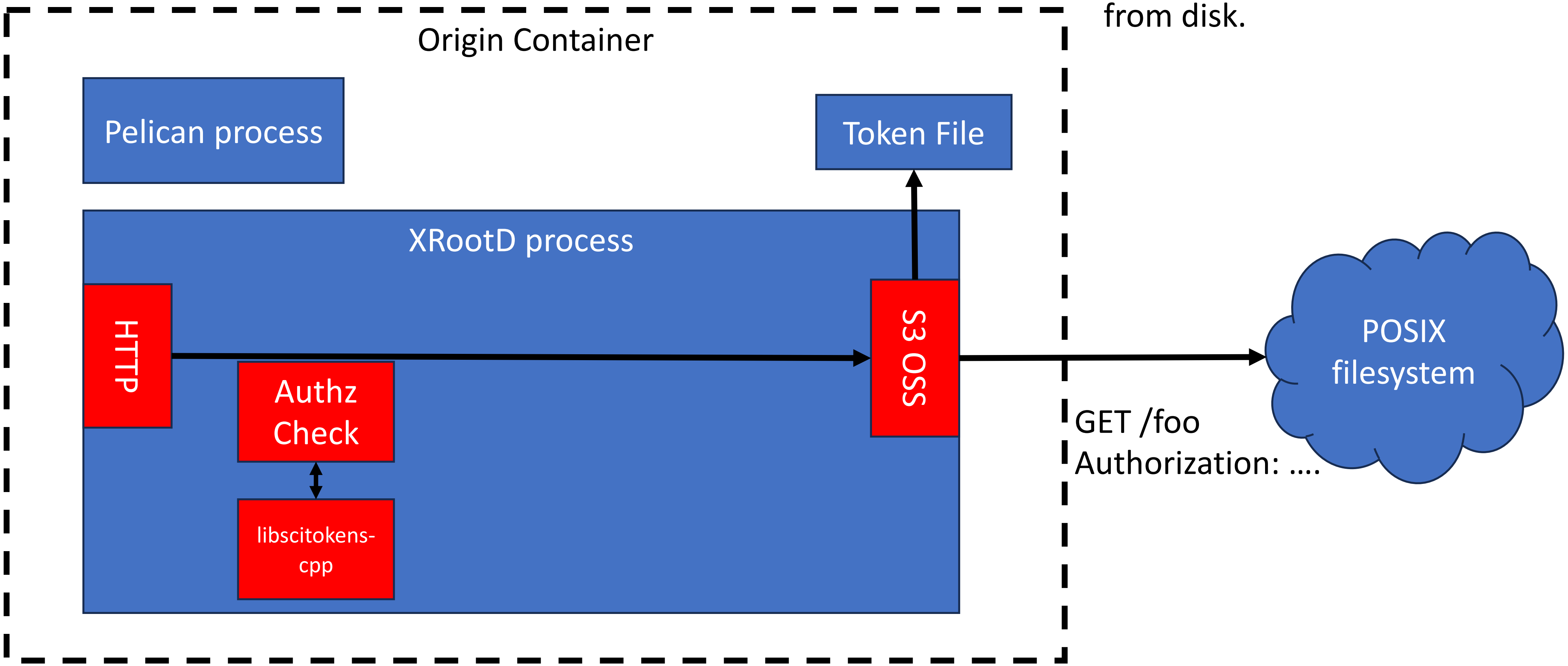


The storage plugin can decide which "credential" (in POSIX, the UID), it presents to the filesystem

Origin <-> object store

- The storage plugin translates the storage operation(s) into a sequence of commands the object store understands.
  - This might be conceptually simple. For POSIX, this is “open” followed by many “read” followed by “close” on a mounted filesystem.
  - For HTTP-esque object stores (including S3), it the translation may be a sequence of GETs or PUTs.
- The plugin assumes that once it is invoked, the request is authorized – and the remaining decision is “how do I interact with the object store”.
  - It may decide to use the same credentials for each request.
  - It may select a credential to use based on information derived from the token.
  - It may select a credential based on the bucket the object is read from.
  - It *never* runs its own authorization logic.

Authorization: S3



For S3, based on the bucket name, the plugin decides which S3 credential to read from disk.

What a whirlwind tour!

- As when you look “under the hood” of a car, it’ll take awhile to understand each component.
  - I hope this provides you a feel for some of our approaches.
  - The rest of the session looks at technical details from other angles.
- Pelican is <1 year old – this is the first time trying to explain the ecosystem to this crowd.

What else would you like to learn about?

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.