Table of Contents

HTTP / WebDAV Third-Party-Copy Technical Details....1
High-level overview....1
Transfer Cancellation and Failure....2
GridSite delegation....2
Server requests delegation....3
Request Headers....3
Response....4
Monitoring copy progress....5

HTTP / WebDAV Third-Party-Copy Technical Details

The HTTP TPC mechanism relies on utilizing the existing WebDAV COPY verb (see RFC 4918) and interoperable implementations / interpretations of this part of the specification.

Particularly, in many WebDAV implementations, `COPY` is limited to resources inside the same service; for third-party-copy, we allow this to trigger transfers from remote services.

High-level overview

We wanted to avoid technologies that only work with X.509 credentials, that are proprietary, and are not universally supported by WLCG storage. Although GridSite allows HTTP TPC transfers to succeed through X.509 delegation, it is not universally available and is limited to X.509. Moreover, as tokens are seen as a future direction, we wanted to focus on a token based flow. Note that, although most storage systems issue a macaroon as this token and the following description uses the term 'macaroon', the protocol does not care if the token is a macaroon.

Rucio delegates an X.509 credential to FTS.

FTS uses this X.509 proxy to request a macaroon from A and another (independent) macaroon from B. Let's call the macaroon that A issued M_A, and call the macaroon from B M_B.

FTS chooses the caveats based on the roles of A and B. If A is the source server and B is the destination server then FTS requests a read-only macaroon from A and a macaroon that allows uploads from B.

Once FTS has the two macaroons (M_A and M_B) it never uses the delegated X.509 credential [not completely true, but doesn't matter].

Macaroons work like bearer tokens, but the way we're using them, they are only ever interpreted/parsed by the server that issued them.

For example, if A is the source server then FTS will check whether the file exists (using an HTTP `PROPFIND` request). To do this, it uses M_A (`M_A`) to authorise the request to A.

```
PROPFIND /path/to/file HTTP/1.1
Host: A
Authorization: Bearer M_A
```

In a similar way, when checking that the destination directory exists, FTS will use M_B (`M_B`) to authorise the corresponding HTTP `PROPFIND` request to B.

```
PROPFIND /destination/path/ HTTP/1.1
Host: B
Authorization: Bearer M_B
```

When issuing the third-party copy request, FTS will use one macaroon to authorise the `COPY` and pass the other macaroon as part of this `COPY` request. The works because any headers in the `COPY` request that start `TransferHeader` are copied into the data-bearing HTTP request without this prefix; so the `TransferHeaderAuthorization` header in the `COPY` request is copied as the `Authorization` in the data-bearing HTTP request.

For example, if FTS is telling B to copy the file from A, it issues the `COPY` request to B and uses M_B to

authorise this request. It also gives M_A to B, so B can issue a `GET` request, using M_A to authorise that `GET` request.

Copying `https://A/source/path/my-data.root` to `https://B/path/within/destination/my-data.root` looks like:

The HTTP request from the client to B, requesting the TPC:

```
COPY /path/within/destination/my-data.root HTTP/1.1
Host: B
Source: https://A/source/path/my-data.root
Authorization: Bearer M_B
TransferHeaderAuthorization: Bearer M_A
```

This triggers B to make an HTTP `GET` request to A, to fetch the data:

```
GET /source/path/my-data.root HTTP/1.1
Host: A
Authorization: Bearer M_A
```

This shows an example of an HTTP pull request: B *pulls* the data from A.

A similar interaction happens for HTTP push request (A *pushes* data into B). The difference is the client issues the `COPY` request to A with a `Destination` request header, authorised with M_A and including M_B in that request. This triggers A to make an HTTP `PUT` request to B, authorized with M_B.

In all these cases, the macaroon is only understood/processed by the server that issued it (M_B by B, M_A by A).

Because of this, there's no cross compatibility issues here.

To illustrate that this really isn't a problem: StoRM has implemented support for HTTP TPC without using macaroons; instead, it is providing a JWT token. This is working, as demonstrated within the DOMA-TPC testbed.

The fact that StoRM isn't using a macaroon doesn't matter because only StoRM needs to parse the token.

Transfer Cancellation and Failure

To cancel a COPY transfer, the client should close the TLS and TCP connection. If the server encounters a TCP disconnect, it should interpret it as a cancellation of the request, stop the transfer, and release any resources associated with the transfer.

If a file has been partially copied when a cancellation or failure occurs, it is implementation-defined whether the partial file at the destination is kept or deleted. The behavior may also depend on the file transfer mode: for a push from the source to the destination, when the HTTP PUT is terminated, the destination may determine whether the partial file is deleted.

GridSite delegation

If one endpoint supports third-party-copy requests and the other endpoint supports authorisation bearer tokens (e.g., macaroons) then it is possible to achieve the transfer using a bearer token. If this is not the case, one of the endpoints must support third-party-copy with GridSite delegation to achieve the file transfer.

The following lists expectations if a third-party-transfer with gridsite delegation is desired:

- The client/user SHOULD have access to a credential that is valid for at least 20 minutes when making a GridSite COPY request.
- The server SHOULD request the client delegates a fresh credential when the client makes a GridSite COPY request and either has no delegated credential or the delegated credential has expired.
- The server MAY request the client delegates a fresh credential when the client makes a GridSite COPY request and the delegated credential has less than 20 minutes validity.
- The server SHOULD NOT request the client delegates a fresh credential when the client makes a GridSite COPY request and the delegated credential has more than 20 minutes validity.
- The server SHOULD reject the GridSite COPY request if the delegated credential has expired and the client failed to delegate a non-expired credential when requested.
- The server MAY reject the GridSite COPY request if the delegated credential has less than 20 minutes validity and the client failed to delegate a credential with greater than 20 minutes validity when requested.

Server requests delegation

A server requests the client delegates a credential by responding to the COPY request with a 302 response that includes the **X-Delegate-To** response header and a **Location** response header.

The **X-Delegate-To** header's value is a space-separated list of absolute URLs. Each URL is a GridSite delegation endpoint.

The client SHOULD delegate a credential to one of the listed GridSite delegation endpoints. If a delegation attempt fails then the client SHOULD attempt to delegate with another of the supplied GridSite URLs. The client MAY contact the GridSite endpoints in any order, but SHOULD try all endpoints before failing.

Once the client has successfully delegated a credential then the client SHOULD issue the same COPY request to the URL provided in the **Location** response header.

Request Headers

We utilize several HTTP headers in the WebDAV Third-Party Copy (TPC) request that modify how the copy works. These are listed below:

RequireChecksumVerification

Controls whether a successful transfer requires the service to obtain a remote checksum value that using the same checksum algorithm of a known checksum value. This could be because the 3rd-party server does not support reporting checksum values or the local server does not know how to generate the supplied checksum. Valid values are `true` (fail transfer if the remote server does not provide a matching checksum value) and `false` (allow transfer to succeed in the absence of checksum information). Regardless of the value of `RequireChecksumVerification`, if the server determines the checksum is incorrect, it SHOULD fail the transfer. If not specified, then the behavior is implementation-specific. Clients SHOULD specify this header and not rely on implementation-specific behavior. We note two distinct behaviors:

- ◊ **DPM**: DPM ignores this header and never performs checksum validation.
- ◊ **dCache**: If not specified then (by default) `true` is used (the admin may change this default). dCache will always use RFC 3230 to request a checksum from the remote host and fail the transfer if there is a checksum mismatch. *However*, if the remote host does not return a checksum (or does not support the checksum type), this option controls whether or not dCache will fail the transfer.

Credential

Controls from which source the TPC request credentials will be obtained. Clients performing a COPY request SHOULD always set the **Credential** header.

- ◊ Currently defined values are `gridsite` (obtain via gridsite delegation), `oidc` (obtain credential via OAuth 2.0 Token Exchange), or `none`.
- ◊ If the client does not specify the **Credential** header, then the handling of credential delegation is implementation-specific.
- ◊ If token authentication (or any other HTTP header-based authentication) with the inactive endpoint is used, then the client may set **Credential** to `none` and specify headers for the transfer via the `TransferHeader*` mechanism.
- ◊ If the client specifies the **Credential** header, the server MUST utilize the corresponding mechanism *or* reject the request. If the server does not support the mechanism, it MUST response with 400 status code.

Overwrite
: Controls whether the client desires the TPC request to overwrite an existing file. Valid values are `T` (overwrite any existing file) and `F` (fail request if file already exists). If not specified then `T` is used.

Source
: The URL from which data will be read. This makes the COPY request a pull request. The value must be a valid URL. Must not be specified in a request that defines the `Destination` header.

Destination
: The URL to which data will be written. This makes the COPY request a push request. Must not be specified in the same request that defines the `Source` header.

TransferHeader*
: any header that starts `TransferHeader` is copied into the GET or PUT request but without this prefix; for example a header `TransferHeaderAuthorization: Bearer foo` is coped into TPC requests as `Authorization: Bearer foo`.

X-Number-Of-Streams
: The client-specified number of TCP streams the server should utilize to perform the transfer. The server SHOULD interpret this as a hint and may ignore it if the number is large (or multiple streams is not supported by the implementation.

X-No-Delegate
: DPM-specific. If set to `true`, DPM will not require gridsite delegation. Implementers are not suggested to use this but rather set **Credential: none** header outlined above.

Response

In addition to the general status codes possible, the following status codes have specific applicability to COPY:

201 (Created)
: The source resource was successfully copied. The COPY operation resulted in the creation of a new resource.

202 (Accepted)
: Copy request accepted. Servers may provide progress information about the COPY request as part of chunked encoding response (RFC 7230).

204 (No Content)
: The source resource was successfully copied to a preexisting destination resource.

207 (Multi-Status)

Support for the WebDav (RFC-2518 10.6)

30X (Redirections)
All redirections code MUST be supported by the client.

403 (Forbidden)
The operation is forbidden. A special case for COPY could be that the source and destination resources are the same resource.

409 (Conflict)
A resource cannot be created at the destination until one or more intermediate parent resources have been created.

412 (Precondition Failed)
A precondition header check failed, e.g., the Overwrite header is "F" and the destination URL is already mapped to a resource.

Note that the **202 (Accepted)** response code indicates that the transfer request was accepted, but does not necessarily indicate whether the transfer request will succeed, nor is it an indication that the request has even started. There are a number of early checks that can cause the transfer request to fail before any bytes are moved (e.g., invalid URL provided in the `Source:` header). It is implementation-defined whether any of these are performed prior to accepting the transfer, or if "acceptance" is just into an internal queue.

The **202 (Accepted)** response (and corresponding copy progress discussed below) is the only mechanism provided to indicate progress of a transfer. Accordingly, client applications (such as FTS) may time out long-running transfers if this mechanism is not used.

Monitoring copy progress

The **202 (Accepted)** response must be followed by a sequence of chunks as part of a chunked transfer encoding response. Each chunk is a referred to as a "performance marker" and updates the client about the progress of the transfer. Each performance marker is meant to be processed independently by the client (i.e., not encoded into multiple chunks) and sent periodically by the server. The Xrootd implementation, for example, sends a performance marker every 5 seconds.

The format of the marker is as follows (newline characters are shown unencoded but are encoded in the response):

```
Perf Marker\n
Timestamp: $(UNIX TIMESTAMP)\n
Stripe Index: 0\n
Stripe Bytes Transferred: $(BYTES)\n
Total Stripe Count: 1\n
RemoteConnections: $(CONNECTIONS)\n
End\n
```

If no bytes have been transferred yet, then `Stripe Index`, `Stripe Bytes Transferred`, and `Total Stripe Count` may be omitted.

dCache provides Perf Marker description in the WebDAV documentation and they use additional keywords: `State description`, `Stripe Start Time`, `Stripe Last Transferred`, `Stripe Transfer Time`, `Stripe Bytes Transferred` and `Stripe Status`. XRootD on the other side can finish transfer with `aborted: msg` defined in the XrdTpc documentation.

The `RemoteConnections` line is optional. If present, it should list the existing network connections currently

associated with this transfer. This should be a comma-separated list formatted as `::`, where should be set to `tcp` for a TCP-based connection. An example may be `tcp:129.93.3.4:1234`. Notes on this field:

- This should indicate the connections that currently exist when the performance marker was created. It should not be the list of all historical connections for this transfer.
- Connections should be provided even if the transfer has not started (but the connection exists - an example may be a HEAD request prior to the transfer) or if the transfer is done via a non-HTTPS-based protocol.
- Another potential transport may be `udt`.
- IPv4 address should be given as the normal dot-decimal representation. Clients should accept IPv4-mapped (e.g., `::ffff:192.0.2.128`) addresses and interpret them as an IPv4-based connection.
- Ordering of the connections is not considered significant.

Example:

```
Perf Marker\n
Timestamp: 1537788010\n
Stripe Index: 0\n
Stripe Bytes Transferred: 238745\n
Total Stripe Count: 1\n
RemoteConnections: tcp:129.93.3.4:1234,tcp:[2600:900:6:1301:268a:7ff:fef6:a590]:2345\n
End\n
```

The last response chunk should be either:

```
success: Created
```

or

```
failure: $(ERROR MESSAGE)
```

where `$(ERROR MESSAGE)` is a message explaining the failure, meant to be human readable.

Historically, DPM has a separate success message (fixed in 1.13.0):

```
Success
```

And also failure messages were different (fixed in 1.13.1):

```
Failed: $(ERROR MESSAGE)
```

Clients should accept this for maximum compatibility but new implementations should use `success: Created` as above.

This topic: LCG > HttpTpcTechnical
Topic revision: r15 - 2020-03-25 - PetrVokac