Version: 0.8

ToPoS III Reference Manual

Pieter van Beek BSc.

February 23, 2009

<mailto:topos@hpcv.projects.sara.nl>

SARA's Token Pool Server (ToPoS) is a REST web service that facilitates distribution of tokens across clients. Most commonly, this is used to distribute computational tasks across a compute cluster, compute cloud, Grid, or processes on a super computer. ToPoS supports exclusive locking of tokens, so clients are guaranteed to be the only possessor of of a certain token. Finally, ToPoS supports the creations of processing pipelines and workflow by allowing clients to create and destroy tokens in different token pools in one atomic action.

Contents

1 ToPoS III and HTTP

1.1 Supported methods

ToPoS III uses HTTP[1] as its application protocol[2]. HTTP defines the methods OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT. ToPoS III uses these methods as follows:

GET Used for all read-only access to the server. For tokens URLS, it retrieves the token value of the particular token. On directory-style URLS (recognizable by the trailing slash "/") like token pools and realms, it will return a directory listing (i.e. a list of all tokens) in XHTML[3] format. On all other URLS, it will return whatever is appropriate for that resource.

HEAD Just according to the HTTP/1.1 spec, a HEAD request returns the headers as if the request were a GET request, without any entity body. Available for *every* ToPoS III resource.

POST Used primarily to POST new tokens into a token pool (optionally destroying other tokens at the same time). Additionally, the POST method is used to "masquerade" other HTTP methods; see 1.2 for details.

PUT Used to insert new tokens.

DELETE To delete individual tokens, token pools, and realms from the server.

OPTIONS This method SHOULD return all available methods per resource, but this hasn't been implemented yet.

TRACE and CONNECT are not used.

1.2 Method masquerading

ToPoS III allows users to use the HTTP POST method to "masquerade" other HTTP methods, by specifying an `http_method` query parameter in the request URL. E.g. the following two HTTP requests are semantically identical:

- ```
  DELETE /some_resource HTTP/1.1
  Host: topos.grid.sara.nl
  Date: Mon, 09 Sep 2008 08:17:35 GMT
  ```
- ```
  POST /some_resource?http_method=DELETE HTTP/1.1
  Host: topos.grid.sara.nl
  Date: Mon, 09 Sep 2008 08:17:35 GMT
  Content-Length: 0
  ```

In XHTML, this request could be interfaced with a "delete button", like this:

[1] See RFC2616: Hypertext Transfer Protocol – HTTP/1.1

[2] Programmers that use SOAP may have been lead to believe that HTTP is a *transport protocol*. But it's really an *application protocol*, powering the biggest distributed client–server application in the world: the Web.

[3] See XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition) `http://www.w3.org/TR/2002/REC-xhtml1-20020801`

```
<form action="/some_resource?http_method=DELETE" method="post">
  <input type="submit" value="Delete some_resource"/>
</form>
```

If you spoof an HTTP GET method as a POST method, *and* the `Content-Type` of the request body is `application/x-www-form-urlencoded`, then query parameters of the request body are treated as if they are "GET parameters". E.g. the GET request

```
GET /some_resource?param=value HTTP/1.1
Host: topos.grid.sara.nl
Date: Mon, 09 Sep 2008 08:17:35 GMT
```

can be spoofed as:

```
POST /some_resource?http_method=GET HTTP/1.1
Host: topos.grid.sara.nl
Date: Mon, 09 Sep 2008 08:17:35 GMT
Content-Type: application/x-www-form-urlencoded
Content-Length: 11

param=value
```

Method masquerading is commonly used in two cases:

1. To perform HTTP GET requests where the total length of all query parameters is too long to fit into a URL. Although there are no theoretical limits to the length of a URL, in practice many clients and servers have practical limits, often as small as 64k bytes.
2. To perform any kind of request, other than POST or GET, from within a browser. Unfortunately, most modern browsers only support the HTTP GET and POST methods. So in order to DELETE a resource from within a browser (which is a perfectly reasonable use case), the request will have to be masqueraded.

2 URL space

The text below lists all HTTP methods available per URL, except the GET and HEAD methods, which are supported by all URLs. On directories, GET always returns an XHTML response body with a directory index. For certain directories, this XHTML will contain additional information. For example, if a resource supports other HTTP methods, applicable XHTML forms will be added.

ToPoS III allows both HTTP and HTTPS connections. For performance reasons, the HTTPS connections are limited to low-grade encryption algorithms. Theoretically, if you limit your Web-browser to the use of the highest grade encryption protocols, the browser might complain about not being able to negotiate an appropriate encryption method with the server.

2.1 The root

All URLs start with the root URL: `http(s)://topos.grid.sara.nl/3/`.

GET Returns (links to) some general information about ToPoS III.

2.2 /newRealm

GET Returns an **HTTP/1.1 303 See Other** to `/realms/<realm>/`, thus redirecting the client to a new, uniquely named, empty realm. If you need a new realm URL, this is the preferred method to get one; it will avoid name clashes with your other realms, or (even worse) other users.

2.3 /realms/

GET Lists all existing realms, i.e. all realms that contain one or more tokens. For unprivileged users (probably including you), this URL will return an **HTTP/1.1 403 Forbidden**. This is a security feature, to avoid users finding out about each other's realms. So, it's important to remember your realm names.

2.4 /realms/<realm>/

A ToPoS III realm is a set of token pools. Realms can be used as security contexts, by giving them obscure names (see also 2.2). You can also use realms as workflow instances, when you're modeling Petri-net workflows with ToPoS III.

GET Returns a directory listing in XHTML, and maybe some forms.

DELETE Deletes this realm and all the pools and tokens in it.

POST with request MIME-type `application/x-www-form-urlencoded` and parameter `pool`: this is semantically equivalent to POSTing to `/realms/<realm>/pools/<pool>/` as explained in section 2.9.

2.5 /realms/<realm>/nextToken

This logical URL is used both for uploading new tokens and for retrieving the next available (locked) token.

Tokens are dispatched in order of age, oldest first. If all tokens have been assigned, and another client requests a new token, then ToPoS III will just start over again, assigning the oldest token. This means that, without exclusive locking, tokens may be assigned more than once.

GET If a token is still available, an **HTTP/1.1 303 See Other** status code is returned, with the URL of the token in the `Location` header and in the response body. If no tokens are available, an **HTTP/1.1 404 Not Found** is returned.
Parameters:

pool (optional) By default, GET yields a token from any of the pools within the realm. If you want a token from a specific subset of pools, you can specify query parameter "`pool`" containing a Perl-style regular expression against which to match pool names. Make sure to properly URL-encode the regular expression, i.e. `%2A` (for "`*`"), `%3F` (for "`?`") etcetera. An example regular expression is `^pool[012]$`, in which case the (relative) URL would be `/realms/myRealm/nextToken?pool=%5Epool%5B012%5D%24`.

token (optional) A Perl-style regular expression against which to match token values. If you have a lot of tokens, this is a time-consuming operation!

timeout (optional) If you specify this parameter, a locked token is returned. The parameter value is the lock timeout in seconds. If this query parameter is specified, two extra response headers are returned:

Topos-OpaqueLockToken a WebDAV-style opaquelocktoken

Topos-LockURL the URL of the lock. You must use this URL to prolong the token lifespan, as explained in section 3.2.

PUT Upload a token to a token pool. If you specify a `Content-Type` header in your request body, then this MIME-type will be remembered for the token.
URL query parameters:

pool (required) The pool in which to put the token.

delete (optional) a comma-separated list of token names to delete. Alternatively, you can specify parameter `delete[]` (i.e. with straight brackets) multiple times.

2.6 /realms/<realm>/locks/

GET Returns a directory listing in XHTML, and maybe some forms.

2.7 /realms/<realm>/locks/<lock>

GET Returns some info about an exclusive lock. By specifying a `timeout` query parameter, you can set a new timeout for the lock.

DELETE Deletes the lock.

2.8 /realms/<realm>/pools/

GET Returns a directory listing in XHTML, and maybe some forms.

2.9 /realms/<realm>/pools/<pool>/

GET Returns a directory listing in XHTML, and maybe some forms.

DELETE Deletes a token pool and all tokens in it.

POST with `Content-Type: multipart/form-data`: This method is used to create and delete multiple tokens in one atomic request. All the new tokens will be created in the token pool directory that the POST request is sent to.
Parameters:

delete (optional) A comma separated list of token names to delete. Alternatively, you can specify parameter `delete[]` (i.e. with straight brackets) multiple times. This parameter may be both a query parameter (i.e. in the URL) or a `multipart/form-data` parameter (i.e. inside the POST request body).

POST with `Content-Type: application/x-www-form-urlencoded`: Allows you to populate a pool with (a large number of) tokens. The tokens will have `Content-Type: text/plain`, and will contain a unique number, ranging from 0 to $n-1$, with n the number of created tokens. Currently, the maximum number of tokens you can create this way is limited to 10^6 for performance reasons. If you need more, contact us. Parameters:

tokens The number of tokens to create.

2.10 /realms/<realm>/pools/<pool>/nextToken

This URL is semantically identical to `/realms/<realm>/nextToken?pool=<pool>` described in section 2.5, except that `<pool>` isn't considered to be a regexp but just a literal pool name.

2.11 /realms/<realm>/pools/<pool>/progress

This resource supports HTTP content negotiation; By default, this resource returns XHTML, but if the client specifically requests a text/plain response (e.g. by issuing an `Accept: text/plain` request header), this resource just returns some plain text.

GET by default displays an XHTML progress bar, indicating which percentage of all tokens in the current realm is in the current pool.

GET with response `Content-Type: text/plain`: returns the number of tokens in this pool, divided by the total number of tokens in this realm.

Query parameters:

total (optional) the number of tokens by which to divide the number of tokens in this pool. Defaults to the total number of tokens in the current realm.

width (optional, for XHTML only) The width of the progress bar, in pixels.

2.12 /realms/<realm>/pools/<pool>/tokens/

GET Returns a directory listing in XHTML, and maybe some forms.

2.13 /realms/<realm>/pools/<pool>/tokens/<token>

GET Returns the token value.

DELETE Deletes the token.

3 Shared vs. exclusive tokens

ToPoS III allows two methods of operation for clients: shared tokens and exclusive tokens. When using exclusive tokens, the client can be absolutely sure that it is the only holder of the token. When using shared tokens, the client must take into account the possibility that other clients are "processing" the same token concurrently.

3.1 Using shared tokens

To use shared tokens, you don't have to do anything special, as this is the default mode of operation.

A client retrieves a token URL from the `/realms/<realm>/nextToken` resource, and starts operating on the token. The client must be aware of the fact that other processes might be operating on the same token. When the client tries to destroy the token after processing, it may find out that the token has already been deleted by another process.

3.2 Using exclusive tokens

If you don't want tokens to be dispatched more than once, you can use an exclusive locking mechanism on tokens. This mode of operation works as follows:

1. Get a new token URL from the `/realms/<realm>/nextToken` resource. If no new token is available, quit.
2. Lock the resource for n seconds. If locking fails, go back to 1.[4]
3. Operate on the token, refreshing the lock at least every n seconds. If refreshing fails, go to 1.[5]
4. Move or destroy the token. Again, this operation might fail.
5. Optionally, restart at 1.

[4] Locking might fail because the token was deleted (returns **HTTP/1.1 404 Not Found**) or locked by another actor (returns **HTTP/1.1 412 Precondition Failed**).

[5] Refreshing may fail because of network timeouts, or because some actor (probably you) destroyed/moved the resource despite its lock (returns **HTTP/1.1 404 Not Found**).

3.3 Pros and cons of exclusive and shared tokens

- Using shared tokens is much simpler than using exclusive tokens.
- Shared tokens require clients to be programmed "thread-safe", i.e. it shouldn't matter if more than one client is operating on the same token. If making clients thread-safe requires more effort than using exclusive tokens, you should consider using exclusive tokens.
- When using shared tokens, you'll normally waste some resources because multiple clients might operate on the same token. The estimated "overhead" of using shared tokens is $c/2t$ with c the number of clients and t the number of tokens.[6] If this overhead is unacceptable to you, use exclusive tokens.
- Sometimes, the possibility of having more than one client operate on a single token can be an advantage. When using a Grid-, cluster-, cloud-, or super-computing, some processes will run much slower than other processes, because they're running on a defective node. Such a slow or even defective processor may keep a token locked indefinitely. When using pipelines or processing synchronization, this may kill overall throughput. Using shared tokens allows "fast" nodes to start operating on tokens that have previously been assigned to a defective node, *increasing* the overall throughput instead of wasting resources.

4 Using `curl` as a ToPoS III client

`Curl` is both an HTTP library (libcurl) and a command line HTTP client. It comes with an excellent man-page. The information below is provided as a quick reference for usage with ToPoS III.

4.1 Options of general use

The following `curl` options might be handy when working with ToPoS III:

--cacert <CA certificate> Tells `curl` to use the specified certificate file to verify the peer. The file may contain multiple CA certificates. The certificate(s) must be in PEM format. `Curl` recognizes the environment variable named CURL_CA_BUNDLE if that is set, and uses the given path as a path to a CA cert bundle. This option overrides that variable.

--capath <CA certificate directory> Tells `curl` to use the specified certificate directory to verify the peer. The certificates must be in PEM format, and the directory must have been processed using the c_rehash utility supplied with openssl.

-d/--data/--data-ascii <data> Sends the specified data in a POST request to the HTTP server, in a way that can emulate as if a user has filled in a HTML form and pressed the submit button. Note that the data is sent exactly as specified with no extra processing (with all newlines cut off). The data is expected to be URL-encoded. This will cause `curl` to pass the data to the server using the content-type `application/x-www-form-urlencoded`. Compare to -F/--form. If this option is used more than once on the same command line, the data pieces specified will be merged together with a separating &-letter. Thus, using `-d name=daniel -d skill=lousy` would generate a post chunk that looks like `name=daniel&skill=lousy`.
If you start the data with the letter @, the rest should be a file name to read the data from, or - if you want `curl` to read the data from stdin. The contents of the file must already be URL-encoded. Multiple files can also be specified. Posting data from a file named "foobar" would thus be done with `--data @foobar`.
To post data purely binary, you should instead use the –data-binary option.
If this option is used several times, the ones following the first will append data.

[6] So when letting 10 clients operate on 1,000 tokens, the overhead is approximately $10/(2 \cdot 1{,}000) = 0.5\%$.

--data-binary <data> This posts data in a similar manner as --data-ascii does, although when using this option the entire con- text of the posted data is kept as-is. If you want to post a binary file without the strip-newlines feature of the --data-ascii option, this is for you.
If this option is used several times, the ones following the first will append data.

--compressed Request a compressed response using one of the algorithms libcurl supports, and return the uncompressed document. If this option is used and the server sends an unsupported encoding, `Curl` will report an error.
If this option is used several times, each occurrence will toggle it on/off.

-D/--dump-header <file> Write the protocol headers to the specified file.

-F/--form <name=content> This lets `curl` emulate a filled in form in which a user has pressed the submit button. This causes `curl` to POST data using the Content-Type `multipart/form-data` according to RFC1867. This enables uploading of binary files etc. To force the 'content' part to be a file, prefix the file name with an @ sign. To just get the content part from a file, prefix the file name with the letter <. The difference between @ and < is then that @ makes a file get attached in the post as a file upload, while the < makes a text field and just get the contents for that text field from a file.

--form-string <name=string> Similar to --form except that the value string for the named parameter is used literally. Leading '@' and '<' characters, and the ';type=' string in the value have no special meaning. Use this in preference to --form if there's any possibility that the string value may accidentally trigger the '@' or '<' features of --form.

-G/--get When used, this option will make all data specified with -d/--data or --data-binary to be used in a HTTP GET request instead of the POST request that otherwise would be used. The data will be appended to the URL with a '?' separator.
If used in combination with -I, the POST data will instead be appended to the URL with a HEAD request.

-H/--header <header> Extra header to use in the HTTP request.

-i/--include Include the HTTP-header in the output.

-I/--head Fetch the HTTP-header only!

-k/--insecure Don't check server certificate validity. The ToPoS III server certificate was issued by the DjinnIT CA, which may not be in your standard repository of trusted CAS.

-L/--location If the server reports that the requested page has moved to a different location (indicated with a Location: header and a 3xx response code) this option will make `curl` redo the request on the new place. If used together with -i/--include or -I/--head, headers from all requested pages will be shown.
When `curl` follows a redirect and the request is not a plain GET (for example POST or PUT), it will do the following request with a GET if the HTTP response was 301, 302, or 303. If the response code was any other 3xx code, `curl` will re-send the following request using the same unmodified method.

-o/--output <file> Write output to <file> instead of stdout.

-O/--remote-name Write output to a local file named like the remote file we get. (Only the file part of the remote file is used, the path is cut off.)

-R/--remote-time When used, this will make libcurl attempt to figure out the timestamp of the remote file, and if that is available make the local file get that same timestamp.

-s/--silent Silent mode. Don't show progress meter or error messages. Makes `curl` mute.

-T/--upload-file <file> This transfers the specified local file to the remote URL using the HTTP PUT method. If there is no file part in the specified URL, `curl` will append the local file name. You must use a trailing / on the last directory to really prove to `curl` that there is no file name or

curl will think that your last directory name is the remote file name to use. That will most likely cause the upload operation to fail. Use the file name "-" (a single dash) to use stdin instead of a given file.

-X/--request <command> Specifies a custom request method to use when communicating with the HTTP server.

4.1.1 Error handling options

--connect-timeout <seconds> Maximum time in seconds that you allow the connection to the server to take. This only limits the connection phase, once curl has connected this option is of no more use. See also the -m/--max-time option.
If this option is used several times, the last one will be used.

-f/--fail Fail silently (no output at all) on server errors. This is mostly done like this to better enable scripts etc to better deal with failed attempts. In normal cases when a HTTP server fails to deliver a document, it returns an XHTML document stating so (which often also describes why and more). This flag will prevent curl from outputting that and return error 22.

-m/--max-time <seconds> Maximum time in seconds that you allow the whole operation to take. This is useful for preventing your batch jobs from hanging for hours due to slow networks or links going down. See also the --connect-timeout option.

--retry <num> If a transient error is returned when curl tries to perform a transfer, it will retry this number of times before giving up. Transient error means an HTTP 5xx response code.
When curl is about to retry a transfer, it will first wait one second and then for all forthcoming retries it will double the waiting time until it reaches 10 minutes which then will be the delay between the rest of the retries. By using --retry-delay you disable this exponential backoff algorithm. See also --retry-max-time to limit the total time allowed for retries.

--retry-delay <seconds> Make curl sleep this amount of time between each retry when a transfer has failed with a transient error. This option is only interesting if --retry is also used.

--retry-max-time <seconds> The retry timer is reset before the first transfer attempt. Retries will be done as usual (see --retry) as long as the timer hasn't reached this given limit. Notice that if the timer hasn't reached the limit, the request will be made and while performing, it may take longer than this given time period.

-S/--show-error When used with -s it makes curl show error message if it fails.

4.2 Curl examples

4.2.1 Getting a new realm URL

```
#!/bin/bash
TOPOS_ROOT=https://topos.grid.sara.nl/3/
```

The safest way to get a new realm URL is to get one from the system:

```
MYREALM=$( curl -s -H "Accept: text/plain" ${TOPOS_ROOT}newRealm )
echo ${MYREALM}
https://topos.grid.sara.nl/3/realms/c72ee09eccf583d86412b406/
```

Alternatively, you could make up your own realm URL which is easier to remember (but also easier to guess by crackers!):

```
MYREALM=${TOPOS_ROOT}realms/myRealm/
```

If curl complains about the SSL handshake, use the -k option to make curl ignore the problem, or (better) put the DutchGrid CA Root Certificate[7] in your directory of trusted CA's, and make curl look in that directory. E.g. you could put the line

[7] See http://ca.dutchgrid.nl/medium/cacert.pem

```
capath=/etc/ssl/certs/
```

in your ~/.curlrc.

4.2.2 Initializing a pool with a set of unique numbers

To initialize a pool named "pool_1" with 1000 tokens, with each token containing a unique number from 0 to 999 inclusive:

```
curl -f -s -d tokens=1000 ${MYREALM}pools/pool_1/ \
  >/dev/null || echo 'Oops!'
```

The -f option allows you to trap run-time errors in your bash scripts. The HTTP-response is sent to /dev/null, because it's of no real interest.

4.2.3 Uploading a single file to a token pool

To upload file file.txt to a pool named pool_2, you could do the following:

```
curl -s -i -T file.txt -H "Content-Type: text/plain" \
  ${MYREALM}pools/pool_2/nextToken \
  | grep '^Location:'
Location: https://topos.grid.sara.nl/3/realms/myRealm/pool/pool_2/tokens/1234
```

Note that the content-type of the file is specified. Later, when the token is retrieved, this Content-Type header will be reproduced by the server. Also, the server returns the URL of the newly created resource in the Location: response header, as per HTTP specs.

4.2.4 Uploading a single file and deleting a token in one atomic transaction

To upload file file.txt to a pool named pool_2, and at the same time delete token 1234 in one of your pools, you could do the following:

```
curl -s -i -T file.txt -H "Content-Type: text/plain" \
  ${MYREALM}pools/pool_2/nextToken?delete=1234 \
  | grep '^Location:'
Location: https://topos.grid.sara.nl/3/realms/myRealm/pool/pool_2/tokens/1235
```

4.2.5 Uploading multiple files to a token pool

Of course, you could upload a lot of files by putting the single file-upload statement above inside some loop construct in a shell script. But this would require a new HTTP request and response for each file. It's much faster to do it with a single HTTP request, using a multipart/form-data request body:

```
curl -H "Accept:  text/tdv" \
  -F "file[]=@file1.jpg;type=image/jpeg" \
  -F "file[]=@file2.jpg;type=image/jpeg" \
  ${MYREALM}pools/pool_3/
1234    file1.jpg
1235    file2.jpg
```

In this example, the client asks for a particular content-type in the response body: text/tdv (AKA tab-delimited values). The default content-type is HTML, and you can also request text/csv (AKA comma-separated values).

4.2.6 Uploading and deleting tokens in one atomic transaction

```
curl -f -H "Accept:  text/tdv" \
  -F "file[]=@file3.jpg;type=image/jpeg" \
  -F "file[]=@file4.jpg;type=image/jpeg" \
  -F "delete=1234,1235" \
  ${MYREALM}pools/pool_3/ || echo "Oops!"
1236    file3.jpg
1237    file4.jpg
```

4.2.7 Deleting a token

```
curl -s -f -X DELETE ${MYREALM}pools/pool_3/tokens/1236 \
  >/dev/null || echo "Oops!"
```

4.2.8 Deleting a token pool

```
curl -s -f -X DELETE ${MYREALM}pools/pool_3/ \
  >/dev/null || echo "Oops!"
```

4.2.9 Deleting a token realm

```
curl -s -f -X DELETE ${MYREALM} \
  >/dev/null || echo "Oops!"
```

5 Future features

One-shot URLs Currently, if you upload now tokens to the server (either by PUTting to a `nextToken` resource or by POSTing to a pool directory resource) and you get no response from the server, there's no way to tell if the token(s) have been created. In REST web services, this is commonly solved by so called "one-shot URLs".

API's for bash, perl and java.

6 Acknowledgements

ToPoS III is an implementation of the well-known Koot-Vermin Algorithm for Distributed Computing. This work would not have been possible without the inspiring comments of Willem J. Vermin, PhD.