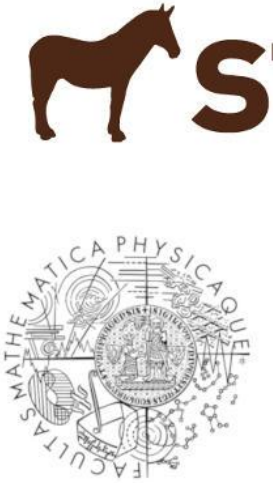
curl://up



FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

wolfSSL

Internals: walking through a transfer

Items

Concepts

Fundamental structs

Protocol handlers

The multi state machine

Connection caching

Concepts

libcurl is oriented around “transfers”

A transfer equals a “CURL *handle” in an application

A “CURL *handle” is “struct Curl_easy” inside the library (always stored in the variable ‘data’)

curl_easy_init() is basically just allocating a “struct Curl_easy”

Everything* internally is made non-blocking

curl_easy_perform() is just a wrapper around curl_multi_perform()

* = Not 100% true but it *should* be true!

The simplest curl program

```c
#include <stdio.h>
#include <curl/curl.h>

int main(void)
{
  CURL *curl;
  CURLcode res;

  curl = curl_easy_init();
  if(curl) {
    curl_easy_setopt(curl, CURLOPT_URL, "https://example.com");
    /* example.com is redirected, so we tell libcurl to follow redirection */
    curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);

    /* Perform the request, res will get the return code */
    res = curl_easy_perform(curl);
    /* Check for errors */
    if(res != CURLE_OK)
      fprintf(stderr, "curl_easy_perform() failed: %s\n",
              curl_easy_strerror(res));

    /* always cleanup */
    curl_easy_cleanup(curl);
  }
  return 0;
}
```

The multi interface

Any amount of parallel transfers

Single thread

Protocol agnostic

The multi interface, setup

```
easy1 = curl_easy_init();
curl_easy_setopt(e1, CURLOPT_…, …);

easy2 = curl_easy_init();
curl_easy_setopt(e2, CURLOPT_…, …);

multi = curl_multi_init();

curl_multi_add_handle(multi, easy1);

curl_multi_add_handle(multi, easy2);
```

The multi interface, transfer

```
do {
  curl_multi_perform();
  curl_multi_wait();
  if(curl_multi_info_read(&msgs, &left))
    Something();
} while (!done);
```

The multi interface, shutting down

```
curl_multi_remove_handle(multi, easy1);
curl_multi_remove_handle(multi, easy2);
curl_easy_cleanup(easy1);
curl_easy_cleanup(easy2);
curl_multi_cleanup(multi);
```

curl_easy_perform()

Creates a multi handle: curl_multi_init()

Adds the easy handle to multi: curl_multi_add_handle()

Runs the loop until transfer is done:

curl_multi_wait()

curl_multi_perform()

Then calls curl_multi_remove_handle()

Done

curl_multi_perform()

Handles any amount of concurrent transfers

Loops over all handles and invokes multi_runsingle() for each of them.

A sorted tree of timeouts knows the nearest timeout, for curl_multi_timeout() and more

multi_runsingle()

A state machine ‘data→mstate’.

All transfers start in INIT

It goes to CONNECT where it initializes a new connection or finds an existing to reuse.

Often, it initiates a name resolve there and switches to the WAITRESOLVE state.

It remains in WAITRESOLVE until the name is resolved (or failed). If successful, it initiates the connection and goes to WAITCONNECT.

It sits in WAITCONNECT until the connection succeeds.

DO is the state where it issues its request

PERFORM is the “payload transfer” phase

DONE is after the transfer is completed

Getting a connection

“struct connectdata” is for a connection (variables for this are named ‘conn’ in the code)

libcurl maintains a “connection cache” of previosly used connections (that weren’t closed)

When a new transfer is to be made, a check is made if an already existing connection can be used.

Yes that’s a fairly complicated check

Protocol handlers

Given the scheme in the URL for a transfer, libcurl will set ‘conn→handler’ to a dedicated struct Curl_handler for that protocol.

Curl_handler holds a set of function pointers for protocol specific functionality.

conn→handler→functionality() is then used from generic code.

Hm

curl://

curl://up

stickermule

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

wolfSSL