NAME

libcurl-tutorial − libcurl programming tutorial

Objective

This document attempts to describe the general principles and some basic approaches to consider when programming with libcurl. The text will focus mainly on the C interface but might apply fairly well on other interfaces as well as they usually follow the C one pretty closely.

This document will refer to 'the user' as the person writing the source code that uses libcurl. That would probably be you or someone in your position. What will be generally referred to as 'the program' will be the collected source code that you write that is using libcurl for transfers. The program is outside libcurl and libcurl is outside of the program.

To get more details on all options and functions described herein, please refer to their respective man pages.

Building

There are many different ways to build C programs. This chapter will assume a UNIX-style build process. If you use a different build system, you can still read this to get general information that may apply to your environment as well.

Compiling the Program

Your compiler needs to know where the libcurl headers are located. Therefore you must set your compiler's include path to point to the directory where you installed them. The 'curl-config'[3] tool can be used to get this information:

```
$ curl-config --cflags
```

Linking the Program with libcurl

When having compiled the program, you need to link your object files to create a single executable. For that to succeed, you need to link with libcurl and possibly also with other libraries that libcurl itself depends on. Like the OpenSSL libraries, but even some standard OS libraries may be needed on the command line. To figure out which flags to use, once again the 'curl-config' tool comes to the rescue:

```
$ curl-config --libs
```

SSL or Not

libcurl can be built and customized in many ways. One of the things that varies from different libraries and builds is the support for SSL-based transfers, like HTTPS and FTPS. If a supported SSL library was detected properly at build-time, libcurl will be built with SSL support. To figure out if an installed libcurl has been built with SSL support enabled, use 'curl-config' like this:

```
$ curl-config --feature
```

And if SSL is supported, the keyword 'SSL' will be written to stdout, possibly together with a few other features that could be either on or off on for different libcurls.

See also the "Features libcurl Provides" further down.

autoconf macro

When you write your configure script to detect libcurl and setup variables accordingly, we offer a prewritten macro that probably does everything you need in this area. See docs/libcurl/libcurl.m4 file - it includes docs on how to use it.

Portable Code in a Portable World

The people behind libcurl have put a considerable effort to make libcurl work on a large amount of different operating systems and environments.

You program libcurl the same way on all platforms that libcurl runs on. There are only very few minor considerations that differ. If you just make sure to write your code portable enough, you may very well create yourself a very portable program. libcurl shouldn't stop you from that.

Global Preparation

The program must initialize some of the libcurl functionality globally. That means it should be done exactly once, no matter how many times you intend to use the library. Once for your program's entire life time. This is done using

curl_global_init()

and it takes one parameter which is a bit pattern that tells libcurl what to initialize. Using *CURL_GLOBAL_ALL* will make it initialize all known internal sub modules, and might be a good default option. The current two bits that are specified are:

CURL_GLOBAL_WIN32
: which only does anything on Windows machines. When used on a Windows machine, it'll make libcurl initialize the win32 socket stuff. Without having that initialized properly, your program cannot use sockets properly. You should only do this once for each application, so if your program already does this or of another library in use does it, you should not tell libcurl to do this as well.

CURL_GLOBAL_SSL
: which only does anything on libcurls compiled and built SSL-enabled. On these systems, this will make libcurl initialize the SSL library properly for this application. This only needs to be done once for each application so if your program or another library already does this, this bit should not be needed.

libcurl has a default protection mechanism that detects if *curl_global_init(3)* hasn't been called by the time *curl_easy_perform(3)* is called and if that is the case, libcurl runs the function itself with a guessed bit pattern. Please note that depending solely on this is not considered nice nor very good.

When the program no longer uses libcurl, it should call *curl_global_cleanup(3)*, which is the opposite of the init call. It will then do the reversed operations to cleanup the resources the *curl_global_init(3)* call initialized.

Repeated calls to *curl_global_init(3)* and *curl_global_cleanup(3)* should be avoided. They should only be called once each.

Features libcurl Provides

It is considered best-practice to determine libcurl features at run-time rather than at build-time (if possible of course). By calling *curl_version_info(3)* and checking out the details of the returned struct, your program can figure out exactly what the currently running libcurl supports.

Handle the Easy libcurl

libcurl first introduced the so called easy interface. All operations in the easy interface are prefixed with 'curl_easy'.

Recent libcurl versions also offer the multi interface. More about that interface, what it is targeted for and how to use it is detailed in a separate chapter further down. You still need to understand the easy interface

first, so please continue reading for better understanding.

To use the easy interface, you must first create yourself an easy handle. You need one handle for each easy session you want to perform. Basically, you should use one handle for every thread you plan to use for transferring. You must never share the same handle in multiple threads.

Get an easy handle with

```
easyhandle = curl_easy_init();
```

It returns an easy handle. Using that you proceed to the next step: setting up your preferred actions. A handle is just a logic entity for the upcoming transfer or series of transfers.

You set properties and options for this handle using *curl_easy_setopt(3)*. They control how the subsequent transfer or transfers will be made. Options remain set in the handle until set again to something different. Alas, multiple requests using the same handle will use the same options.

Many of the options you set in libcurl are "strings", pointers to data terminated with a zero byte. When you set strings with *curl_easy_setopt(3)*, libcurl makes its own copy so that they don't need to be kept around in your application after being set[4].

One of the most basic properties to set in the handle is the URL. You set your preferred URL to transfer with CURLOPT_URL in a manner similar to:

```
curl_easy_setopt(handle, CURLOPT_URL, "http://domain.com/");
```

Let's assume for a while that you want to receive data as the URL identifies a remote resource you want to get here. Since you write a sort of application that needs this transfer, I assume that you would like to get the data passed to you directly instead of simply getting it passed to stdout. So, you write your own function that matches this prototype:

```
size_t write_data(void *buffer, size_t size, size_t nmemb, void *userp);
```

You tell libcurl to pass all data to this function by issuing a function similar to this:

```
curl_easy_setopt(easyhandle, CURLOPT_WRITEFUNCTION, write_data);
```

You can control what data your callback function gets in the fourth argument by setting another property:

```
curl_easy_setopt(easyhandle, CURLOPT_WRITEDATA, &internal_struct);
```

Using that property, you can easily pass local data between your application and the function that gets invoked by libcurl. libcurl itself won't touch the data you pass with *CURLOPT_WRITEDATA*.

libcurl offers its own default internal callback that will take care of the data if you don't set the callback with *CURLOPT_WRITEFUNCTION*. It will then simply output the received data to stdout. You can have the default callback write the data to a different file handle by passing a 'FILE *' to a file opened for writing with the *CURLOPT_WRITEDATA* option.

Now, we need to take a step back and have a deep breath. Here's one of those rare platform-dependent nitpicks. Did you spot it? On some platforms[2], libcurl won't be able to operate on files opened by the program. Thus, if you use the default callback and pass in an open file with *CURLOPT_WRITEDATA*, it will crash. You should therefore avoid this to make your program run fine virtually everywhere.

(*CURLOPT_WRITEDATA* was formerly known as *CURLOPT_FILE*. Both names still work and do the same thing).

If you're using libcurl as a win32 DLL, you MUST use the *CURLOPT_WRITEFUNCTION* if you set *CURLOPT_WRITEDATA* - or you will experience crashes.

There are of course many more options you can set, and we'll get back to a few of them later. Let's instead continue to the actual transfer:

```
success = curl_easy_perform(easyhandle);
```

curl_easy_perform(3) will connect to the remote site, do the necessary commands and receive the transfer. Whenever it receives data, it calls the callback function we previously set. The function may get one byte at a time, or it may get many kilobytes at once. libcurl delivers as much as possible as often as possible. Your callback function should return the number of bytes it "took care of". If that is not the exact same amount of bytes that was passed to it, libcurl will abort the operation and return with an error code.

When the transfer is complete, the function returns a return code that informs you if it succeeded in its mission or not. If a return code isn't enough for you, you can use the CURLOPT_ERRORBUFFER to point libcurl to a buffer of yours where it'll store a human readable error message as well.

If you then want to transfer another file, the handle is ready to be used again. Mind you, it is even preferred that you re-use an existing handle if you intend to make another transfer. libcurl will then attempt to re-use the previous connection.

For some protocols, downloading a file can involve a complicated process of logging in, setting the transfer mode, changing the current directory and finally transferring the file data. libcurl takes care of all that complication for you. Given simply the URL to a file, libcurl will take care of all the details needed to get the file moved from one machine to another.

Multi-threading Issues

The first basic rule is that you must **never** simultaneously share a libcurl handle (be it easy or multi or whatever) between multiple threads. Only use one handle in one thread at any time. You can pass the handles around among threads, but you must never use a single handle from more than one thread at any given time.

libcurl is completely thread safe, except for two issues: signals and SSL/TLS handlers. Signals are used for timing out name resolves (during DNS lookup) - when built without c-ares support and not on Windows.

If you are accessing HTTPS or FTPS URLs in a multi-threaded manner, you are then of course using the underlying SSL library multi-threaded and those libs might have their own requirements on this issue. Basically, you need to provide one or two functions to allow it to function properly. For all details, see this:

OpenSSL

http://www.openssl.org/docs/crypto/threads.html#DESCRIPTION

GnuTLS

http://www.gnu.org/software/gnutls/manual/html_node/Multi_002dthreaded-applications.html

NSS

is claimed to be thread-safe already without anything required.

PolarSSL

Required actions unknown.

yassl

Required actions unknown.

axTLS

Required actions unknown.

Secure Transport

The engine is fully thread-safe, and no additional steps are required.

When using multiple threads you should set the CURLOPT_NOSIGNAL option to 1 for all handles. Everything will or might work fine except that timeouts are not honored during the DNS lookup - which you can work around by building libcurl with c-ares support. c-ares is a library that provides asynchronous name resolves. On some platforms, libcurl simply will not function properly multi-threaded unless this option is set.

Also, note that CURLOPT_DNS_USE_GLOBAL_CACHE is not thread-safe.

When It Doesn't Work

There will always be times when the transfer fails for some reason. You might have set the wrong libcurl option or misunderstood what the libcurl option actually does, or the remote server might return non-standard replies that confuse the library which then confuses your program.

There's one golden rule when these things occur: set the CURLOPT_VERBOSE option to 1. It'll cause the library to spew out the entire protocol details it sends, some internal info and some received protocol data as well (especially when using FTP). If you're using HTTP, adding the headers in the received output to study is also a clever way to get a better understanding why the server behaves the way it does. Include headers in the normal body output with CURLOPT_HEADER set 1.

Of course, there are bugs left. We need to know about them to be able to fix them, so we're quite dependent on your bug reports! When you do report suspected bugs in libcurl, please include as many details as you possibly can: a protocol dump that CURLOPT_VERBOSE produces, library version, as much as possible of your code that uses libcurl, operating system name and version, compiler name and version etc.

If CURLOPT_VERBOSE is not enough, you increase the level of debug data your application receive by using the CURLOPT_DEBUGFUNCTION.

Getting some in-depth knowledge about the protocols involved is never wrong, and if you're trying to do funny things, you might very well understand libcurl and how to use it better if you study the appropriate RFC documents at least briefly.

Upload Data to a Remote Site

libcurl tries to keep a protocol independent approach to most transfers, thus uploading to a remote FTP site is very similar to uploading data to a HTTP server with a PUT request.

Of course, first you either create an easy handle or you re-use one existing one. Then you set the URL to operate on just like before. This is the remote URL, that we now will upload.

Since we write an application, we most likely want libcurl to get the upload data by asking us for it. To make it do that, we set the read callback and the custom pointer libcurl will pass to our read callback. The read callback should have a prototype similar to:

```
size_t function(char *bufptr, size_t size, size_t nitems, void *userp);
```

Where bufptr is the pointer to a buffer we fill in with data to upload and size*nitems is the size of the buffer and therefore also the maximum amount of data we can return to libcurl in this call. The 'userp' pointer is the custom pointer we set to point to a struct of ours to pass private data between the application and the callback.

```
curl_easy_setopt(easyhandle, CURLOPT_READFUNCTION, read_function);

curl_easy_setopt(easyhandle, CURLOPT_READDATA, &filedata);
```

Tell libcurl that we want to upload:

```
curl_easy_setopt(easyhandle, CURLOPT_UPLOAD, 1L);
```

A few protocols won't behave properly when uploads are done without any prior knowledge of the expected file size. So, set the upload file size using the CURLOPT_INFILESIZE_LARGE for all known file sizes like this[1]:

```
/* in this example, file_size must be an curl_off_t variable */
curl_easy_setopt(easyhandle, CURLOPT_INFILESIZE_LARGE, file_size);
```

When you call *curl_easy_perform(3)* this time, it'll perform all the necessary operations and when it has invoked the upload it'll call your supplied callback to get the data to upload. The program should return as much data as possible in every invoke, as that is likely to make the upload perform as fast as possible. The callback should return the number of bytes it wrote in the buffer. Returning 0 will signal the end of the upload.

Passwords

Many protocols use or even require that user name and password are provided to be able to download or upload the data of your choice. libcurl offers several ways to specify them.

Most protocols support that you specify the name and password in the URL itself. libcurl will detect this and use them accordingly. This is written like this:

```
protocol://user:password@example.com/path/
```

If you need any odd letters in your user name or password, you should enter them URL encoded, as %XX where XX is a two-digit hexadecimal number.

libcurl also provides options to set various passwords. The user name and password as shown embedded in the URL can instead get set with the CURLOPT_USERPWD option. The argument passed to libcurl should be a char * to a string in the format "user:password". In a manner like this:

```
curl_easy_setopt(easyhandle, CURLOPT_USERPWD, "myname:thesecret");
```

Another case where name and password might be needed at times, is for those users who need to authenticate themselves to a proxy they use. libcurl offers another option for this, the CURLOPT_PROXYUSERPWD. It is used quite similar to the CURLOPT_USERPWD option like this:

```
curl_easy_setopt(easyhandle, CURLOPT_PROXYUSERPWD, "myname:thesecret");
```

There's a long time UNIX "standard" way of storing ftp user names and passwords, namely in the $HOME/.netrc file. The file should be made private so that only the user may read it (see also the "Security Considerations" chapter), as it might contain the password in plain text. libcurl has the ability to use this file to figure out what set of user name and password to use for a particular host. As an extension to the normal functionality, libcurl also supports this file for non-FTP protocols such as HTTP. To make curl use this file, use the CURLOPT_NETRC option:

```
curl_easy_setopt(easyhandle, CURLOPT_NETRC, 1L);
```

And a very basic example of how such a .netrc file may look like:

```
machine myhost.mydomain.com
login userlogin
password secretword
```

All these examples have been cases where the password has been optional, or at least you could leave it out and have libcurl attempt to do its job without it. There are times when the password isn't optional, like when you're using an SSL private key for secure transfers.

To pass the known private key password to libcurl:

```
curl_easy_setopt(easyhandle, CURLOPT_KEYPASSWD, "keypassword");
```

HTTP Authentication

The previous chapter showed how to set user name and password for getting URLs that require authentication. When using the HTTP protocol, there are many different ways a client can provide those credentials to the server and you can control which way libcurl will (attempt to) use them. The default HTTP authentication method is called 'Basic', which is sending the name and password in clear-text in the HTTP request, base64-encoded. This is insecure.

At the time of this writing, libcurl can be built to use: Basic, Digest, NTLM, Negotiate, GSS-Negotiate and SPNEGO. You can tell libcurl which one to use with CURLOPT_HTTPAUTH as in:

```
curl_easy_setopt(easyhandle, CURLOPT_HTTPAUTH, CURLAUTH_DIGEST);
```

And when you send authentication to a proxy, you can also set authentication type the same way but instead with CURLOPT_PROXYAUTH:

```
curl_easy_setopt(easyhandle, CURLOPT_PROXYAUTH, CURLAUTH_NTLM);
```

Both these options allow you to set multiple types (by ORing them together), to make libcurl pick the most secure one out of the types the server/proxy claims to support. This method does however add a round-trip since libcurl must first ask the server what it supports:

```
curl_easy_setopt(easyhandle, CURLOPT_HTTPAUTH,
CURLAUTH_DIGEST|CURLAUTH_BASIC);
```

For convenience, you can use the 'CURLAUTH_ANY' define (instead of a list with specific types) which allows libcurl to use whatever method it wants.

When asking for multiple types, libcurl will pick the available one it considers "best" in its own internal order of preference.

HTTP POSTing

We get many questions regarding how to issue HTTP POSTs with libcurl the proper way. This chapter will thus include examples using both different versions of HTTP POST that libcurl supports.

The first version is the simple POST, the most common version, that most HTML pages using the <form> tag uses. We provide a pointer to the data and tell libcurl to post it all to the remote site:

```
char *data="name=daniel&project=curl";
curl_easy_setopt(easyhandle, CURLOPT_POSTFIELDS, data);
curl_easy_setopt(easyhandle, CURLOPT_URL, "http://posthere.com/");

curl_easy_perform(easyhandle); /* post away! */
```

Simple enough, huh? Since you set the POST options with the CURLOPT_POSTFIELDS, this automatically switches the handle to use POST in the upcoming request.

Ok, so what if you want to post binary data that also requires you to set the Content-Type: header of the post? Well, binary posts prevent libcurl from being able to do strlen() on the data to figure out the size, so therefore we must tell libcurl the size of the post data. Setting headers in libcurl requests are done in a generic way, by building a list of our own headers and then passing that list to libcurl.

```
struct curl_slist *headers=NULL;
headers = curl_slist_append(headers, "Content-Type: text/xml");

/* post binary data */
curl_easy_setopt(easyhandle, CURLOPT_POSTFIELDS, binaryptr);

/* set the size of the postfields data */
curl_easy_setopt(easyhandle, CURLOPT_POSTFIELDSIZE, 23L);

/* pass our list of custom made headers */
curl_easy_setopt(easyhandle, CURLOPT_HTTPHEADER, headers);

curl_easy_perform(easyhandle); /* post away! */

curl_slist_free_all(headers); /* free the header list */
```

While the simple examples above cover the majority of all cases where HTTP POST operations are required, they don't do multi-part formposts. Multi-part formposts were introduced as a better way to post (possibly large) binary data and were first documented in the RFC1867 (updated in RFC2388). They're called multi-part because they're built by a chain of parts, each part being a single unit of data. Each part has its own name and contents. You can in fact create and post a multi-part formpost with the regular libcurl POST support described above, but that would require that you build a formpost yourself and provide to libcurl. To make that easier, libcurl provides *curl_formadd(3)*. Using this function, you add parts to the form. When you're done adding parts, you post the whole form.

The following example sets two simple text parts with plain textual contents, and then a file with binary contents and uploads the whole thing.

```
struct curl_httppost *post=NULL;
struct curl_httppost *last=NULL;
curl_formadd(&post, &last,
        CURLFORM_COPYNAME, "name",
        CURLFORM_COPYCONTENTS, "daniel", CURLFORM_END);
```

```
curl_formadd(&post, &last,
        CURLFORM_COPYNAME, "project",
        CURLFORM_COPYCONTENTS, "curl", CURLFORM_END);
curl_formadd(&post, &last,
        CURLFORM_COPYNAME, "logotype-image",
        CURLFORM_FILECONTENT, "curl.png", CURLFORM_END);

/* Set the form info */
curl_easy_setopt(easyhandle, CURLOPT_HTTPPOST, post);

curl_easy_perform(easyhandle); /* post away! */

/* free the post data again */
curl_formfree(post);
```

Multipart formposts are chains of parts using MIME-style separators and headers. It means that each one of these separate parts get a few headers set that describe the individual content-type, size etc. To enable your application to handicraft this formpost even more, libcurl allows you to supply your own set of custom headers to such an individual form part. You can of course supply headers to as many parts as you like, but this little example will show how you set headers to one specific part when you add that to the post handle:

```
struct curl_slist *headers=NULL;
headers = curl_slist_append(headers, "Content-Type: text/xml");

curl_formadd(&post, &last,
        CURLFORM_COPYNAME, "logotype-image",
        CURLFORM_FILECONTENT, "curl.xml",
        CURLFORM_CONTENTHEADER, headers,
        CURLFORM_END);

curl_easy_perform(easyhandle); /* post away! */

curl_formfree(post); /* free post */
curl_slist_free_all(headers); /* free custom header list */
```

Since all options on an easyhandle are "sticky", they remain the same until changed even if you do call *curl_easy_perform(3)*, you may need to tell curl to go back to a plain GET request if you intend to do one as your next request. You force an easyhandle to go back to GET by using the CURLOPT_HTTPGET option:

```
curl_easy_setopt(easyhandle, CURLOPT_HTTPGET, 1L);
```

Just setting CURLOPT_POSTFIELDS to "" or NULL will *not* stop libcurl from doing a POST. It will just make it POST without any data to send!

Showing Progress

For historical and traditional reasons, libcurl has a built-in progress meter that can be switched on and then makes it present a progress meter in your terminal.

Switch on the progress meter by, oddly enough, setting CURLOPT_NOPROGRESS to zero. This option is set to 1 by default.

For most applications however, the built-in progress meter is useless and what instead is interesting is the ability to specify a progress callback. The function pointer you pass to libcurl will then be called on

irregular intervals with information about the current transfer.

Set the progress callback by using CURLOPT_PROGRESSFUNCTION. And pass a pointer to a function that matches this prototype:

```
int progress_callback(void *clientp,
                      double dltotal,
                      double dlnow,
                      double ultotal,
                      double ulnow);
```

If any of the input arguments is unknown, a 0 will be passed. The first argument, the 'clientp' is the pointer you pass to libcurl with CURLOPT_PROGRESSDATA. libcurl won't touch it.

libcurl with C++

There's basically only one thing to keep in mind when using C++ instead of C when interfacing libcurl:

The callbacks CANNOT be non-static class member functions

Example C++ code:

```
class AClass {
  static size_t write_data(void *ptr, size_t size, size_t nmemb,
                           void *ourpointer)
  {
    /* do what you want with the data */
  }
}
```

Proxies

What "proxy" means according to Merriam-Webster: "a person authorized to act for another" but also "the agency, function, or office of a deputy who acts as a substitute for another".

Proxies are exceedingly common these days. Companies often only offer Internet access to employees through their proxies. Network clients or user-agents ask the proxy for documents, the proxy does the actual request and then it returns them.

libcurl supports SOCKS and HTTP proxies. When a given URL is wanted, libcurl will ask the proxy for it instead of trying to connect to the actual host identified in the URL.

If you're using a SOCKS proxy, you may find that libcurl doesn't quite support all operations through it.

For HTTP proxies: the fact that the proxy is a HTTP proxy puts certain restrictions on what can actually happen. A requested URL that might not be a HTTP URL will be still be passed to the HTTP proxy to deliver back to libcurl. This happens transparently, and an application may not need to know. I say "may", because at times it is very important to understand that all operations over a HTTP proxy use the HTTP protocol. For example, you can't invoke your own custom FTP commands or even proper FTP directory listings.

Proxy Options

To tell libcurl to use a proxy at a given port number:

```
curl_easy_setopt(easyhandle, CURLOPT_PROXY, "proxy-host.com:8080");
```

Some proxies require user authentication before allowing a request, and you pass that information similar to this:

```
curl_easy_setopt(easyhandle, CURLOPT_PROXYUSERPWD, "user:password");
```

If you want to, you can specify the host name only in the CURLOPT_PROXY option, and set the port number separately with CURLOPT_PROXYPORT.

Tell libcurl what kind of proxy it is with CURLOPT_PROXYTYPE (if not, it will default to assume a HTTP proxy):

```
curl_easy_setopt(easyhandle, CURLOPT_PROXYTYPE, CURLPROXY_SOCKS4);
```

Environment Variables

libcurl automatically checks and uses a set of environment variables to know what proxies to use for certain protocols. The names of the variables are following an ancient de facto standard and are built up as "[protocol]_proxy" (note the lower casing). Which makes the variable 'http_proxy' checked for a name of a proxy to use when the input URL is HTTP. Following the same rule, the variable named 'ftp_proxy' is checked for FTP URLs. Again, the proxies are always HTTP proxies, the different names of the variables simply allows different HTTP proxies to be used.

The proxy environment variable contents should be in the format "[protocol://][user:password@]machine[:port]". Where the protocol:// part is simply ignored if present (so http://proxy and bluerk://proxy will do the same) and the optional port number specifies on which port the proxy operates on the host. If not specified, the internal default port number will be used and that is most likely *not* the one you would like it to be.

There are two special environment variables. 'all_proxy' is what sets proxy for any URL in case the protocol specific variable wasn't set, and 'no_proxy' defines a list of hosts that should not use a proxy even though a variable may say so. If 'no_proxy' is a plain asterisk ("*") it matches all hosts.

To explicitly disable libcurl's checking for and using the proxy environment variables, set the proxy name to "" - an empty string - with CURLOPT_PROXY.

SSL and Proxies

SSL is for secure point-to-point connections. This involves strong encryption and similar things, which effectively makes it impossible for a proxy to operate as a "man in between" which the proxy's task is, as previously discussed. Instead, the only way to have SSL work over a HTTP proxy is to ask the proxy to tunnel trough everything without being able to check or fiddle with the traffic.

Opening an SSL connection over a HTTP proxy is therefor a matter of asking the proxy for a straight connection to the target host on a specified port. This is made with the HTTP request CONNECT. ("please mr proxy, connect me to that remote host").

Because of the nature of this operation, where the proxy has no idea what kind of data that is passed in and out through this tunnel, this breaks some of the very few advantages that come from using a proxy, such as caching. Many organizations prevent this kind of tunneling to other destination port numbers than 443 (which is the default HTTPS port number).

Tunneling Through Proxy

As explained above, tunneling is required for SSL to work and often even restricted to the operation intended for SSL; HTTPS.

This is however not the only time proxy-tunneling might offer benefits to you or your application.

As tunneling opens a direct connection from your application to the remote machine, it suddenly also re-introduces the ability to do non-HTTP operations over a HTTP proxy. You can in fact use things such as FTP upload or FTP custom commands this way.

Again, this is often prevented by the administrators of proxies and is rarely allowed.

Tell libcurl to use proxy tunneling like this:

```
curl_easy_setopt(easyhandle, CURLOPT_HTTPPROXYTUNNEL, 1L);
```

In fact, there might even be times when you want to do plain HTTP operations using a tunnel like this, as it then enables you to operate on the remote server instead of asking the proxy to do so. libcurl will not stand in the way for such innovative actions either!

Proxy Auto-Config

Netscape first came up with this. It is basically a web page (usually using a .pac extension) with a Javascript that when executed by the browser with the requested URL as input, returns information to the browser on how to connect to the URL. The returned information might be "DIRECT" (which means no proxy should be used), "PROXY host:port" (to tell the browser where the proxy for this particular URL is) or "SOCKS host:port" (to direct the browser to a SOCKS proxy).

libcurl has no means to interpret or evaluate Javascript and thus it doesn't support this. If you get yourself in a position where you face this nasty invention, the following advice have been mentioned and used in the past:

- Depending on the Javascript complexity, write up a script that translates it to another language and execute that.

- Read the Javascript code and rewrite the same logic in another language.

- Implement a Javascript interpreter; people have successfully used the Mozilla Javascript engine in the past.

- Ask your admins to stop this, for a static proxy setup or similar.

Persistence Is The Way to Happiness

Re-cycling the same easy handle several times when doing multiple requests is the way to go.

After each single *curl_easy_perform(3)* operation, libcurl will keep the connection alive and open. A subsequent request using the same easy handle to the same host might just be able to use the already open connection! This reduces network impact a lot.

Even if the connection is dropped, all connections involving SSL to the same host again, will benefit from libcurl's session ID cache that drastically reduces re-connection time.

FTP connections that are kept alive save a lot of time, as the command- response round-trips are skipped, and also you don't risk getting blocked without permission to login again like on many FTP servers only

allowing N persons to be logged in at the same time.

libcurl caches DNS name resolving results, to make lookups of a previously looked up name a lot faster.

Other interesting details that improve performance for subsequent requests may also be added in the future.

Each easy handle will attempt to keep the last few connections alive for a while in case they are to be used again. You can set the size of this "cache" with the CURLOPT_MAXCONNECTS option. Default is 5. There is very seldom any point in changing this value, and if you think of changing this it is often just a matter of thinking again.

To force your upcoming request to not use an already existing connection (it will even close one first if there happens to be one alive to the same host you're about to operate on), you can do that by setting CURLOPT_FRESH_CONNECT to 1. In a similar spirit, you can also forbid the upcoming request to be "lying" around and possibly get re-used after the request by setting CURLOPT_FORBID_REUSE to 1.

HTTP Headers Used by libcurl

When you use libcurl to do HTTP requests, it'll pass along a series of headers automatically. It might be good for you to know and understand these. You can replace or remove them by using the CURLOPT_HTTPHEADER option.

Host This header is required by HTTP 1.1 and even many 1.0 servers and should be the name of the server we want to talk to. This includes the port number if anything but default.

Accept "*/*".

Expect When doing POST requests, libcurl sets this header to "100-continue" to ask the server for an "OK" message before it proceeds with sending the data part of the post. If the POSTed data amount is deemed "small", libcurl will not use this header.

Customizing Operations

There is an ongoing development today where more and more protocols are built upon HTTP for transport. This has obvious benefits as HTTP is a tested and reliable protocol that is widely deployed and has excellent proxy-support.

When you use one of these protocols, and even when doing other kinds of programming you may need to change the traditional HTTP (or FTP or...) manners. You may need to change words, headers or various data.

libcurl is your friend here too.

CUSTOMREQUEST

If just changing the actual HTTP request keyword is what you want, like when GET, HEAD or POST is not good enough for you, CURLOPT_CUSTOMREQUEST is there for you. It is very simple to use:

```
curl_easy_setopt(easyhandle, CURLOPT_CUSTOMREQUEST, "MYOWNREQUEST");
```

When using the custom request, you change the request keyword of the actual request you are performing. Thus, by default you make a GET request but you can also make a POST operation (as described before) and then replace the POST keyword if you want to. You're the boss.

Modify Headers

HTTP-like protocols pass a series of headers to the server when doing the request, and you're free to pass any amount of extra headers that you think fit. Adding headers is this easy:

```
struct curl_slist *headers=NULL; /* init to NULL is important */

headers = curl_slist_append(headers, "Hey-server-hey: how are you?");
headers = curl_slist_append(headers, "X-silly-content: yes");

/* pass our list of custom made headers */
curl_easy_setopt(easyhandle, CURLOPT_HTTPHEADER, headers);

curl_easy_perform(easyhandle); /* transfer http */

curl_slist_free_all(headers); /* free the header list */
```

... and if you think some of the internally generated headers, such as Accept: or Host: don't contain the data you want them to contain, you can replace them by simply setting them too:

```
headers = curl_slist_append(headers, "Accept: Agent-007");
headers = curl_slist_append(headers, "Host: munged.host.line");
```

Delete Headers

If you replace an existing header with one with no contents, you will prevent the header from being sent. For instance, if you want to completely prevent the "Accept:" header from being sent, you can disable it with code similar to this:

```
headers = curl_slist_append(headers, "Accept:");
```

Both replacing and canceling internal headers should be done with careful consideration and you should be aware that you may violate the HTTP protocol when doing so.

Enforcing chunked transfer-encoding

By making sure a request uses the custom header "Transfer-Encoding: chunked" when doing a non-GET HTTP operation, libcurl will switch over to "chunked" upload, even though the size of the data to upload might be known. By default, libcurl usually switches over to chunked upload automatically if the upload data size is unknown.

HTTP Version

All HTTP requests includes the version number to tell the server which version we support. libcurl speaks HTTP 1.1 by default. Some very old servers don't like getting 1.1-requests and when dealing with stubborn old things like that, you can tell libcurl to use 1.0 instead by doing something like this:

```
curl_easy_setopt(easyhandle, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_0);
```

FTP Custom Commands

Not all protocols are HTTP-like, and thus the above may not help you when you want to make, for example, your FTP transfers to behave differently.

Sending custom commands to a FTP server means that you need to send the commands exactly as the FTP server expects them (RFC959 is a good guide here), and you can only use commands that work on the control-connection alone. All kinds of commands that require data interchange and thus need a data-connection must be left to libcurl's own judgement. Also be aware that libcurl will do its very best to change directory to the target directory before doing any transfer, so if you change directory (with CWD or similar) you might confuse libcurl and then it might not attempt to transfer the file in the correct remote directory.

A little example that deletes a given file before an operation:

```
headers = curl_slist_append(headers, "DELE file-to-remove");

/* pass the list of custom commands to the handle */
curl_easy_setopt(easyhandle, CURLOPT_QUOTE, headers);

curl_easy_perform(easyhandle); /* transfer ftp data! */

curl_slist_free_all(headers); /* free the header list */
```

If you would instead want this operation (or chain of operations) to happen _after_ the data transfer took place the option to *curl_easy_setopt(3)* would instead be called CURLOPT_POSTQUOTE and used the exact same way.

The custom FTP command will be issued to the server in the same order they are added to the list, and if a command gets an error code returned back from the server, no more commands will be issued and libcurl will bail out with an error code (CURLE_QUOTE_ERROR). Note that if you use CURLOPT_QUOTE to send commands before a transfer, no transfer will actually take place when a quote command has failed.

If you set the CURLOPT_HEADER to 1, you will tell libcurl to get information about the target file and output "headers" about it. The headers will be in "HTTP-style", looking like they do in HTTP.

The option to enable headers or to run custom FTP commands may be useful to combine with CURLOPT_NOBODY. If this option is set, no actual file content transfer will be performed.

FTP Custom CUSTOMREQUEST

If you do want to list the contents of a FTP directory using your own defined FTP command, CURLOPT_CUSTOMREQUEST will do just that. "NLST" is the default one for listing directories but you're free to pass in your idea of a good alternative.

Cookies Without Chocolate Chips

In the HTTP sense, a cookie is a name with an associated value. A server sends the name and value to the client, and expects it to get sent back on every subsequent request to the server that matches the particular conditions set. The conditions include that the domain name and path match and that the cookie hasn't become too old.

In real-world cases, servers send new cookies to replace existing ones to update them. Server use cookies to "track" users and to keep "sessions".

Cookies are sent from server to clients with the header Set-Cookie: and they're sent from clients to servers with the Cookie: header.

To just send whatever cookie you want to a server, you can use CURLOPT_COOKIE to set a cookie string

like this:

```
curl_easy_setopt(easyhandle, CURLOPT_COOKIE, "name1=var1; name2=var2;");
```

In many cases, that is not enough. You might want to dynamically save whatever cookies the remote server passes to you, and make sure those cookies are then used accordingly on later requests.

One way to do this, is to save all headers you receive in a plain file and when you make a request, you tell libcurl to read the previous headers to figure out which cookies to use. Set the header file to read cookies from with CURLOPT_COOKIEFILE.

The CURLOPT_COOKIEFILE option also automatically enables the cookie parser in libcurl. Until the cookie parser is enabled, libcurl will not parse or understand incoming cookies and they will just be ignored. However, when the parser is enabled the cookies will be understood and the cookies will be kept in memory and used properly in subsequent requests when the same handle is used. Many times this is enough, and you may not have to save the cookies to disk at all. Note that the file you specify to CURLOPT_COOKIEFILE doesn't have to exist to enable the parser, so a common way to just enable the parser and not read any cookies is to use the name of a file you know doesn't exist.

If you would rather use existing cookies that you've previously received with your Netscape or Mozilla browsers, you can make libcurl use that cookie file as input. The CURLOPT_COOKIEFILE is used for that too, as libcurl will automatically find out what kind of file it is and act accordingly.

Perhaps the most advanced cookie operation libcurl offers, is saving the entire internal cookie state back into a Netscape/Mozilla formatted cookie file. We call that the cookie-jar. When you set a file name with CURLOPT_COOKIEJAR, that file name will be created and all received cookies will be stored in it when *curl_easy_cleanup(3)* is called. This enables cookies to get passed on properly between multiple handles without any information getting lost.

FTP Peculiarities We Need

FTP transfers use a second TCP/IP connection for the data transfer. This is usually a fact you can forget and ignore but at times this fact will come back to haunt you. libcurl offers several different ways to customize how the second connection is being made.

libcurl can either connect to the server a second time or tell the server to connect back to it. The first option is the default and it is also what works best for all the people behind firewalls, NATs or IP-masquerading setups. libcurl then tells the server to open up a new port and wait for a second connection. This is by default attempted with EPSV first, and if that doesn't work it tries PASV instead. (EPSV is an extension to the original FTP spec and does not exist nor work on all FTP servers.)

You can prevent libcurl from first trying the EPSV command by setting CURLOPT_FTP_USE_EPSV to zero.

In some cases, you will prefer to have the server connect back to you for the second connection. This might be when the server is perhaps behind a firewall or something and only allows connections on a single port. libcurl then informs the remote server which IP address and port number to connect to. This is made with the CURLOPT_FTPPORT option. If you set it to "-", libcurl will use your system's "default IP address". If you want to use a particular IP, you can set the full IP address, a host name to resolve to an IP address or even a local network interface name that libcurl will get the IP address from.

When doing the "PORT" approach, libcurl will attempt to use the EPRT and the LPRT before trying PORT, as they work with more protocols. You can disable this behavior by setting CURLOPT_FTP_USE_EPRT to zero.

Headers Equal Fun

Some protocols provide "headers", meta-data separated from the normal data. These headers are by default not included in the normal data stream, but you can make them appear in the data stream by setting CURLOPT_HEADER to 1.

What might be even more useful, is libcurl's ability to separate the headers from the data and thus make the callbacks differ. You can for example set a different pointer to pass to the ordinary write callback by setting CURLOPT_WRITEHEADER.

Or, you can set an entirely separate function to receive the headers, by using CURLOPT_HEADERFUNCTION.

The headers are passed to the callback function one by one, and you can depend on that fact. It makes it easier for you to add custom header parsers etc.

"Headers" for FTP transfers equal all the FTP server responses. They aren't actually true headers, but in this case we pretend they are! ;-)

Post Transfer Information

[curl_easy_getinfo]

Security Considerations

The libcurl project takes security seriously. The library is written with caution and precautions are taken to mitigate many kinds of risks encountered while operating with potentially malicious servers on the Internet. It is a powerful library, however, which allows application writers to make trade offs between ease of writing and exposure to potential risky operations. If used the right way, you can use libcurl to transfer data pretty safely.

Many applications are used in closed networks where users and servers can be trusted, but many others are used on arbitrary servers and are fed input from potentially untrusted users. Following is a discussion about some risks in the ways in which applications commonly use libcurl and potential mitigations of those risks. It is by no means comprehensive, but shows classes of attacks that robust applications should consider. The Common Weakness Enumeration project at http://cwe.mitre.org/ is a good reference for many of these and similar types of weaknesses of which application writers should be aware.

Command Lines

If you use a command line tool (such as curl) that uses libcurl, and you give options to the tool on the command line those options can very likely get read by other users of your system when they use 'ps' or other tools to list currently running processes.

To avoid this problem, never feed sensitive things to programs using command line options. Write them to a protected file and use the −K option to avoid this.

.netrc

.netrc is a pretty handy file/feature that allows you to login quickly and automatically to frequently visited sites. The file contains passwords in clear text and is a real security risk. In some cases, your .netrc is also stored in a home directory that is NFS mounted or used on another network based file system, so the clear text password will fly through your network every time anyone reads that file!

To avoid this problem, don't use .netrc files and never store passwords in plain text anywhere.

Clear Text Passwords

Many of the protocols libcurl supports send name and password unencrypted as clear text (HTTP Basic authentication, FTP, TELNET etc). It is very easy for anyone on your network or a network nearby yours to just fire up a network analyzer tool and eavesdrop on your passwords. Don't let the fact that HTTP Basic uses base64 encoded passwords fool you. They may not look readable at a first glance, but they very easily "deciphered" by anyone within seconds.

To avoid this problem, use HTTP authentication methods or other protocols that don't let snoopers see your password: HTTP with Digest, NTLM or GSS authentication, HTTPS, FTPS, SCP, SFTP and FTP-Kerberos are a few examples.

Redirects

The CURLOPT_FOLLOWLOCATION option automatically follows HTTP redirects sent by a remote server. These redirects can refer to any kind of URL, not just HTTP. A redirect to a file: URL would cause the libcurl to read (or write) arbitrary files from the local filesystem. If the application returns the data back to the user (as would happen in some kinds of CGI scripts), an attacker could leverage this to read otherwise forbidden data (e.g. file://localhost/etc/passwd).

If authentication credentials are stored in the ~/.netrc file, or Kerberos is in use, any other URL type (not just file:) that requires authentication is also at risk. A redirect such as ftp://some-internal-server/private-file would then return data even when the server is password protected.

In the same way, if an unencrypted SSH private key has been configured for the user running the libcurl application, SCP: or SFTP: URLs could access password or private-key protected resources, e.g. sftp://user@some-internal-server/etc/passwd

The CURLOPT_REDIR_PROTOCOLS and CURLOPT_NETRC options can be used to mitigate against this kind of attack.

A redirect can also specify a location available only on the machine running libcurl, including servers hidden behind a firewall from the attacker. e.g. http://127.0.0.1/ or http://intranet/delete-stuff.cgi?delete=all or tftp://bootp-server/pc-config-data

Apps can mitigate against this by disabling CURLOPT_FOLLOWLOCATION and handling redirects itself, sanitizing URLs as necessary. Alternately, an app could leave CURLOPT_FOLLOWLOCATION enabled but set CURLOPT_REDIR_PROTOCOLS and install a CURLOPT_OPENSOCKETFUNCTION callback function in which addresses are sanitized before use.

Private Resources

A user who can control the DNS server of a domain being passed in within a URL can change the address of the host to a local, private address which a server-side libcurl-using application could then use. e.g. the innocuous URL http://fuzzybunnies.example.com/ could actually resolve to the IP address of a server behind a firewall, such as 127.0.0.1 or 10.1.2.3. Apps can mitigate against this by setting a CURLOPT_OPENSOCKETFUNCTION and checking the address before a connection.

All the malicious scenarios regarding redirected URLs apply just as well to non-redirected URLs, if the user is allowed to specify an arbitrary URL that could point to a private resource. For example, a web app providing a translation service might happily translate file://localhost/etc/passwd and display the result. Apps can mitigate against this with the CURLOPT_PROTOCOLS option as well as by similar mitigation techniques for redirections.

A malicious FTP server could in response to the PASV command return an IP address and port number for a server local to the app running libcurl but behind a firewall. Apps can mitigate

against this by using the CURLOPT_FTP_SKIP_PASV_IP option or CURLOPT_FTPPORT.

IPv6 Addresses

libcurl will normally handle IPv6 addresses transparently and just as easily as IPv4 addresses. That means that a sanitizing function that filters out addressses like 127.0.0.1 isn't sufficient--the equivalent IPv6 addresses ::1, ::, 0:00::0:1, ::127.0.0.1 and ::ffff:7f00:1 supplied somehow by an attacker would all bypass a naive filter and could allow access to undesired local resources. IPv6 also has special address blocks like link-local and site-local that generally shouldn't be accessed by a server-side libcurl-using application. A poorly-configured firewall installed in a data center, organization or server may also be configured to limit IPv4 connections but leave IPv6 connections wide open. In some cases, the CURL_IPRESOLVE_V4 option can be used to limit resolved addresses to IPv4 only and bypass these issues.

Uploads

When uploading, a redirect can cause a local (or remote) file to be overwritten. Apps must not allow any unsanitized URL to be passed in for uploads. Also, CURLOPT_FOLLOWLOCATION should not be used on uploads. Instead, the app should handle redirects itself, sanitizing each URL first.

Authentication

Use of CURLOPT_UNRESTRICTED_AUTH could cause authentication information to be sent to an unknown second server. Apps can mitigate against this by disabling CURLOPT_FOLLOWLOCATION and handling redirects itself, sanitizing where necessary.

Use of the CURLAUTH_ANY option to CURLOPT_HTTPAUTH could result in user name and password being sent in clear text to an HTTP server. Instead, use CURLAUTH_ANYSAFE which ensures that the password is encrypted over the network, or else fail the request.

Use of the CURLUSESSL_TRY option to CURLOPT_USE_SSL could result in user name and password being sent in clear text to an FTP server. Instead, use CURLUSESSL_CONTROL to ensure that an encrypted connection is used or else fail the request.

Cookies

If cookies are enabled and cached, then a user could craft a URL which performs some malicious action to a site whose authentication is already stored in a cookie. e.g. http://mail.example.com/delete-stuff.cgi?delete=all Apps can mitigate against this by disabling cookies or clearing them between requests.

Dangerous URLs

SCP URLs can contain raw commands within the scp: URL, which is a side effect of how the SCP protocol is designed. e.g. scp://user:pass@host/a;date >/tmp/test; Apps must not allow unsanitized SCP: URLs to be passed in for downloads.

Denial of Service

A malicious server could cause libcurl to effectively hang by sending a trickle of data through, or even no data at all but just keeping the TCP connection open. This could result in a denial-of-service attack. The CURLOPT_TIMEOUT and/or CURLOPT_LOW_SPEED_LIMIT options can be used to mitigate against this.

A malicious server could cause libcurl to effectively hang by starting to send data, then severing the connection without cleanly closing the TCP connection. The app could install a CURLOPT_SOCKOPTFUNCTION callback function and set the TCP SO_KEEPALIVE option to

mitigate against this. Setting one of the timeout options would also work against this attack.

A malicious server could cause libcurl to download an infinite amount of data, potentially causing all of memory or disk to be filled. Setting the CURLOPT_MAXFILESIZE_LARGE option is not sufficient to guard against this. Instead, the app should monitor the amount of data received within the write or progress callback and abort once the limit is reached.

A malicious HTTP server could cause an infinite redirection loop, causing a denial-of-service. This can be mitigated by using the CURLOPT_MAXREDIRS option.

Arbitrary Headers

User-supplied data must be sanitized when used in options like CURLOPT_USERAGENT, CURLOPT_HTTPHEADER, CURLOPT_POSTFIELDS and others that are used to generate structured data. Characters like embedded carriage returns or ampersands could allow the user to create additional headers or fields that could cause malicious transactions.

Server-supplied Names

A server can supply data which the application may, in some cases, use as a file name. The curl command-line tool does this with --remote-header-name, using the Content-disposition: header to generate a file name. An application could also use CURLINFO_EFFECTIVE_URL to generate a file name from a server-supplied redirect URL. Special care must be taken to sanitize such names to avoid the possibility of a malicious server supplying one like "/etc/passwd", "\autoexec.bat", "prn:" or even ".bashrc".

Server Certificates

A secure application should never use the CURLOPT_SSL_VERIFYPEER option to disable certificate validation. There are numerous attacks that are enabled by apps that fail to properly validate server TLS/SSL certificates, thus enabling a malicious server to spoof a legitimate one. HTTPS without validated certificates is potentially as insecure as a plain HTTP connection.

Showing What You Do

On a related issue, be aware that even in situations like when you have problems with libcurl and ask someone for help, everything you reveal in order to get best possible help might also impose certain security related risks. Host names, user names, paths, operating system specifics, etc. (not to mention passwords of course) may in fact be used by intruders to gain additional information of a potential target.

Be sure to limit access to application logs if they could hold private or security-related data. Besides the obvious candidates like user names and passwords, things like URLs, cookies or even file names could also hold sensitive data.

To avoid this problem, you must of course use your common sense. Often, you can just edit out the sensitive data or just search/replace your true information with faked data.

Multiple Transfers Using the multi Interface

The easy interface as described in detail in this document is a synchronous interface that transfers one file at a time and doesn't return until it is done.

The multi interface, on the other hand, allows your program to transfer multiple files in both directions at the same time, without forcing you to use multiple threads. The name might make it seem that the multi interface is for multi-threaded programs, but the truth is almost the reverse. The multi interface can allow a single-threaded application to perform the same kinds of multiple, simultaneous transfers that multi-

threaded programs can perform. It allows many of the benefits of multi-threaded transfers without the complexity of managing and synchronizing many threads.

To use this interface, you are better off if you first understand the basics of how to use the easy interface. The multi interface is simply a way to make multiple transfers at the same time by adding up multiple easy handles into a "multi stack".

You create the easy handles you want and you set all the options just like you have been told above, and then you create a multi handle with *curl_multi_init(3)* and add all those easy handles to that multi handle with *curl_multi_add_handle(3)*.

When you've added the handles you have for the moment (you can still add new ones at any time), you start the transfers by calling *curl_multi_perform(3)*.

curl_multi_perform(3) is asynchronous. It will only execute as little as possible and then return back control to your program. It is designed to never block.

The best usage of this interface is when you do a select() on all possible file descriptors or sockets to know when to call libcurl again. This also makes it easy for you to wait and respond to actions on your own application's sockets/handles. You figure out what to select() for by using *curl_multi_fdset(3)*, that fills in a set of fd_set variables for you with the particular file descriptors libcurl uses for the moment.

When you then call select(), it'll return when one of the file handles signal action and you then call *curl_multi_perform(3)* to allow libcurl to do what it wants to do. Take note that libcurl does also feature some time-out code so we advise you to never use very long timeouts on select() before you call *curl_multi_perform(3)*, which thus should be called unconditionally every now and then even if none of its file descriptors have signaled ready. Another precaution you should use: always call *curl_multi_fdset(3)* immediately before the select() call since the current set of file descriptors may change when calling a curl function.

If you want to stop the transfer of one of the easy handles in the stack, you can use *curl_multi_remove_handle(3)* to remove individual easy handles. Remember that easy handles should be *curl_easy_cleanup(3)*ed.

When a transfer within the multi stack has finished, the counter of running transfers (as filled in by *curl_multi_perform(3)*) will decrease. When the number reaches zero, all transfers are done.

curl_multi_info_read(3) can be used to get information about completed transfers. It then returns the CURLcode for each easy transfer, to allow you to figure out success on each individual transfer.

SSL, Certificates and Other Tricks

[seeding, passwords, keys, certificates, ENGINE, ca certs]

Sharing Data Between Easy Handles

You can share some data between easy handles when the easy interface is used, and some data is share automatically when you use the multi interface.

When you add easy handles to a multi handle, these easy handles will automatically share a lot of the data that otherwise would be kept on a per-easy handle basis when the easy interface is used.

The DNS cache is shared between handles within a multi handle, making subsequent name resolving faster, and the connection pool that is kept to better allow persistent connections and connection re-use is also shared. If you're using the easy interface, you can still share these between specific easy handles by using the share interface, see *libcurl-share(3)*.

Some things are never shared automatically, not within multi handles, like for example cookies so the only way to share that is with the share interface.

Footnotes

[1] libcurl 7.10.3 and later have the ability to switch over to chunked Transfer-Encoding in cases where HTTP uploads are done with data of an unknown size.

[2] This happens on Windows machines when libcurl is built and used as a DLL. However, you can still do this on Windows if you link with a static library.

[3] The curl-config tool is generated at build-time (on UNIX-like systems) and should be installed with the 'make install' or similar instruction that installs the library, header files, man pages etc.

[4] This behavior was different in versions before 7.17.0, where strings had to remain valid past the end of the *curl_easy_setopt(3)* call.