hURL file transfer library

1.0.0

Reference Manual

Jennifer Bi, Maria Javier, Aryeh Zapinsky

Table of Contents

Introduction

Networking allows for computers to communicate with one another and most importantly share data. Providing a reliable, efficient, and safe mechanism for network programming across different languages allows for users to build applications that not only provide some sort of utility but also contribute towards a greater sense of connectivity in a world that thrives on data accessibility.

hURL file transfer library is a reimplementation of curl/cURL, the command line tool and the library written in C++ using the Asio library to provide a networking interface. Asio provides portability across different operating systems while also maintaining the efficiency of asynchronous I/O. Asio is also a widely used networking library so this project will specifically be more useful for businesses that already use the framework.

This manual will introduce all the methods required in order to retrieve information from the internet that is equivalent to the curl command but extended to include options depending on user preferences.

A basic overview of the Asio interface will be introduced in order to provide context of how the tcp connections are being made but is by no means representative of the full capability of the library.

The manual is intended for programmers that are familiar with C++ and networking basics. You can find resources below in order to learn more about both Asio and libcurl.

Resources Sites

This manual is intended to give a comprehensive overview of hURL's user interface. For a more detailed guide on asynchronous programming and networking, please see:

https://think-async.com/Asio/Documentation contains an exhaustive manual for Asio networking library, which is used to implement this library, and may also be used in conjunction with features in this library.

https://ec.haxx.se/libcurl--libcurl.html contains documentation on libcurl, the library that extends the curl command, and the library which hURL is based on.

Source Code

Dependencies

This library requires Asio, OpenSSL, and cmake libraries to be already installed in your operating system.

https://think-async.com/Asio/Download contains the latest version of Asio that was used for the implementation of this project.

https://github.com/openssl/openssl contains the source code that can be downloaded manually.

http://www.libressl.org/ Finally, libressl is another library that may be installed which allows you to use ssl features in hURL. It is an extension of OpenSSL that also comes with libtls, which is used for ssl transfers.

> An important note about OpenSSL and LibreSSL: If you do not have the latest version of OpenSSL and/or LibreSSL, it is likely your ssl transfers (e.g., using our sslclient) will fail. This is especially true if you are running on MacOS, which is no longer supported by OpenSSL. In using an old version, building hURL will cause warnings about deprecated X509 hash functions (which is never a good sign in life).

Latest Version

The source code can be retrieved by accessing the github repo.

```
git clone https://github.com/jenniferbi/cpp-libcurl.git
```

In order to compile the source code, make a directory called build at the root of the repository. Move into that build directory and then continue with the following:

```
cmake ..
make
```

This will initialize cmake for the project followed by compile all library files and examples. The example test script executables will be placed in build/bin after the execution of the previous commands.

Development

For developers interested in contributing to hURL.

Code Layout

examples/

Contains several test examples that exhibit the functionality of the curl handle. Simple performs a standard curl request while threaded aims to provide asynchronous requests results using Asio's multithreaded library extensions.

lib/include

Contains all header files that define data structures and functions used to make http connections, or http connections over SSL/TLS. This includes the handle class that implements the handle interface introduced by libcurl. For more explanation on the handle interface, and available options please see the next sections.

lib/src

Contains the HTTP connection implementation using Asio.

Networking

Asio Library Basics

This section provides a basic understanding of how Asio implements fundamental networking HTTP/HTTPS protocol by going through parts of an example provided by Asio Boost libraries. A template for many of the functions needed is provided here but for full implementation details, refer to the source code: Boost_Asio_Http/Client

Asio provides an object, io_service, that serves as a liaison between the socket, the operating system, the web and the program running. Every application that plans to use Asio specifically for networking will need an io_service object declared as the following.

```
asio::io_service& io_service;
```

There are many ways to conduct a simple HTTP request using Asio. hURL library specifically uses Asio's endpoint template class in order to support tcp protocol. An endpoint in Asio is a class template that is structured to support different protocols such as datagram, raw, sequential packets or even just a stream. With the endpoint class, Asio is able to create a sockaddr object associated with our io_service object in order to perform the data transfer.

In order to send the request, we first need to set up not only our socket attributes but also acquire the necessary endpoints associated with the server name that is provided. This is exactly what a resolver does in Asio. An example below is shown for an http request.

```
tcp::resolver resolver(io_service);
tcp::resolver::query query(<server_name>, "http");
tcp::resolver::iterator endpoint_iterator = resolver.resolve(query);
```

Now we are ready to create our socket and connect.

```
ip::tcp::socket socket(io_service);
asio::connect(socket, resolver.resolve(query));
```

This is an example of a synchronous connection. The resolver will attempt to find the right endpoint associated with the right port number needed to connect the socket to its rightful destination. In the hURL library, an asynchronous connection was implemented in which the resolver tries to test each endpoint until one matches which allows for minimal error checking.

```
asio::async_connect(socket, endpoint_iterator, this,asio::error_code& err);
```

After we make a successful connection, we can send the requests. For the purpose of this example, the user can configure their own path, url, port, and protocol. hURL library contains options that allows the user to specify these options which are saved in a struct and each option is passed to an http handle that makes the request.

A request can be formulated in two ways: ostream provided by the standard c++ library or a stream buffer provided by the Asio library.

```
asio::streambuf request;
std::ostream request_stream(&request)
request_stream << "GET" << <path> << "HTTP/1.0\r\n";
request_stream << "HOST" << <server> << "r\n";
request_stream << "Accept:" */* \r\n";
request_stream << "Connection: close\r\n\r\n";
```

Now we need to write the request and perform error checking on the response. The error handle provided by Asio will hold any extraneous errors that are usually a mistake on behalf of the user for example, a response that wasn't formatted correctly. If the connection was not made successfully, the socket needs to be closed and reconnected incrementing the endpoint iterator in order to see if another endpoint can be used. Assuming the connection is successful, we read the response and check for a 200 OK status. As a reminder, Asio provides options to do this either synchronously or asynchronously.

```
asio::async_write(socket, request, asio::error_code& err, size_t byes);
<...error checking...>
asio::async_read_until(socket,response,"\r\n",asio::error_code& err, size_t
bytes);
Std::istream response_stream(response);
response_stream >> <http_version>;
response_stream >> <status_code>;
<...check for 200 OK and HTTP/ tag>
```

Finally, we can skip the headers by reading them in through a stream until we get the actual information that is needed (not shown here), output this information into standard out and close the socket.

```
Std::istream response_stream(response)
asio::async_read(socket, response, asio::transfer_at_least(1),
asio::error_code& err, size_t bytes)
std::cout << response;
<HTTP TRANSFER SUCCESS :D >
```

It is important to remember to check for errors especially by utilizing the Asio error handler which provides checks for standard errors like a closed socket or End of File reached. This same test can be implemented again using multithreading which Asio supports and it is how hURL performs multiple requests asynchronously.

hURL Basics

`Curl::Easy`

In order to begin using the hURL library, the user must create an easy handle. The nomenclature is taken from libcurl so those familiar with the libcurl library will also find this implementation easy to use and understand. The handle does not need to be initiated, merely declared. The constructor for the handle will take care of instantiation and create a timer that allows for blocking within a certain time frame.

```
Easy myhandle;
```

Once the handle is created, the user has the opportunity to set options for the transfer including specifying a url, a path, a maximum number of bytes to retrieve, etc.

```
myhandle.setOpt(CURLPP_OPT_URL,”www.example.com/”,
CURLPP_OPT_TIMEOUT, 1000);
```

The setOpt method is implemented as a variadic function. Users can add as many options as needed. The url option is the only option that is required. Options also need to be set before performing the HTTP request. Once the request is made a perform() call can be made and the output will appear in standard out unless the option to write to a file is set.

```
myhandle.perform();
```

Curl::Multi
The multi interfaces shares similar functionality to the easy interface. If the user wishes to perform a request using the Multi handle, the only requirements are declaring the handle, specifying the minimum option of a url, and calling perform on that handle.

```
Multi myhandle;
myhandle.setOpt(CURLPP_OPT_URL, "www.cs.columbia.edu/");
myhandle.perform();
```

Curl::setOpt

setOpt Option	Set Value
CURLPP_OPT_URL	<string>(required)specifies url for request Sets CURL_OPT_SCHEME to the given protocol, e.g. HTTP or HTTPS
CURLPP_OPT_HOST	<string> specify network hostname
CURLPP_OPT_PATH	<string> specify path after url
CURLPP_OPT_PORT	<long> specify port number

CURLPP_OPT_VERSION	<float> transfer protocol, i.e 1.0/1.1
CURLPP_OPT_MAXFILE	<int> maximum file size to access in bytes
CURLPP_OPT_TIMEOUT	<int> specify time to wait for request (blocking), at limit socket will be disconnected
CURLPP_OPT_SSLCERT	<string> specify SSL client certificate
CURLPP_OPT_WRITEFN	<string> specify callback function before a socket write, used to process received data
CURLPP_OPT_READFN	<string> specify callback function before a socket read happens, used for HTTP POST
CURLPP_OPT_SCHEME	Takes on the value of either CURL_OPT_HTTP or CURL_OPT_HTTPS, explicitly set HTTP/HTTPS protocol
CURLPP_OPT_HTTP	<string> use HTTP protocol
CURLPP_OPT_HTTPS	<string> use HTTP protocol

Tutorials

The following tutorials exhibit the functionality of hURL library as a C++ tool that build upon the cURL and libcurl framework.

Performing a simple transfer

Here we formally walk the user through the steps of using this library to make an HTTP/HTTPS request.

We need to include several files in order to use hURL's library.

```
#include <iostream>
#include "handle.h"
#include "urldata.h"
```

urldata contains the data structures needed to handle the HTTP/HTTPS protocol through an http handle. It also contains a user defined struct that would populate the user defined options. We then declare our easy handle which can also be replaced by a multi handle.

```
int main(void)
{
     Easy myhandle;

    // Equivalent to CURL *curl;
    // CURLcode res;
    // curl = curl_easy_init();
```

In this tutorial, we will just set the URL by specifying the option flag and URL.

```
     myhandle.setOpt(CURLPP_OPT_URL,
                     "http://www.stroustrup.com/Bjarne2018.jpg");
```

Now that we have specified our options, we can call perform on our handle. Without a writeback function, the result of perform will be printed to std::out. You can redirect the output to a file and then open it. If the request is unsuccessful an error code will appear in std::err.

```
     myhandle.perform();
     return 0;
}
```

Congratulations on your first HTTP client implementation using hURL file transfer library!

Performing a https request

In order to perform an HTTPS request, we just have to change the options we set in our setOpt function. Firstly, we can can set several options including specifying the hostname, a path, a scheme, a timer, and a ssl certificate.

```
     myhandle.setOpt(CURLPP_OPT_HOST, "images.metmuseum.org",
                CURLPP_OPT_PATH, "CRDImages/as/original/DP141263",
                CURLPP_OPT_SCHEME, CURLPP_OPT_HTTPS,
                CURLPP_OPT_SSLCERT, "ca.pem"
);  // This will default to a working certificate, if you have one.
```

Or more simply, we can just pass in a URL with the scheme prepended to the beginning, like so:

```
     myhandle.setOpt(CURLPP_OPT_URL,
"https://images.metmuseum.org/CRDImages/dp/web-large/DP875552.jpg");
```

That’s all it takes.

Performing a transfer with a write callback

First, we will need to define a function with the prototype `int f(const unsigned char * s, size_t size)` . An easy way to do this is with the auto keyword and a lambda--the return type is inferred from the single return statement:

```
auto foo = [](const unsigned char * s, size_t size)
        {
            cout << "**************************************\n";
            cout << size << " received\n";
            cout << "**************************************\n";
            return 0;
        }
```

Next, we simply set the option to use foo as our write callback, and specify other options as usual:

```
 Easy myhandle;
    myhandle.setOpt(CURLPP_OPT_HOST, "www.columbia.edu",
      CURLPP_OPT_PATH, "/cu/bulletin/uwb/sel/COMS_Fall2018.html",
      CURLPP_OPT_WRITEFN, foo);

    myhandle.perform();
    return 0;
```

This is quite a trivial write callback. In a common case, we would use the write callback to buffer and/or process data.