hURL Library Design Document

Maria Javier, Jennifer Bi, Aryeh Zapinsky

April 2018

hURL is a file transfer library supporting HTTP protocol, inspired by the cURL command line tool and libcurl library, implemented with Asio 1.12.1 (non-Boost).

In this document, we discuss motivation for the library, intended usage of its features, and broader design principles. We share our experience from using the libcurl and Asio interfaces and testing programs, which has largely informed our decisions.

`libcurl` Overview

The starting point for design was libcurl. We broadly follow libcurl's usage pattern, to make hURL easy to use for a large userbase already familiar with libcurl.

Tutorials and documentation for libcurl can be found at `https://curl.haxx.se/libcurl/`. We assume basic knowledge of libcurl.

1. create handle
2. set options (configurability is important)
3. perform transfer
4. extract info upon completion (via callback functions)
5. repeat (easy handles are designed to be reused)

This user interface also captures a few of libcurl's design principles of configurability, easy reuse, and use of callbacks.

Data Structures and Classes

Libcurl offers easy and multi handle interfaces. Easy handles are used for synchronous transfers, multi handles are used for asynchronous parallel transfers in a single thread. Operations on handles happen through functions in the format of `curl_easy_*` and `curl_multi_*`. Multi-handles hold a linked list of easy handles.

Our approach is slightly different–naturally, it's object oriented. Easy and Multi are concrete classes inheriting from the abstract class Curl. The Easy wraps around an httphand or sslhand, for http or https transfer. Each Multi holds a vector of Easys, or for a quick transfer without the desire to reuse handles, a vector of httphands. In Multi, we perform multi-threaded asynchronous transfers by default. The reason for setting multi-threaded as default is that Asio library lends itself to it–we discuss this more in the Threading section.

Given the above, we can now say that in practice, a transfer looks like:

```
Easy myhandle;

myhandle.setOpt(CURLPP_OPT_URL, "images.metmuseum.org",
CURLPP_OPT_PATH, "/CRDImages/as/original/DP141263.jpg",
CURLPP_OPT_TIMEOUT, 1000);

myhandle.perform();
```

The easy handle performs an asynchronous transfer. We set minimal options, a URL and filepath. After performing the transfer, we get data in the default way–without a callback–by simply printing to standard output.

Setting options

Setting options is easier and more type-safe. This is an improvement upon libcurl's setopt function, which uses the not-type-safe `va_arg` macro. We use variadic templates, a feature that is only possible in C++11 and beyond[1]:

[1] see Eli Bendersky's article on variadic templates in C++11 for an excellent explanation of finer design points

```
template<typename T>
void Curl::setOpt(int a, T b) {
    Curl::_setopt(a, b);
}

template<typename T, typename... Args>
void Curl::setOpt(int a, T b, Args... args) {
    Curl::_setopt(a, b);
    Curl::setOpt(args...);
}
```

Using `Asio`

Our user interface is informed by libcurl design principles, while our implementation somewhat conforms to Asio design principles. Asio design rationale, as given by the reference manual, emphasizes portability, scalability, ease of use. Particularly important to us was the following two principles:

> **Efficiency.** The library should support techniques such as scatter-gather I/O, and allow programs to minimise data copying.
>
> **Basis for further abstraction.** The library should permit the development of other libraries that provide higher levels of abstraction. For example, implementations of commonly used protocols such as HTTP.

Threading

A main consequence of using Asio is the ability to provide thread safety guarantees. We hide implementation details of threading from the user without sacrificing flexibility. This improves upon libcurl's guarantees, specified in libcurl documentation:

> libcurl is thread safe but has no internal thread synchronization.
>
> **Handles.** You must never share the same handle in multiple threads. 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.
>
> **Shared Objects.** You can share certain data between multiple handles by using the share interface but you must provide your own locking.

Our thread safety guarantees rely on Asio's thread safety guarantees as well as C++ `shared_ptr` guarantees. Shared data structures may be accessed by different handles in different threads without additional locking.

Expanding the project

We have designed the class hierarchies to be expanded upon. We hope to add more options, for example more memory configuration such as scatter-gather I/O buffering. We hope to provide more levels of threading–from single thread synchronous to multi-thread asynchronous, and everything in between! (The protocol abstract class is designed as the interface to allow for different implementations). Variations in threading include single threaded parallel transfers using coroutines, offloading long read operations to a separate thread using `asio::post` and `asio::dispatch` message passing.

Acknowledgements

References and More Documentation

libcurl documentation, `https://curl.haxx.se/libcurl/`

Asio 1.11.0 documentation, `http://think-async.com/Asio/asio-1.11.0/doc/`

Bendersky, Eli, *Variadic templates in C++*, `https://eli.thegreenplace.net/2014/variadic-templates-in-c/`