Document Number: D3389=12-0079
Date: 2012-04-23
Reply to: Christopher Kohlhoff <chris@kohlhoff.com>

Urdl: a simple library for accessing web content

This paper introduces the Urdl C++ library. Urdl, first developed in 2009, builds on Boost.Asio to provide a simple abstraction for accessing and downloading web content. Possible uses for Urdl include:

- Downloading application configuration or data files.
- Downloading software updates.
- Accessing RSS feeds.
- Web service clients (e.g. SOAP, XML-RPC or REST).
- Web scraping.

In addition to providing a simple utility, Urdl was intended as an exercise in, and an example of, developing higher-level abstractions on top of Asio.

Urdl may obtained from http://sourceforge.net/projects/urdl.

1. Implementation

Urdl's implementation is based purely on Asio and OpenSSL. Consequently, Urdl is portable to all platforms that are supported by those libraries.

However, the library's public interface avoids exposing the implementation details. For example, there is no access to the underlying socket. This means that, in theory, an implementation may choose from a range of implementation strategies, including:

- Adopting a scheme for connection pooling and reuse.
- Abstracting existing operating system support for HTTP.
- Reusing existing libraries such as libcurl.

2. Downloading content using I/O streams

Many applications need only basic web and networking facilities. To cater to these use cases, Urdl provides a high-level interface that is designed around the familiar C++ I/O streams framework.

Using the library in this way is as easy as constructing a stream object with a specific web resource's URL:

```
urdl::istream is("http://www.boost.org/LICENSE_1_0.txt");
```

then checking whether the resource was opened successfully:

```
if (!is)
{
  std::cerr << "Unable to open URL\n";
  return 1;
}
```

and finally receiving and processing the response:

```
std::string line;
while (std::getline(is, line))
  std::cout << line << "\n";
```

3. Checking for errors

If we are unable to open a resource for any reason, we may obtain a `std::error_code` object that represents the most recent error:

```
urdl::istream is("http://somehost/path");
if (!is)
{
  std::error_code ec = is.error();
  std::cerr << "Unable to open URL: ";
  std::cerr << ec.message() << std::endl;
  return 1;
}
```

Alternatively we may test for specific errors:

```
urdl::istream is("http://somehost/path");
if (!is)
{
  if (is.error() == urdl::http::errc::not_found)
  {
    // Hmm, maybe we can try downloading the file from somewhere else...
  }
}
```

This is achieved by extending the C++11 system error facility with additional error categories:

```
namespace urdl {
  namespace http {
    enum class errc
    {
      ...
      not_found = 404,
      ...
    };
    std::error_code make_error_code(errc e);
  } // namespace http
} // namespace urdl

namespace std {
  template <> struct is_error_code_enum<urdl::http::errc> : true_type {}
} // namespace std
```

4. Setting options to perform an HTTP POST

To perform less common operations, such as uploading over HTTP, we set options on the stream prior to opening the URL:

```
urdl::istream is;

// We're doing an HTTP POST ...
is.set_option(urdl::http::request_method("POST"));

// ... where the MIME type indicates plain text ...
is.set_option(urdl::http::request_content_type("text/plain"));

// ... and here's the content.
```

```
is.set_option(urdl::http::request_content("Hello, world!"));

// All options set, so now we can open the URL.
is.open("http://somehost/path");
```

5. Specifying timeouts

To prevent unresponsive servers from indefinitely hanging a program, the `urdl::istream` class uses a timeout when opening the stream and when reading content. The interface closely follows that provided by Asio's `ip::tcp::iostream` class:

```
urdl::istream is;

// Fail if the URL cannot be opened within 60 seconds.
is.expires_from_now(std::chrono::seconds(60));
is.open("http://somehost/path");
if (!is)
{
  // If the open operation timed out then:
  //   is.error() == boost::system::errc::timed_out
  // holds true.
}

...

// Fail if the expected data is not read within 30 seconds.
is.expires_from_now(std::chrono::seconds(30));

// From here on, use urdl::istream like any other std::istream object.
std::string line;
while (std::getline(is, line))
{
  std::cout << line << std::endl;
}

// If a read operation timed out then:
//   is.error() == boost::system::errc::timed_out
// holds true.
```

6. Working with URLs

The urdl::url class gives us the ability to parse URLs and access their component parts. The constructor:

```
urdl::url url("http://somehost/path");
```

provides a conversion from `std::string` or `const char*` to URLs. If the URL does not parse correctly, the constructor throws an exception of type `std::system_error`. It is this constructor that is used when we write:

```
urdl::istream is("http://somehost/path");
```

We can also use the `urdl::url::from_string` static member function to explicitly parse a URL, with the option of choosing between a throwing overload:

```
urdl::url url = urdl::url::from_string("http://somehost/path");
```

and an overload that does not throw an exception on failure:

```
std::error_code ec;
urdl::url url = urdl::url::from_string("http://somehost/path", ec);
```

7. Integrating with Asio

The `urdl::read_stream` class allows applications to use Urdl's functionality in conjunction with Asio. For example, to asynchronously open a URL, we may write:

```
asio::io_service io_service;
...
urdl::read_stream stream(io_service);
...
stream.async_open("http://somehost/path",
    [](const std::error_code& ec)
    {
      if (ec)
      {
        // URL successfully opened.
        ...
      }
      else
      {
        std::cerr << "Unable to open URL: ";
        std::cerr << ec.message() << std::endl;
      }
    });
```

and the callback function `open_handler` will be invoked once the asynchronous operation completes.

The `urdl::read_stream` class implements Asio's `SyncReadStream` and `AsyncReadStream` type requirements. This means we can use it with generic algorithms, such as the synchronous functions `asio::read` or `asio::read_until`, and the asynchronous functions `asio::async_read` or `asio::async_read_until`:

```
urdl::read_stream stream(io_service);
std::array<char, 512> data;
...
asio::async_read(stream, asio::buffer(data),
    [](const std::error_code& ec, std::size_t length)
    {
      ...
    });
```

Use cases for the `urdl::read_stream` class include:

- Performing concurrent access to multiple URLs, without requiring the overhead and complexity of additional threads.
- Efficiently managing the download of large resources, such as streaming video over HTTP.
- Combining web access with other asynchronous facilities provided by Asio (sockets, timers and so on).