Document Number: N3388=12-0078
Date: 2012-04-23
Reply to: Christopher Kohlhoff <chris@kohlhoff.com>

Using Asio with C++11

This paper is intended as both an introduction to the Asio library and as a brief overview of its implementation and use in conjunction with C++11.

1. C++11 variant of the Asio library

Rather than using the Boost distribution of the Asio library, this paper is based around a variant of Asio that stands alone from Boost. The goals of this variant include:

- using only C++11 language and library features in the interface;
- demonstrating that the library can be implemented using only the C++11 standard library and the facilities provided by the operating system.

This variant is available at http://github.com/chriskohlhoff/asio/tree/cpp11-only.

2. Addressing simple use cases with I/O streams

In many applications, networking is not a core feature, nor is it seen as a core competency of the application's programmers. To cater to these use cases, Asio provides a high-level interface to TCP sockets 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 the remote host's details:

```
tcp::iostream s("www.boost.org", "http");
```

Next, deciding whether you want the socket to give up if the host is non-responsive:

```
s.expires_from_now(std::chrono::seconds(60));
```

Then, send and receive any data as needed. In this case you send a request:

```
s << "GET / HTTP/1.0\r0.2cm\n";
s << "Host: www.boost.org\r\n";
s << "Accept: */*\r\n";
s << "Connection: close\r\n\r\n";
```

And then receive and process the response:

```
std::string header;
while (std::getline(s, header) && header != "\r")
  std::cout << header << "\n";
std::cout << s.rdbuf();
```

If at any time there is an error, the `tcp::iostream` class's `error()` member function may be used to determine the reason for failure:

```
if (!s)
{
  std::cout << "Socket error: " << s.error().message() << "\n";
  return 1;
}
```

3. Understanding synchronous operations

Synchronous operations are functions that do not return control to the caller until the corresponding operating system operation completes. In Asio-based programs their use cases typically fall into two categories:

- Simple programs that do not care about timeouts, or are happy to rely on the timeout behaviour provided by the underlying operating system.
- Programs that require fine grained control over system calls, and are aware of the conditions under which synchronous operations will or will not block.

Asio may be used to perform both synchronous and asynchronous operations on I/O objects such as sockets. However, synchronous operations provide an opportunity to give a simple, conceptual overview of the various parts of Asio, your program, and how they work together. As an introductory example, let's consider what happens when you perform a synchronous connect operation on a socket.

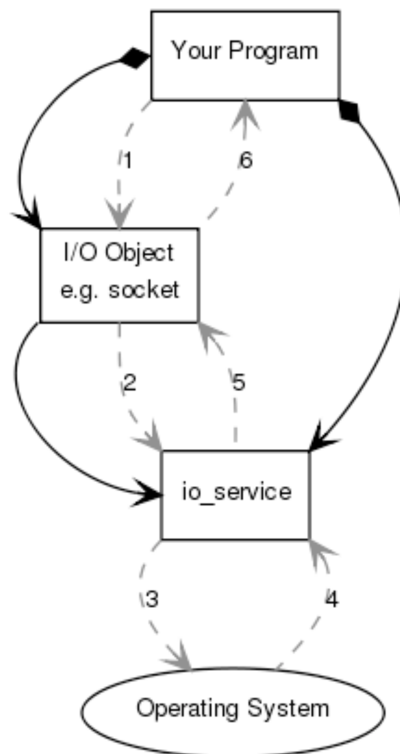
Your program will have at least one `io_service` object. The `io_service` represents your program's link to the operating system's I/O services.

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

To perform I/O operations your program will need an *I/O object* such as a TCP socket:

```
tcp::socket socket(io_service);
```

When a synchronous connect operation is performed, the following sequence of events occurs:



1. Your program initiates the connect operation by calling the I/O object:

   ```
   socket.connect(server_endpoint);
   ```

2. The I/O object forwards the request to the `io_service`.
3. The `io_service` calls on the operating system to perform the connect operation.

4. The operating system returns the result of the operation to the `io_service`.

5. The `io_service` translates any error resulting from the operation into an object of type `std::error_code`. The result is then forwarded back up to the I/O object.

6. The I/O object throws an exception of type `std::system_error` if the operation failed. If the code to initiate the operation had instead been written as:

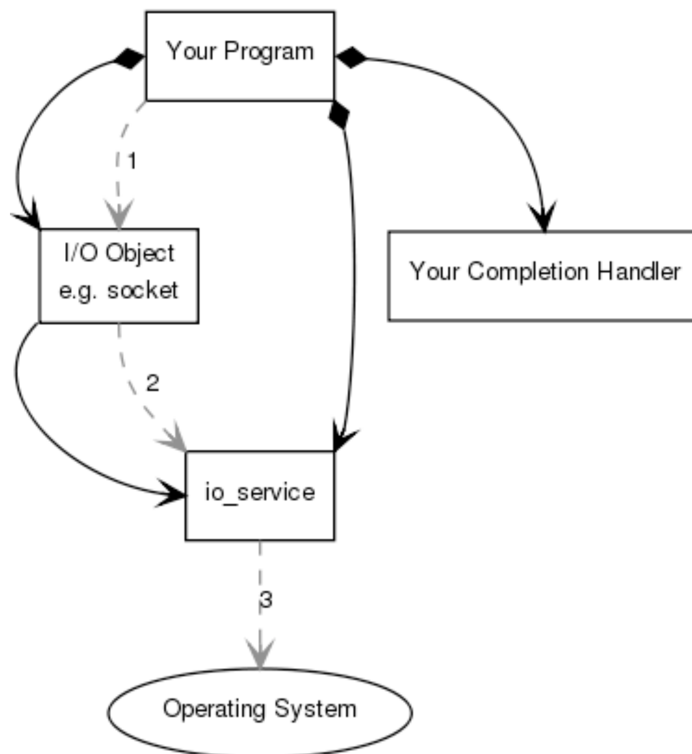
```
std::error_code ec;
socket.connect(server_endpoint, ec);
```

   then the `error_code` variable `ec` would be set to the result of the operation, and no exception would be thrown.

1. Understanding asynchronous operations

Asynchronous operations do not block the caller, but instead involve the delivery of a notification to the program when the corresponding operating system operation completes. Most non-trivial Asio-based programs will make use of asynchronous operations.

When an asynchronous operation is used, the following sequence of events occurs.



1. Your program initiates the asynchronous connect operation by calling the I/O object:

```
socket.async_connect(
    server_endpoint,
    your_completion_handler);
```

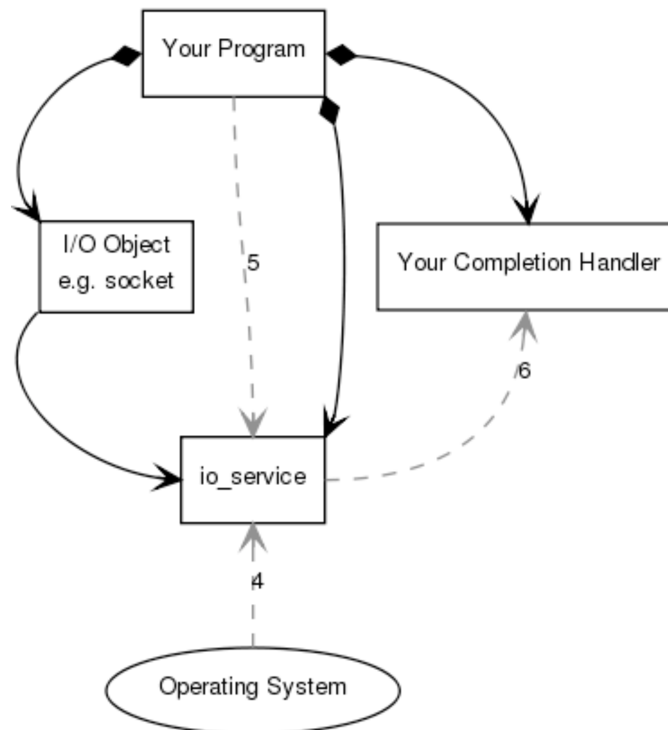
   The `async_connect()` function is an *initiating function*. Initiating functions in Asio are named with the prefix `async_`. An initiating function takes a function object, called a *completion handler,* as its final parameter. In this particular case, `your_completion_handler` is a function or function object with the signature:

```
void your_completion_handler(
    const std::error_code& ec);
```

The exact signature required for the completion handler depends on the asynchronous operation being performed. The Asio reference documentation indicates the appropriate form for each operation.

2. The I/O object forwards the request to the `io_service`.
3. The `io_service` signals to the operating system that it should start an asynchronous connect operation.

Time passes. (In the synchronous case this wait would have been contained entirely within the duration of the connect operation.) During this time the operating system is notionally responsible for the asynchronous operation, which is referred to as outstanding *work*.



4. The operating system indicates that the connect operation has completed by placing the result on a queue, ready to be picked up by the `io_service`.
5. Your program must make a call to `io_service::run()` (or to one of the similar `io_service` member functions) in order for the result to be retrieved. A call to `io_service::run()` blocks while there is outstanding work[1]. You would typically call it as soon as you have started your first asynchronous operation.
6. While inside the call to `io_service::run()`, the `io_service` dequeues the result of the operation, translates it into an `error_code`, and then passes it to your completion handler.

2. Chaining asynchronous operations

An asynchronous operation is considered outstanding work until its associated completion handler is called and has returned. The completion handler may in turn call other initiating functions, thus creating more outstanding work.

---

[1] Outstanding work is logically represented by an object of type `io_service::work`. The `io_service::run()` function blocks if one or more `io_service::work` objects exist, and all asynchronous operations behave *as-if* they have an associated `io_service::work` object.

Consider the case where a connect is followed by other I/O operations on the socket:

```
socket.async_connect(
    server_endpoint,
    [&](std::error_code ec)
    {
      if (!ec)
      {
        socket.async_read_some(
            asio::buffer(data),
            [&](std::error_code ec, std::size_t length)
            {
              if (!ec)
              {
                async_write(socket,
                    asio::buffer(data, length),
                    [&](std::error_code ec, std::size_t length)
                    {
                      // ...
                    });
              }
            });
      }
    });
```

The completion handler for the asynchronous connect operation, here expressed as a C++11 lambda, initiates an asynchronous read operation. This has its own associated outstanding work, and its completion handler creates yet more work in the form of an asynchronous write. Consequently, the `io_service::run()` does not return until all operations in the chain have completed.

Of course, in real programs these chains are often longer and may contain loops or forks. In these programs, `io_service::run()` may execute indefinitely.

3. Handling errors

Asio's approach to error handling is based on the view that exceptions are not always the right way to handle errors. In network programming, for example, there are commonly encountered errors such as:

- You were unable to connect to a remote IP address.
- Your connection dropped out.
- You tried to open an IPv6 socket but no IPv6 network interfaces are available.

These might be exceptional conditions, but equally they may be handled as part of normal control flow. If you reasonably expect it to happen, it's not exceptional. Respectively:

- The IP address is one of a list of addresses corresponding to a host name. You want to try connecting to the next address in the list.
- The network is unreliable. You want to try to reestablish the connection and only give up after *n* failures.
- Your program can drop back to using an IPv4 socket.

Whether the error is exceptional depends on the context within the program. Furthermore, some domains may be unwilling or unable to use exceptions due to code size or performance constraints. For this reason, all synchronous operations provide both throwing:

```
socket.connect(server_endpoint); // Throws std::system_error on error.
```

and non-throwing overloads:

```
std::error_code ec;
socket.connect(server_endpoint, ec); // Sets ec to indicate error.
```

For similar reasons, Asio does not use separate completion handlers based on whether or not an operation completes with an error. To do so would create a fork in the chain of asynchronous operations which may not match the program's idea of what constitutes an error condition.

4. Managing object lifetimes

When using asynchronous operations, a particular challenge is object lifetime management. Asio adopts an approach where there is no explicit support for managing object lifetime. Instead, the lifetime requirements are placed under program control according to rules based on how an initiating function is declared:

- **By value, const reference and rvalue reference parameters.**

  These are copied or moved as required by the library implementation until no longer required. For example, the library implementation maintains a copy of the completion handler object until after the handler is called.

- **Non-const reference parameters, this pointer.**

  The program is responsible for ensuring that the object remains valid until the asynchronous operation is complete.

An approach employed in many Asio-based programs is to tie object lifetime to the completion handler. This may be achieved using `std::shared_ptr<>` and `std::enable_shared_from_this<>`:

```
class connection :
    std::enable_shared_from_this<connection>
{
  tcp::socket socket_;
  vector<unsigned char> data_;
  // ...
  void start_read()
  {
    socket_.async_read_some(socket_,
        asio::buffer(data_),
        std::bind(&connection::handle_read,
          shared_from_this(), _1, _2));
  }
  // ...
};
```

With C++11, this approach can give an excellent tradeoff between usability and performance. Asio is able to leverage movable completion handlers to minimise the costs associated with reference counting. Since programs typically consist of chains of asynchronous operations, ownership of the pointer can be transferred along the chain; the reference count is only updated at the beginning and end of the chain.

However, some programs require fine grained control over object lifetimes, memory usage, and execution efficiency. By putting object lifetime under program control, these use cases are also supported. For example, an alternative approach is to create all objects at program startup. The completion handlers are then not required to play a role in object lifetime, and may be trivially cheap to copy.

5. Optimising memory allocation

Many asynchronous operations need to allocate an object to store state associated with the operation. For example, a Windows implementation needs `OVERLAPPED`-derived objects to pass to Win32 API functions.

By default, Asio will allocate space for this bookkeeping information using `::operator new()`. However, chains of asynchronous operations provide an opportunity for optimisation here. Each chain can have a block of memory associated with it, and the same block can be reused for each sequential operation in the chain. This means it is possible to write asynchronous protocol implementations that perform no ongoing memory allocations.

The hook for this custom memory allocation is the asynchronous operation's completion handler. The handler identifies the larger context in which the operation is being performed. By passing this completion handler to the initiating function, Asio is able to allocate the necessary memory prior to signalling the operating system to start the asynchronous operation.

6. Dealing with concurrency

Protocol implementations typically involve the coordination of multiple chains of asynchronous operations. For example, one chain of operations may handle message sending, another receiving, and a third may implement application-layer timeouts. All of these chains require access to common variables, such as sockets, timers, and buffers. Furthermore, these asynchronous operations may continue indefinitely.

Asynchronous operations provide a way to realise concurrency without the overhead and complexity of threads. However, Asio's interface is designed to support a spectrum of threading approaches, some of which are outlined below.

Single-threaded designs

Asio guarantees that completion handlers will be called only from within `io_service::run()`[2]. Therefore, by calling `io_service::run()` from exactly one thread, a program can prevent parallel execution of handlers.

This design is the recommended starting point for most programs, as no explicit synchronisation mechanism is required. However, care must be taken to keep handlers short and non-blocking.

Use threads for long running tasks

A variant of the single-threaded design, this design still uses a single `io_service::run()` thread for implementing protocol logic. Long running or blocking tasks are passed to a background thread and, once completed, the result is posted back to the `io_service::run()` thread.

By using a shared-nothing message passing approach[3], programs can ensure that objects are not shared between the `io_service::run()` thread and any background worker thread. Thus, explicit synchronisation is still not required.

Multiple io_services, one thread each

In this design, I/O objects are assigned a "home" `io_service`, which is run from a single thread. Different objects should communicate only by message passing.

This design can make more effective use of multiple CPUs, while limiting sources of contention. Explicit synchronisation is not required, but handlers must be kept short and non-blocking.

---

[2] Or from one of the similar `io_service` member functions `run_one()`, `poll()` or `poll_one()`.

[3] Asio supports message passing through the `io_service` member functions `post()` and `dispatch()`.

One io_service, multiple threads

The `io_service::run()` function may be called from multiple threads to set up a thread pool for a single `io_service`. The implementation distributes available work across the threads in an arbitrary fashion.

As completion handlers can be called from any of the threads then, unless the protocol implementation is trivial and consists of a single chain of operations, some form of synchronisation may be required. Asio provides the `io_service::strand` class for this purpose.

A strand prevents concurrent execution of any completion handler associated with it. In the example above, where we have a protocol implementation that consists of three chains of operations (for sending, receiving and timeouts), a strand ensures that the related chains' handlers are serialised. Other protocol implementations running on other strands are still able to utilise any of the other threads in the pool. Furthermore, unlike a mutex, if a strand is "in use" it will not block the calling thread, but will instead switch to run other ready handlers on other strands to keep the thread busy.

As with custom memory allocation, strand synchronisation uses an hook associated with the completion handler. That is, the completion handler identifies the larger context in which the operation is being performed. This custom invocation hook allows the synchronisation mechanism to scale up to abstractions that are based on composition of operations, as we will see below.

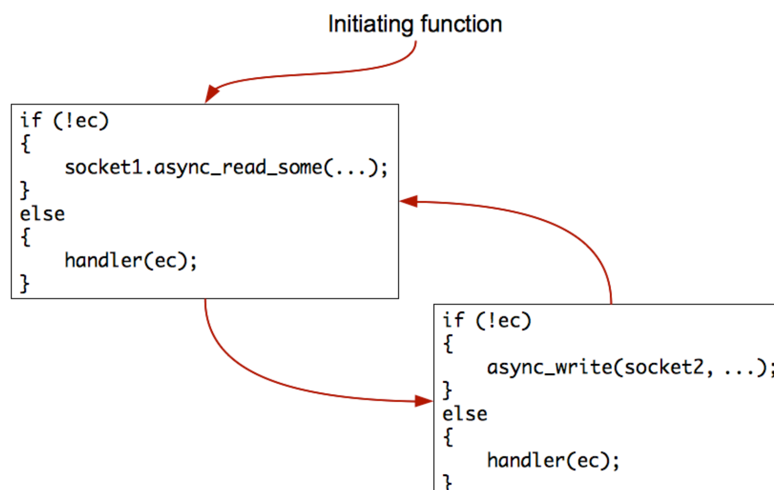
7. Passing the buck: developing efficient abstractions

A key design goal of Asio is to support the creation of higher level abstractions. The primary mechanism for this is composition of asynchronous operations. In Asio terminology, these are known simply as "composed operations".

As an example, consider a hypothetical user-defined asynchronous operation that implements the pass-through transfer of all data read from one socket to another. The initiating function might be declared as follows:

```
template <typename Handler>
void async_transfer(
    tcp::socket& socket1, tcp::socket& socket2,
    std::array<unsigned char, 1024>& working_buffer,
    Handler handler);
```

This function would be implemented in terms of two underlying asynchronous operations: a read from `socket1` and a write to `socket2`. Each of these operations has an intermediate completion handler, and the relationship between them is illustrated in the image below:



These intermediate completion handlers can "pass the buck" by customising the allocation and invocation hooks to simply call the user's completion handler's hooks. In this way, the composition has deferred all choices on memory allocation and synchronisation to the user of the abstraction. The user of the abstraction gets to select the appropriate tradeoff between ease of use and efficiency, and does not need to pay for any synchronisation cost if explicit synchronisation is not required.

Asio provides a number of these composed operations out-of-the-box, such as the non-member functions `async_connect()`, `async_read()`, `async_write()` and `async_read_until()`. Closely related composed operations may also be grouped in objects as in Asio's `buffered_stream<>` and `ssl::stream<>` templates.

8. Scope and extensibility

Given the size of the Asio library, the proposals put before WG21 have been limited to a minimal viable subset that is focused primarily on networking with TCP and UDP, buffers and timers. However, Asio's interface is designed to allow both user and implementor extensibility through a number of mechanisms. Some of these mechanisms are described below.

Additional I/O services

The `io_service` class implements an extensible, type-safe, polymorphic set of *I/O services*, indexed by service type. I/O services exist to manage the logical interface to the operating system on behalf of the I/O objects. In particular, there are resources that are shared across a class of I/O objects. For example, timers may be implemented in terms of a single timer queue. The I/O services manage these shared resources, and are designed so that they impose no cost if not used.

Asio implements additional I/O services to provide access to certain operating system-specific functionality. Examples include:

- Windows-specific services for performing overlapped I/O on `HANDLE`s
- A Windows-specific service for waiting on kernel objects such as events, processes and threads.
- A POSIX-specific service for stream-oriented file descriptors
- A service for safe integration of signal handling via `signal()` or POSIX `sigaction()`.

New I/O services may be added without impacting existing users of the library.

Socket type requirements

Although the Asio-based proposal is limited to TCP and UDP sockets, the interface is based around type requirements, such as `Protocol` and `Endpoint`. These type requirements are designed to allow the library to work with other types of sockets. The Asio library itself has used these type requirements to add support for ICMP and UNIX domain sockets.

Stream type requirements

The Asio library defines several type requirements for both synchronous and asynchronous stream-oriented I/O. These type requirements are implemented by the TCP socket interface, and also by abstractions such as `ssl::stream<>` and `buffered_stream<>`. By implementing these type requirements, a class may be used with the composed operations like `async_read()`, `async_read_until()` and `async_write()`. The type requirements are also intended for use in higher level abstractions such as an asynchronous wrapper over HTTP.