Document Number: N3360=12-0050
Date: 2012-02-03
Reply to: Christopher Kohlhoff <chris@kohlhoff.com>

Networking Library Status Report

Since the revised TR2 proposal based on Boost.Asio (N2175) was submitted in 2007, the use and application of the library has greatly expanded among C++ programmers. As the author would like to see the TR2 proposal revived, the purpose of this document is to give an overview of what has happened in the interim.

1. Expanded User Base

The library has found a home in thousands of software projects, including:

- The small — mobile and embedded applications
- The large — highly scalable internet-facing servers
- The fast — ultra-low latency applications, utilising InfiniBand and 10 Gb Ethernet
- The widely-used — online games; business, consumer and professional software

The library has been used to create abstractions for a range of different network protocols. A small selection of some freely-available protocol implementations is listed below:

- HTTP — Pion (http://www.pion.org)
- DNS — BIND 10 (http://www.isc.org/bind10)
- XMPP — Swift IM (http://swift.im)
- WebSockets — websocketpp (http://github.com/zaphoyd/websocketpp)
- BitTorrent — libtorrent (http://www.libtorrent.org)
- Distributed Network Protocol 3 — dnp3 (http://code.google.com/p/dnp3/)
- FIX Adapted for Streaming (FAST Protocol) — QuickFAST (http://code.google.com/p/quickfast/)

2. Improved Implementation

Much effort has been spent on reducing the abstraction penalty, in particular:

- Improving single-threaded performance
- Improving scalability across multiple processors
- Minimising latency
- Reducing compile times
- Reducing generated code size

Most of this work has been accomplished without changes to the library's interface. In fact, other than the new features described below, the library interface and core concepts have changed little.

3. Wider Platform Support

The Boost.Asio library now supports (or, via end-user patches, has supported) many different operating systems and platforms, including Windows, Linux, Mac OS X, FreeBSD, NetBSD, AIX, Solaris, HP-UX, iOS, Android, Windows CE, QNX Neutrino, VxWorks and Symbian.

4. Added Platform Features

Since 2007, feature coverage has expanded beyond basic networking facilities (i.e. TCP and UDP sockets, IP version independence, buffer management, etc.) to include:

- Raw sockets and ICMP
- Sequenced packet sockets
- UNIX domain sockets
- POSIX file descriptors
- Serial ports
- Signals
- Windows overlapped I/O (such as files, named pipes, `TransmitFile`, etc.)
- Windows kernel objects (such as anonymous pipes, events, semaphores, etc.)

In addition, people have expanded with their own offerings, including support for:

- Directory and file monitoring
- Packet Capture
- Process management

While these are not currently intended for inclusion in a standards proposal for networking, it demonstrates that the library allows extensibility without impacting the core API.

5. C++11 Support

In the past year, the library interface has been modified to make use of certain C++11 features.

5.1. Movable I/O Objects

I/O objects such as sockets, while they remain non-copyable, now support move construction and move assignment.

5.2. Movable Completion Handlers

As an optimisation, user-defined completion handlers may provide move constructors, and Boost.Asio's implementation will use a handler's move constructor in preference to its copy constructor. In certain circumstances, Boost.Asio may be able to eliminate all calls to a handler's copy constructor.

In conjunction with the library's custom memory allocation facility, this makes it possible to write programs that use `std::shared_ptr<>` for safe memory management, but perform no ongoing memory allocations or reference counting. Experience also shows that this approach works well with composition to create efficient higher-level abstractions.

5.3. Chrono Support

New timer interfaces have been added which, rather using the Boost.DateTime library, are based around C++11's clocks, durations and time points.

5.4. Variadic templates

Where possible, multiple overloads to support variable numbers of arguments have been updated to use variadic template functions.

6. New Usage Styles

Asynchronous operations in Boost.Asio are built around the concept of completion handlers, a.k.a. function objects as callbacks. As of 2007, most applications implemented these using hand-rolled

function objects, or by using function object binders (`boost::bind`, `std::tr1::bind` and now `std::bind`). For example:

```
void my_class::read_handler(const error_code& ec, size_t length)
{
  ...
}
...
my_socket.async_read_some(my_buffer,
    std::bind(&my_class::read_handler, this, _1, _2));
```

Since then, library users have explored and used other approaches, some of which are illustrated below.

1. C++11 Lambdas

C++11's monomorphic lambdas may be used to implement the completion handler's code at the point where the operation is initiated:

```
my_socket.async_read_some(my_buffer,
    [&](const error_code& ec, size_t length)
    {
      ...
    });
```

2. Promises and Futures

C++11's promises and futures may be used to perform a synchronous wait for an asynchronous operation:

```
auto p = make_shared<promise<size_t>>();
my_socket.async_read_some(my_buffer, my_promise_adapter(p));
size_t result = p->get_future().get();
```

3. "Stackless" Coroutines

By employing a Duff's Device-like approach, and some suitably defined macros, more complex chains of asynchronous operations may be written in a concise fashion:

```
while (!ec)
{
  yield my_socket->async_read_some(my_buffer, *this);
  if (ec) break;
  yield async_write(*my_socket, buffer(my_buffer, length), *this);
}
```

4. "Stackful" Coroutines

Similarly, libraries like the recently accepted Boost.Context may also be used to undo the inversion of control, with the added benefit of preserving the call stack at the point where an asynchronous operation is initiated:

```
while (!ec_)
{
  my_socket->async_read_some(my_buffer, *this);
  my_continuation->suspend();
  if (ec_) break;
  async_write(*my_socket, buffer(my_buffer, length_), *this);
  my_continuation->suspend();
}
```