Boost.Asio

Christopher Kohloff
번역 박종혁(reister@naver.com)

Boost.Asio 는 현대적인 C++접근법을 통해, 개발자에게 비 동기적인 입출력 모델을 제공하는 플랫폼 독립적인 네트워크 프로그래밍 라이브러리 입니다.

목차

Boost.Asio 사용하기

지원되는 플랫폼

아래에 나오는 플랫폼과 컴파일러는 이미 테스트가 완료되었습니다..

- Visual C++7.1 과 Visual C++ 8.0 을 사용하는 Win32 와 Win64
- MinGW 를 사용하는 Win32
- CygWin 을 사용하는 Win32(__USE_W32_SOCKETS 가 반드시 define 되어야 합니다.)
- g++3.3 이상의 버전을 사용하는 Linux(2.4 혹은 2.6 커널)
- g++3.3 이상의 버전을 사용하는 Solaris
- g++3.3 이상의 버전을 사용하는 Max OS X 10.4

아래 플랫폼에서도 아마 잘 될 겁니다.

- XL C/X++ v9 를 사용하는 AIX 5.3
- 패치된 aC++ A.06.14 를 사용하는 HP-UX 11i v3
- g++3.3 이상의 버전을 사용하는 QNX Neutrino
- Sun Studio 11 이상을 사용하는 Solaris
- Compaq C++ v7.1 을 사용하는 Tru64 v5.1
- Borland C++ 5.9.2 를 사용하는 Win32

종속성

아래 라이브러리들은 Boost.Asio 를 사용하기 위해 반드시 설치되어 있어야 하는 것들입니다.

- boost::system::error_code 와 boost::system::system_error 클래스를 사용하기 위해 Boost.System 이 필요합니다.
- boost::regex 를 오버로드한 read_until()이나 async_resd_until()를 사용하려면 Boost.Regex 가 필요합니다.(필수적이지는 않습니다.)
- Boost.Asio 의 SSL 지원을 이용하고 싶다면 OpenSSL 이 필요합니다.(이것도 필수는 아닙니다.)

또한, 몇 가지 예제에서는 Boost.Thread 나 Boost.Date_Time 또는 Boost.Serialization 라이브러리를 사용합니다.

Note

MSVC 나 Borland c++ 에서는, Boost.Date_Timer 과 Boost.Regex 라이브러리의 자동링크를 막기 위해 프로젝트 세팅에 DBOOST_DATE_TIME_NO_LIB 과 DBOOST_REGEX_NO_LIB 매크로를 추가해야 합니다.

Boost 라이브러리 빌드하기

Boost.Asio 의 하부 라이브러리들을 사용하기 위해서는 Boost 를 빌드해야 할 수도 있습니다. Boost.Asio 와 예제들은 Boost 다운로드 패키지에 포함된 아래의 명령으로 실행될 수 있습니다.

```
bjam --with-system --with-thread --with-date_time --with-regex --with-serialization stage
```

이것은 여러분이 이미 bjam 을 빌드했다고 가정합니다. Boost.Build 문서에서 더 많은 정보를 얻어보세요.

Macros

Boost.Asio 의 행동을 제어하기 위해 사용되는 매크로들을 아래에 나열했습니다.

Macro	Description
BOOST_ASIO_ENABLE_BUFFER_DEBUGGING	잘못된 버퍼를 읽거나 쓰는 문제를 해결하기 위한 Boost.Asio 의 버퍼 디버깅 지원 기능을 활성화 합니다. (예. 만약 std::string 객체가 쓰기 동작이 끝나기도 전에 파괴되는 경우). 이 매크로는 MSVC 에서는 itorator 디버깅 지원을 활성화 해 두었다면, 일부러 BOOST_ASIO_DISABLE_BUFFER_DEBUGGING 매크로를 정의하지 않는 이상, 자동으로 정의됩니다. g++에서는 표준 디버깅 라이브러리가 활성화 되어 있다면(_GLIBCXX_DEBUG 가 정의되어있을 때), BOOST_ASIO_DISABLE_BUFFER_D`BUGGING 을 일부러 정의하지 않는 이상 자동으로 정의됩니다.
BOOST_ASIO_DISABLE_BUFFER_DEBUGGING	명시적으로 Boost.Asio 의 버퍼 디버깅 지원 기능을 비활성화 합니다.
BOOST_ASIO_DISABLE_DEV_POLL	select 기반의 구현을 위해, 명시적으로 Solasis 에서의 /dev/poll 지원을 비활성화 합니다.
BOOST_ASIO_DISABLE_EPOLL	select 기반의 구현을 위해, 명시적으로 Linux 의 epoll 지원을 비활성화 합니다. Explicitly disables epoll support on Linux, forcing the use of a select-based implementation.
BOOST_ASIO_DISABLE_KQUEUE	select 기반의 구현을 위해, 명시적으로 Max OS X 와 BSD 류의 kqueue 지원을 비활성화 합니다.
BOOST_ASIO_DISABLE_IOCP	select 기반의 구현을 위해, 명시적으로 Windows 의 IOCP 지원을 비활성화 합니다.
BOOST_ASIO_NO_WIN32_LEAN_AND_MEAN	Boost.Asio 는 기본적으로 Windows 에서 컴파일 될 때, 인클루드 되는 수많은 Windows SDK 의 헤더파일과 요소들을 최소화 하기 위해 WIN32_LEAN_AND_MEAN 을 자동으로 정의합니다. 이 때, BOOST_ASIO_NO_WIN32_LEAN_AND_MEAN

Macro	Description
	매크로는 WIN32_LEAN_AND_MEAN 매크로가 정의 되는 것을 방지합니다.
BOOST_ASIO_NO_DEFAULT_LINKED_LIBS	MSVC 나 Borland C++을 윈도우에서 컴파일 할 때, Boost.Asio 는 자동으로 소켓 지원에 필요한 Windows SDK 라이브러리들을 링크합니다. (예를들어 Windows CE 라면 ws2_32.lib and mswsock.lib, or ws2.lib 같은 것들)
BOOST_ASIO_SOCKET_STREAMBUF_MAX_ARITY	basic_socket_streambuf 클래스 템플릿의 connect 멤버함수에 전달된 인자의 최대 개수를 결정합니다. 기본값은 5 입니다.
BOOST_ASIO_SOCKET_IOSTREAM_MAX_ARITY	basic_socket_iostream 클래스 템플릿의 생성자와 connect 멤버함수에 전달될 인자의 최대 개수를 결정합니다. 기본값은 5 입니다.
BOOST_ASIO_ENABLE_CANCELIO	구 버전의 Windows 에서 CancelIo 함수를 사용할 수 있게 합니다. 만약, 이 매크로가 활성화 되어있지 않다면, Windows XP, Windows Server 2003 이하의 버전에서는, 소켓 객체가 cancel() 함수를 호출할 때 마다 항상 asio::error::operation_not_supported 와 함께 실패할것입니다. 반면에, Windows Vista 와 Windows Server 2008 이상에서는 항상 CancelIoEx 가 사용됩니다. CancelIo 함수의 사용을 활성화 하기 전에 두 가지 이슈를 유념해 두어야 합니다. *CancelIo 함수는 현재 스레드에서 초기화된 비동기적인 작업만을 취소할 것입니다. *CancelIo 함수는 에러 없이 완료된 것 처럼 보일지도 모릅니다. 하지만, 끝나지 않은 작업에 대한 취소 요청은 운영체제에 의해서, 조용히 무시될 수도 있습니다.성공했는지 실패했는지의 여부는 설치된 드라이버에 달려있는 것 같습니다. 적합한 취소방법으로 다음의 대안들을 고려해보십시오. For portable cancellation, consider using one of the following alternatives: *BOOST_ASIO_DISABLE_IOCP 매크로를 정의함으로써, asio 의 IOCP 백 엔드를 비활성화 하십시오. *끝나지 않은 작업(outstanding)을 취소하는 것과 동시에 소켓을 닫기 위해 소켓 오브젝트의 close()함수를 이용하십시오.

튜토리얼

기본적인 스킬

첫 번째 섹션의 튜토리얼 프로그램은 **asio** 툴킷을 사용하는데 필요한 기본적인 개념을 소개합니다. 이 튜토리얼 프로그램들은 여러분들이 복잡한 네트워크 프로그래밍의 세상으로 뛰어들기 전에, 간단한 비동기 타이머를 통해서 기본적인 스킬을 알려드립니다.

- Timer.1 – 동기적으로 타이머 사용하기
- Timer.2 – 비동기적으로 타이머 사용하기
- Timer.3 – 인자를 핸들러에 바인딩하기
- Timer.4 – 멤버함수를 핸들러로 사용하기
- Timer.5 – 멀티 스레드 프로그램에서 핸들러를 동기화하기

소켓에 대한 소개

이 섹션의 튜토리얼 프로그램은 간단한 **Client-Server** 프로그램을 개발하는데 **asio** 가 어떻게 사용되는지를 보여줍니다. 이 튜토리얼 프로그램은 **daytime** 프로토콜 기반에서 **TCP** 와 **UDP** 를 모두 지원합니다.

처음 세개의 튜토리얼 프로그램은 **TCP** 를 이용한 **daytime** 프로토콜의 구현입니다.

- Daytime.1 - 동기적인 TCP daytime client
- Daytime.2 - 동기적인 TCP daytime server
- Daytime.3 - 비동기적인 TCP daytime server

다음 세개의 튜토리얼 프로그램은 **UDP** 를 이용한 **daytime** 프로토콜의 구현입니다.

- Daytime.4 - 동기적인 UDP daytime client
- Daytime.5 - 동기적인 UDP daytime server
- Daytime.6 - 비동기적인 UDP daytime server

이 섹션의 마지막 튜토리얼 프로그램은 **asio** 를 사용하면 얼마나 쉽게 하나의 프로그램에서 **TCP** 와 **UDP** 서버를 연동할 수 있는지를 설명합니다.

- Daytime.7 – TCP 와 UDP 가 결합된 비동기적 서버

TIMER. 1 – 동기적으로 타이머 사용하기

이 튜토리얼 프로그램은 어떻게 프로그램을 블록상태로 대기하게 하는가를 보여줌으로써, asio 를 소개합니다.

필요한 헤더파일을 인클루드 하는 것부터 시작해 봅시다.

모든 asio 클래스들은 단순히 "asio.hpp"만 인클루드 하면 사용할 수 있습니다.

```
#include <iostream>
#include <boost/asio.hpp>
```

이 예제는 타이머를 사용하기 때문에, 시간을 다루기 위해 적당한 Boost.Date_Time 헤더파일을 인클루드 해야 합니다.

```
#include <boost/date_time/posix_time/posix_time.hpp>
```

asio 를 사용하는 모든 프로그램은 적어도 하나이상의 boost::asio::io_service 객체를 가지고 있어야 합니다. 이 클래스는 I/O 기능에 접근하 것을 제공하지요. 그럼, 제일 먼저 이 타입의 객체 하나를 main 함수에 선언해봅시다.

```
int main()
{
  boost::asio::io_service io;
```

그 다음 type boost::asio::deadline_timer 타입의 객체를 선언합니다. I/O 기능을 제공하는 asio 의 핵심 클래스들은 (지금의 경우에는 타이머 기능 이겠죠.) 항상 생성자의 첫 번째 인자로 io_service 의 참조 값을 받습니다. 생성자의 두 번째 인자는, 5 초 후에 타이머가 죽도록 세팅 하는것 이지요.

```
  boost::asio::deadline_timer t(io, boost::posix_time::seconds(5));
```

이 간단한 예제를 통해, 우리는 타이머에 의한 블록 대기를 수행할 수 있습니다. 이것은, 타이머가 생성 된지 5 초가 지나서 타이머가 죽기 전까지는 boost::asio::deadline_timer::wait()은 리턴 되지 않을 것 이라는 겁니다.(wait 이 호출 된지 5 초가 아니에요.)

deadline 타이머는 항상 둘 중 하나의 상태를 가집니다. 죽었거나, 죽지 않았거나.
만약 타이머가 죽어있는 상태에서 boost::asio::deadline_timer::wait()가 호출되면, 그 즉시 리턴 되어 버릴 겁니다.

```
t.wait();
```

마지막으로, 타이머가 죽는 순간, 우리는 지겹도록 보아왔던 "Hello,World!" 메시지를 출력할 수 있게 되었습니다.

```
  std::cout << "Hello, world!\n";

  return 0;
}
```

TIMER.2 - 비동기적으로 타이머 사용하기

이 튜토리얼 프로그램은 타이머의 비 동기적인 대기를 수행하기 위해 튜토리얼 Timer1 을 수정함으로써, 어떻게 asio 의 비 동기 콜백 기능을 사용하는지를 설명합니다.

```
#include <iostream>


#include <boost/asio.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
```

asio 의 비동기적인 기능을 사용한다는 것은, 비동기적으로 수행이 완료 될 수 있는 콜백 함수를 가지고 있다는 뜻입니다. 그럼 일단 비동기 대기가 종료될 때 호출되는 print 함수를 정의해 봅시다.

```
void print(const boost::system::error_code& /*e*/)
{
  std::cout << "Hello, world!\n";
}

int main()
{
  boost::asio::io_service io;

  boost::asio::deadline_timer t(io, boost::posix_time::seconds(5));
```

그 다음엔, 튜토리얼 Timer.1 에서 블록대기를 했던 것 대신에, 비동기 대기를 수행하기 위해 boost::asio::deadline_timer::async_wait()함수를 호출할 수 있습니다.이 함수를 호출될 때 앞에서 정의한 print 함수의 콜백 핸들러를 넘겨줘야겠지요?

```
  t.async_wait(print);
```

마지막으로, io_service 객체의 boost::asio::io_service::run() 멤버함수를 반드시 호출해 주어야 합니다.

asio 라이브러리에서 콜백 핸들러는, 오직 현재 boost::asio::io_service::run() 함수를 호출한 스레드로 부터 호출될 것 이라는 것을 보장해 줍니다. 따라서, boost::asio::io_service::run()함수가 호출되지 않았다면, 비동기 대기의 완료 콜백은 절대로 일어나지 않습니다.

또한 boost::asio::io_service::run()함수는, 아직 "할 일"이 남아있는 동안에는 계속 수행됩니다.
이 예제에서 그 "할 일"이란, 타이머의 비동기적인 대기입니다. 그래니까 타이머가 죽고 콜백이 완료되기 전까지는 함수의 호출이 리턴되지 못하는 것이죠.

그리고, boost::asio::io_service::run()를 호출하기 전에 io_service 에게 무언가 할 일을 주는 것은 굉장히 중요합니다.
예를 들어 위에서 boost::asio::deadline_timer::async_wait()의 호출을 생략했다면, io_service 는 아무런 할 일이 없습니다.
그러니, 당연 하게도 boost::asio::io_service::run()는 곧바로 리턴 되겠지요.

```
  io.run();
  return 0;
}
```

TIMER.3 – 인자를 핸들러에 바인딩하기

이 튜토리얼 프로그램에서는 1 초마다 한번씩 타이머에 붙이들어오도록 튜토리얼 Timer2 를 고쳐봅시다. 이것을 통해 여러분은 어떻게 추가적인 파라매터를 여러분의 핸들러함수에 전달할 수 있는지를 보게될 것입니다.

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
```

타이머를 반복적으로 사용하도록 구현하기 위해서 여러분은 콜백 함수 내부에서 타이머의 종료시간을 바꾸어주어야합니다. 그리고, 새로운 비동기 대기를 시작 해야지요. 이것은 명백하게 콜백함수가 타이머 객체에 접근 할 수 있게 만들 필요가 있다는 것을 뜻합니다. 이를 위해 우리는 print 함수에 두개의 매개변수를 추가합니다.

- 타이머 객체의 포인터.
- 타이머가 6 번째 켜질 때 프로그램을 종료하기 위한 카운터

```
void print(const boost::system::error_code& /*e*/,
    boost::asio::deadline_timer* t, int* count)
{
```

앞에서 언급한 것 처럼, 이 튜토리얼 프로그램은, 카운터가 6 번 켜지는 것을 만들기 위해 카운터를 사용합니다. 하지만, 여러분은 io_service 를 멈추기 위한 명시적인 호출이 없다는 것을 알아 채셨을 겁니다. 튜토리얼 Timer2 로 돌아가서 생각해봅시다. boost::asio::io_service::run() 함수는 더 이상 “할일”이 없을 때 완료된다는 것을 배운적이 있지요? 새롭게 시작된 타이머의 비동기적 대기가 없기 때문에, 카운터가 5 에 다다르면, io_service 는 할일을 다 마치고 실행울 중지하게 되는겁니다.

```
if (*count < 5)
  {
    std::cout << *count << "\n";
    ++(*count);
```

그 다음에, 우리는 이전의 종료시간으로부터 다음 종료시간을 1 초 만큼 옮깁니다. 예전 종료시간으로부터 새로운 종료시간을 계산하기 때문에, 타이머가 핸들러를 처리하는데 지연이 생기더라도, 시간이 뒤로 밀리지 않는다고 보장 할 수 있는거지요.

```
    t->expires_at(t->expires_at() + boost::posix_time::seconds(1));
```

이제 타이머를 통해 새로운 비동기적 대기를 시작합니다. 여러분도 아시겠지만, boost::bind()함수는 콜백 함수에 여분의 매개변수를 연관시키는데 사용됩니다. boost::asio::deadline_timer::async_wait() 함수는 void(const boost::system::error_code&) 형식의 핸들러 함수(혹은 함수객체)을 받습니다. 추가적인 파라매터를 바인딩하는 것은 print 함수를 이러한 형식에 맞는 함수객체로 변환시켜 줍니다.

Boost::bind() 함수를 어떻게 쓰는지 더 알고싶다면 Boost.Bind documentation 을 참고하시길 바랍니다.

이 예제에서, boost::bind()에 인자로 들어가는 boost::asio::placeholders::error 는 핸들러에 전해지는 에러 객체를 위한 이름이 정해져있는 지정자 (placeholder) 입니다. boost::bind()를 이용해서 비동기적인 오퍼레이션을 초기화한다면, 여러분은 핸들러의 매개변수 목록에 적합한 인자를 지정해 주어야 합니다. 튜토리얼 Timer.4 에서는 콜백핸들러를 필요로 하지 않기 때문에 이 지정자가 제거된 것을 볼수 있을겁니다.

```
t->async_wait(boost::bind(print,
          boost::asio::placeholders::error, t, count));
  }
}

int main()
{
  boost::asio::io_service io;
```

타이머가 6 번째 켜졌을 때 프로그램을 멈추기 위해서, 새로운 count 변수가 추가되었습니다.

```
  int count = 0;
  boost::asio::deadline_timer t(io, boost::posix_time::seconds(1));
```

4 번째 단계에서 우리는, main 함수에서 boost::asio::deadline_timer::async_wait()를 호출할 때, print 함수에 필요한 추가적인 매개변수를 바인드합니다.

```
t.async_wait(boost::bind(print,
        boost::asio::placeholders::error, &t, &count));

  io.run();
```

마지막으로, count 변수가 print 핸들러 함수에서 사용되었음이 증명하기 위해서, 새로운 값을 출력해 봅니다.

```
std::cout << "Final count is " << count << "\n";
  return 0;
}
```

TIMER.4 - 멤버함수를 핸들러로 사용하기

이 튜토리얼에서는 클래스 멤버 함수를 어떻게 콜백 핸들러로 사용하는지는 보여줍니다. 이 프로그램이 하는 기능은 튜토리얼 Timer.3 과 똑같다고 보시면 됩니다.

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
```

앞선 튜토리얼 프로그램들 처럼 자유 함수(free function) print 를 콜백함수로 정의하는 것 대신에, printer 라고 불리는 클래스를 정의합니다.

```
class printer
{
public:
```

이 클래스의 생성자는 io_service 의 참조값을 받아서 timer_ 멤버를 초기화할 때 사용합니다. 이제는 프로그램을 종료하기 위한 카운터 역시 클래스의 멤버입니다.

```
printer(boost::asio::io_service& io)
    : timer_(io, boost::posix_time::seconds(1)),
      count_(0)
  {
```

boost::bind()함수는 자유 함수에서 잘 작동하는 것 처럼 클래스 멤버 함수 에서도 역시 잘 작동합니다. 모든 비 정적(non-static)멤버함수들은 암시적으로 this 매개변수를 가지고 있기 때문에, 함수에 this 를 함께 바인딩 해야합니다. 튜토리얼 Timer.3 에서처럼, boost::bind()가 콜백 핸들러(지금은 멤버함수죠)를, 마치 void(const boost::system::error_code&)를 가지고있는 것 처럼 불러올 수 있는 함수객체로 변환 시킵니다.

print 멤버함수는 매개변수로 에러 객체를 받지 않기 때문에, 여기에서는 boost::asio::placeholders::error placeholder 가 명시되지 않았다는것에 주의해 주십시오.

```
    timer_.async_wait(boost::bind(&printer::print, this));
  }
```

클래스의 소멸자에서 카운터의 최종 값을 출력해 줍니다.

```
~printer()
  {
    std::cout << "Final count is " << count_ << "\n";
  }
```

print 멤버 함수는 튜토리얼 Timer.3 의 print 함수와 매우 비슷합니다. 매개변수로 받은 카운터 대신에, 멤버변수 카운터를 쓰고있다는 것만 빼고는요.

```
void print()
  {
    if (count_ < 5)
    {
      std::cout << count_ << "\n";
      ++count_;

      timer_.expires_at(timer_.expires_at() + boost::posix_time::seconds(1));
      timer_.async_wait(boost::bind(&printer::print, this));
    }
  }

private:
  boost::asio::deadline_timer timer_;
  int count_;
};
```

메인 함수는, 전보다 훨씬 단순해졌습니다. io_service 를 수행하기 전에 로컬 printer 객체만 선언해주면 되니까요.

```
int main()
{
  boost::asio::io_service io;
  printer p(io);
  io.run();

  return 0;
}
```

TIMER.5 - 멀티 스레드 프로그램에서 핸들러를 동기화하기

이 튜토리얼은 멀티 스레드 프로그램에서 콜백 핸들러를 동기화 하기 위해 the boost::asio::strand 클래스를 사용하는 것을 설명합니다.

앞선 네개의 튜토리얼들은 boost::asio::io_service::run() 함수의 호출을 하나의 쓰레드로 제한함으로써 핸들러의 동기화 이슈를 피해왔습니다. 여러분들도 아시다시피, asio 라이브러리는 콜백 핸들러가 현재 boost::asio::io_service::run()를 호출한 스레드에서만 호출되는 것을 보장해줍니다. 따라서, 하나의 쓰레드에서만 boost::asio::io_service::run()을 호출하는 것은 콜백핸들러가 동시에 실행되지 않는다는 것을 보장합니다.

asio 를 이용한 응용프로그램 개발을 시작할 때 싱글 스레드로 접근하는 것은 보통 최선의 방법입니다. 하지만, 그것이 서버 같은 프로그램에서 사용될 때에는 아래와 같은 단점을 가지고 있습니다.

- 핸들러가 완료되는데 오랜시간이 걸린다면, 응답시간이 느려집니다.
- 멀티 프로세서 시스템에서는 아무것도 할 수가 없겠죠.

실제로 여러분이 이러한 한계를 느끼신다면, boost::asio::io_service::run()의 스레드 풀을 가지고있는 것이 대안이 될 수 있습니다.

하지만, 이것을 동시에 실행하는 것을 허락함으로써, 핸들러가, 공유되어있는 리소스나, 스레드에 안전하지 않은 리소스를 접근할수도 있을 때, 동기화 해 줄 방법이 필요합니다.

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
```

앞선 튜토리얼과 비슷하게, printer 라는 클래스를 정의하는것으로 시작해봅니다. 이 클래스는 앞의 튜토리얼을 두개의 타이머가 병렬적으로 실행되도록 확장한 것입니다.

```
class printer
{
public:
```

추가로, boost::asio::deadline_timer 멤버 한쌍과, boost::asio::strand 타입의 trand_멤버를 초기화합니다.

boost::asio::strand 는 자신을 통해 디스패치되는 핸들러에게, 실행중인 핸들러가 완료되어야만 다음 핸들러가 시작될 수 있도록 하는 것을 보장해줍니다. 이것은 boost::asio::io_service::run()가 몇 개의 스레드에서 실행되었던지 상관없습니다. 물론, 핸들러는 같은 boost::asio::strand 를 통해 디스패치되지 않은 다른 핸들러와함께 동시적으로 실행중 이어야 합니다.
아니면 다른 boost::asio::strand 객체를 통해 디스패치 되던가요.

```
printer(boost::asio::io_service& io)
    : strand_(io),
      timer1_(io, boost::posix_time::seconds(1)),
      timer2_(io, boost::posix_time::seconds(1)),
      count_(0)
  {
```

비동기적인 오퍼레이션을 초기화 할 때, 각각의 콜백 핸들러는 boost::asio::strand 객체에 의해 감싸집니다. boost::asio::strand::wrap()함수는 boost::asio::strand 를 통해 포함하게된 핸들러를 자동으로 디스패치해서, 새로운 핸들러를 반환합니다.
감싸진 핸들러들을 같은 boost::asio::strand 객체에서 사용함으로써, 우리는 그 핸들러들이 동시에 실행되지 않는 것을 보장할수 있는 것 입니다.

```
    timer1_.async_wait(strand_.wrap(boost::bind(&printer::print1, this)));
    timer2_.async_wait(strand_.wrap(boost::bind(&printer::print2, this)));
  }

  ~printer()
  {
    std::cout << "Final count is " << count_ << "\n";
  }
```

멀티 스레드 프로그램에서 비동기적인 오퍼레이션을하는 핸들러들은 공유 리소스를 접근하려 할 때, 동기화됩니다. 이 튜토리얼에서는, 핸들러들 (print1 과 print2)에게 공유되는 리소소는 std::cout 과 count_ 데이터멤버입니다.

```
void print1()
  {
    if (count_ < 10)
    {
      std::cout << "Timer 1: " << count_ << "\n";
      ++count_;

      timer1_.expires_at(timer1_.expires_at() + boost::posix_time::seconds(1));
      timer1_.async_wait(strand_.wrap(boost::bind(&printer::print1, this)));
    }
  }

  void print2()
  {
    if (count_ < 10)
    {
      std::cout << "Timer 2: " << count_ << "\n";
      ++count_;

      timer2_.expires_at(timer2_.expires_at() + boost::posix_time::seconds(1));
      timer2_.async_wait(strand_.wrap(boost::bind(&printer::print2, this)));
    }
  }

private:
  boost::asio::strand strand_;
  boost::asio::deadline_timer timer1_;
  boost::asio::deadline_timer timer2_;
  int count_;
};
```

이제 메인함수에서 두개의 스레드에서 boost::asio::io_service::run()가 호출되도록 해 봅시다.(메인스레드와 추가된 스레드) 이것은 boost::thread 객체를 통해서 이루어집니다.

싱글스레드에서의 호출과 마찬가지로, boost::asio::io_service::run()의 호출은 할일이 남아있는동안 동시적인 호출이 계속 일어날겁니다. 백그라운드 스레드는 비동기적인 오퍼레이션이 완료되기 전까지 끝나지 않겠지요.

```
int main()
{
  boost::asio::io_service io;
  printer p(io);
  boost::thread t(boost::bind(&boost::asio::io_service::run, &io));
  io.run();
  t.join();

  return 0;
}
```

DAYTIME.1 - 동기적인 TCP DAYTIME CLIENT

이 튜토리얼 프로그램은 asio 를 가지고 어떻게 TCP 클라이언트 어플리케이션을 구현하는가를 보여줍니다.

필요한 헤더파일을 인클루드 하는것부터 시작하겠습니다.

```
#include <iostream>
#include <boost/array.hpp>
#include <boost/asio.hpp>
```

이 어플리케이션의 목적은 daytime 서비스에 접속하는것입니다. 그러니 서버를 지정해주어야 하겠지요.

```
using boost::asio::ip::tcp;

int main(int argc, char* argv[])
{
  try
  {
    if (argc != 2)
    {
      std::cerr << "Usage: client <host>" << std::endl;
      return 1;
    }
```

asio 를 사용하는 모든 프로그램은 적어도 하나의 boost::asio::io_service 객체를 가지고있어야 합니다.

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

boost::asio::ip::tcp::resolver 객체를 사용하기 위해서, 어플리케이션의 매개변수로 정의된 서버의 이름 TCP 단말점(endpoint)으로 변환해야 합니다.

```
    tcp::resolver resolver(io_service);
```

결정자(resolver)는 쿼리 오브젝트를 받아서 단말점의 목록으로 변환합니다.
여기서는 argv[1]에서 정의된 서버의 이름과 "daytime"이라는 서비스의 이름으로 쿼리를 생성합니다.

```
    tcp::resolver::query query(argv[1], "daytime");
```

단말점의 목록은 boost::asio::ip::tcp::resolver::iterator 타입의 반복자(itorator)를 이용해서 반환됩니다. boost::asio::ip::tcp::resolver::iterator 객체의 기본생성자는 리스트의 끝을 가리키는 반복자를 통해 생성됩니다.

```
    tcp::resolver::iterator endpoint_iterator = resolver.resolve(query);
    tcp::resolver::iterator end;
```

이제, 소켓을 생성하고 연결해 봅시다. 단말점의 목록은 IPv4 와 IPv6 에 상관없이 단말점을 포함하고있습니다. 그래서, 어느것이 맞는것인지 찾아 보아야 합니다. 이것은 클라이언트 프로그램이 정의된 IP 버전에 독립적이게 만들어줍니다.

```
    tcp::socket socket(io_service);
    boost::system::error_code error = boost::asio::error::host_not_found;
    while (error && endpoint_iterator != end)
    {
      socket.close();
      socket.connect(*endpoint_iterator++, error);
    }
    if (error)
      throw boost::system::system_error(error);
```

자, 드디어 커넥션이 열렸습니다. 이제 우리가 해야할 일은 daytime 서비스로부터 응답을 읽는 것 뿐입니다.

우리는 받은 데이터를 저장하기위해 boost::array 를 사용합니다. boost::asio::buffer() 함수는 버퍼 오버런을 방지하기 위해 자동으로 배열의 크기를 결정합니다. boost::array 대신에 char[]이나 std::vector 를 사용할 수도 있습니다.

```
for (;;)
    {
      boost::array<char, 128> buf;
      boost::system::error_code error;

      size_t len = socket.read_some(boost::asio::buffer(buf), error);
```

서버가 커넥션을 닫으면, boost::asio::ip::tcp::socket::read_some() 함수는 boost::asio::error::eof 에러와 함께 종료됩니다. 이를 통해 루프를 빠져나갈 시기를 알 수 있죠.

```
      if (error == boost::asio::error::eof)
        break; // Connection closed cleanly by peer.
      else if (error)
        throw boost::system::system_error(error); // Some other error.

      std::cout.write(buf.data(), len);
    }
```

마지막으로, 혹시 발생했을지도 모르는 나머지 예외를 처리해줍니다.

```
  }
  catch (std::exception& e)
  {
```

```
    std::cerr << e.what() << std::endl;
  }
```

DAYTIME.2 - 동기적인 TCP DAYTIME SERVER

이 튜토리얼 프로그램은 **asio** 를 사용해서 어떻게 **TCP** 서버 어플리케이션을 만드는지를 보여줍니다.

```
#include <ctime>
#include <iostream>
#include <string>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;
```

일단 클라이언트에게 보내줄 문자열을 만들어낼 **make_daytime_string()** 함수를 정의합니다. 이 함수는 앞으로 만들어질 모든 **daytime** 서버에서 재사용 될 것입니다.

```
std::string make_daytime_string()
{
  using namespace std; // For time_t, time and ctime;
  time_t now = time(0);
  return ctime(&now);
}

int main()
{
  try
  {
    boost::asio::io_service io_service;
```

새로운 커넥션을 **listen** 하기 위해 **boost::asio::ip::tcp::acceptor** 객체가 필요합니다. **13** 번 **TCP** 포트에 **IPv4** 로 초기화해서 **listen** 해보겠습니다.

```
tcp::acceptor acceptor(io_service, tcp::endpoint(tcp::v4(), 13));
```

이 서버는 반복적인 서버입니다.**(itorative server).** 즉, 한번에 하나씩의 커넥션을 처리한다는 것이지요. 클라이언트와의 커넥션을 표현할 소켓을 하나 만들고, 커넥션을 기다리게됩니다.

```
for (;;)
    {
      tcp::socket socket(io_service);
      acceptor.accept(socket);
```

클라이언트가 우리의 서비스에 접근하고 있습니다. 현재시간을 결정하고, 그에 대한 정보를 클라이언트에게 전송합니다.

```
      std::string message = make_daytime_string();

      boost::system::error_code ignored_error;
      boost::asio::write(socket, boost::asio::buffer(message),
          boost::asio::transfer_all(), ignored_error);
    }
  }
```

마지막으로 예외처리를 해줍니다.

```
catch (std::exception& e)
  {
    std::cerr << e.what() << std::endl;
  }

  return 0;
}
```

DAYTIME.3 - 비동기적인 TCP DAYTIME SERVER

메인함수

```
int main()
{
  try
  {
```

우선 클라이언트로부터 들어오는 접속을 억셉트하기위해서 서버 객체를 생성해야 합니다. boost::asio::io_service 객체는 소켓과같은 I/O 서비스를 제공해주기 때문에 서버객체로 사용될 수 있지요.

```
    boost::asio::io_service io_service;
    tcp_server server(io_service);
```

여러분의 의도대로 비동기적인 오퍼레이션을 할수 있게, boost::asio::io_service 객체를 실행해 봅시다.

```
    io_service.run();
  }
  catch (std::exception& e)
  {
    std::cerr << e.what() << std::endl;
  }

  return 0;
}
```

tcp_server 클래스

```
class tcp_server
{
public:
```

생성자에서는 **listen** 을 위한 억셉터를 **TCP 13** 번 포트로 초기화 해 줍니다.

```
tcp_server(boost::asio::io_service& io_service)
    : acceptor_(io_service, tcp::endpoint(tcp::v4(), 13))
  {
    start_accept();
  }

private:
```

start_accept()함수는 소켓을 하나 생성하고, 새로운 커넥션을 기다리는 비동기적인 억셉트를 수행합니다.

```
void start_accept()
  {
    tcp_connection::pointer new_connection =
      tcp_connection::create(acceptor_.io_service());

    acceptor_.async_accept(new_connection->socket(),
        boost::bind(&tcp_server::handle_accept, this, new_connection,
          boost::asio::placeholders::error));
  }
```

handle_accept()함수는 비동기적인 억셉트의 수행이 **start_accept()**함수의 종료로 인해서 초기화될 때 호출됩니다.
이 함수는 클라이언트의 요청에 응답하고, 다음 억셉트의 수행을 위해 **start_accept()**함수를 호출합니다.

```
void handle_accept(tcp_connection::pointer new_connection,
      const boost::system::error_code& error)
  {
    if (!error)
    {
      new_connection->start();
      start_accept();
    }
  }
```

tcp_connection 클래스

tcp_connection 객체가 참조되는 동안에는 가능한한 객체를 살려두기 위해서, 앞으로는 **shared_ptr** 과 **enable_shared_from_this** 를 사용 하겠습니다.

```
class tcp_connection
  : public boost::enable_shared_from_this<tcp_connection>
{
public:
  typedef boost::shared_ptr<tcp_connection> pointer;

  static pointer create(boost::asio::io_service& io_service)
  {
    return pointer(new tcp_connection(io_service));
  }

  tcp::socket& socket()
  {
    return socket_;
  }
```

start()함수에서, 클라이언트에게 데이터를 제공하기위해 boost::asio::async_write() 함수를 호출합니다. 명심하세요. 우리는 boost::asio::async_write()를 사용하고 있는것이지, boost::asio::ip::tcp::socket::async_write_some()을 사용하고 있는 것이 아닙니다. 전체 블록이 전송되는 것을 보장하기 위해서 이지요.

```
void start()
  {
```

보내질 데이터들은 비동기적인 오퍼레이션이 완료되는 동안 훼손되지 않도록 message_멤버변수에 저장됩니다.

```
message_ = make_daytime_string();
```

boost::bind()를 사용해서 비동기적인 오퍼레이션이 초기화될 때, 여러분은 반드시 핸들러의 매개변수 목록에 가장 적합한 인자들을 지정해 주어야합니다.
이 프로그램에서는 인자 지정자(placeholder) 두 가지가 모두 (boost::asio::placeholders ::error 와 boost::asio::placeholders:: bytes_transferred) 잠재적으로 제거할 수도 있습니다. 왜냐하면, handel_write()에서 사용 되고있지 않으니까요.

```
boost::asio::async_write(socket_, boost::asio::buffer(message_),
       boost::bind(&tcp_connection::handle_write, shared_from_this(),
         boost::asio::placeholders::error,
         boost::asio::placeholders::bytes_transferred));
```

클라이언트와의 커넥션에 대한 나머지 일에대한 책임은 handle_write()함수에게 있습니다.

```
}

private:
  tcp_connection(boost::asio::io_service& io_service)
    : socket_(io_service)
  {
  }
```

```
  void handle_write(const boost::system::error_code& /*error*/,
      size_t /*bytes_transferred*/)
  {
  }

  tcp::socket socket_;
  std::string message_;
};
```

사용되지 않는 핸들러 매개변수 제거하기

아마 여러분은 error 와 bytes_transferred 매개변수도가 handle_write() 함수의 내부에서 사용되지 않았다는 것을 알고 계실 것입니다. 매개변수가 필요없다면, 다음과 같이 제거해 버리는 것도 가능하겠군요.

```
void handle_write()
  {
  }
```

boost::asio::async_write()함수의 호출은, 보통 다음과 같이 바꿔서 호출할 수도 있습니다.

```
boost::asio::async_write(socket_, boost::asio::buffer(message_),
      boost::bind(&tcp_connection::handle_write, shared_from_this()));
```

DAYTIME.4 - 비동기적인 UDP DAYTIME CLIENT

이 튜토리얼 프로그램은 어떻게 asio 를 통해서 UDP 클라이언트 응용프로그램을 구현하는지를 보여줍니다.

```
#include <iostream>
#include <boost/array.hpp>
#include <boost/asio.hpp>

using boost::asio::ip::udp;
```

이 응용프로그램은 본질적으로 TCP daytime 클라이언트 프로그램과 동일하게 시작합니다.

```
int main(int argc, char* argv[])
{
  try
  {
    if (argc != 2)
    {
      std::cerr << "Usage: client <host>" << std::endl;
      return 1;
    }

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

우리는 boost::asio::ip::udp::resolver 객체를 통해서 호스트와 서비스 이름에 기반한 올바른 원격 종말점을 찾게 됩니다. 쿼리는 boost::asio::ip::udp::v4()인자에 의해서 오직 IPv4 에 제한된 값을 반환합니다.

```
    udp::resolver resolver(io_service);
    udp::resolver::query query(udp::v4(), argv[1], "daytime");
```

boost::asio::ip::udp::resolver::resolve()함수가 실패하지 않는다면, 적어도 하나의 종말점을 리스트중에서 반환할 것 이라는 것이 보장됩니다. 이것은 반환된 값에 대한 직접적인 역참조가 것이 안전하다는 것을 의미합니다.

```
    udp::endpoint receiver_endpoint = *resolver.resolve(query);
```

UDP 는 데이터그램 기반이므로, 스트림소켓을 사용하지 않습니다.
boost::asio::ip::udp::socket 을 생성하고 원격 종말점에 접근하는 것을 초기화 합니다.

```
    udp::socket socket(io_service);
    socket.open(udp::v4());

    boost::array<char, 1> send_buf  = { 0 };
    socket.send_to(boost::asio::buffer(send_buf), receiver_endpoint);
```

이제 우리는 서버가 우리에게 응답을 하던지와 상관없이 억셉트가 준비되어야 합니다. 서버의 응답을 받을 우리쪽의 종말점은 boost::asio::ip::udp::socket::receive_from()으로 통해서 초기화됩니다.

```
boost::array<char, 128> recv_buf;
    udp::endpoint sender_endpoint;
    size_t len = socket.receive_from(
        boost::asio::buffer(recv_buf), sender_endpoint);

    std::cout.write(recv_buf.data(), len);
  }
```

마지막으로, 있을지도 모르는 예외를 처리해줍니다.

Finally, handle any exceptions that may have been thrown.

```
  catch (std::exception& e)
  {
    std::cerr << e.what() << std::endl;
  }

  return 0;
}
```

DAYTIME.5 - 비동기적인 UDP DAYTIME SERVER

이 튜토리얼 프로그램은 asio 를 가지고 어떻게 UDP 로 서버 응용프로그램을 구현하는지를 보여줍니다.

```
int main()
{
  try
  {
    boost::asio::io_service io_service;
```

클라이언트의 요청에 응답하기 위한 boost::asio::ip::udp::socket 객체를 UDP 포트 13 번으로 생성해줍니다.

```
    udp::socket socket(io_service, udp::endpoint(udp::v4(), 13));
```

이제, 우리와의 접속을 초기화 하려는 클라이언트를 기다립시다. remote_endpoint 객체는 boost::asio::ip::udp::socket::receive_from()에 의해서 등록됩니다.(populated)

```
    for (;;)
    {
      boost::array<char, 1> recv_buf;
      udp::endpoint remote_endpoint;
      boost::system::error_code error;
      socket.receive_from(boost::asio::buffer(recv_buf),
          remote_endpoint, 0, error);

      if (error && error != boost::asio::error::message_size)
        throw boost::system::system_error(error);
```

클라이언트에게 보내주어야 할것을 결정해줍시다.

```
     std::string message = make_daytime_string();
```

remote_endpoint 를 통해서 메시지를 전송하십시오.

```
    boost::system::error_code ignored_error;
      socket.send_to(boost::asio::buffer(message),
          remote_endpoint, 0, ignored_error);
    }
  }
```

마지막으로 예외를 처리합니다.

```
  catch (std::exception& e)
  {
    std::cerr << e.what() << std::endl;
```

```
  }

  return 0;
}
```

DAYTIME.6 - 비동기적인 UDP DAYTIME SERVER

메인함수

```
int main()
{
  try
  {
```

클라이언트의 접속요청에 억셉트 하기위한 서버 객체를 생성하고, boost::asio::io_service 객체를 실행합니다.

```
    boost::asio::io_service io_service;
    udp_server server(io_service);
    io_service.run();
  }
  catch (std::exception& e)
  {
    std::cerr << e.what() << std::endl;
  }

  return 0;
}
```

udp_server 클래스

```
class udp_server
{
public:
```

생성자는 UDP 포트 13 번에서 리슨 하기 위한 소켓을 생성합니다.

```
    udp_server(boost::asio::io_service& io_service)
    : socket_(io_service, udp::endpoint(udp::v4(), 13))
  {
    start_receive();
  }

private:
  void start_receive()
  {
```

boost::asio::ip::udp::socket::async_receive_from()함수는 응용프로그램이 새로운 요청에 대한 응답을 백그라운드에서 리슨할 수 있게 해줍니다.
요청을 받으면, boost::asio::io_service 객체는 handle_receive()함수를 두개의 인자로

호출합니다. 함수의 수행이 성공했는지의 여부를 알려주는 boost::system::error_code 타입과, 몇바이트를 받았는지를 알려주는 size_t 타입의 byte_transfer 가 그것이죠.

```
    socket_.async_receive_from(
        boost::asio::buffer(recv_buffer_), remote_endpoint_,
        boost::bind(&udp_server::handle_receive, this,
          boost::asio::placeholders::error,
          boost::asio::placeholders::bytes_transferred));
  }
```

handle_receive()함수는 클라이언트의 요청에 서비스를 해줍니다.

```
  void handle_receive(const boost::system::error_code& error,
      std::size_t /*bytes_transferred*/)
  {
```

error 라는 매개변수는 비동기적 오퍼레이션의 결과를 가지고 있습니다. 우리는 클라이언트의 요청을 받기위해서 단지 1 바이트의 recv_buffer_만을 제공했기 때문에, boost::asio::io_service 객체는 클라이언트가 뭔가 더 큰 것을 보내면 에러를 반환 하게 됩니다. 이러한 에러의 발생은 무시할수 있습니다.

```
if (!error || error == boost::asio::error::message_size)
    {
```

무엇을 보내려고 하는지를 결정해주십시오.

```
        boost::shared_ptr<std::string> message(
          new std::string(make_daytime_string()));
```

이제, 클라이언트에세 데이터를 제공하기 위해서 boost::asio::ip::udp::socket::async_send_to() 함수를 호출합니다.

```
        socket_.async_send_to(boost::asio::buffer(*message), remote_endpoint_,
          boost::bind(&udp_server::handle_send, this, message,
            boost::asio::placeholders::error,
            boost::asio::placeholders::bytes_transferred));
```

boost::bind()를 통해서 비동기적인 오퍼레이션이 초기화 되면, 핸들러의 매개변수 목록에 맞는 인자를 명세해 주어야 합니다. 이 프로그램에서는 boost::asio::placeholders::error 와 boost::asio::placeholders::bytes_transferred 모두 잠재적으로 제거될 수 있습니다.

다음 클라이언트의 요청에 대해 리스닝을 시작해주십시오.

```
        start_receive();
```

클라이언트의 요청에 대한 나머지 일에 대한 책임은 handle_send()가 가지고있습니다.

```
    }
  }
```

handle_send()함수는 서비스의 요청이 완료되면 호출됩니다.

```
  void handle_send(boost::shared_ptr<std::string> /*message*/,
      const boost::system::error_code& /*error*/,
      std::size_t /*bytes_transferred*/)
  {
  }

  udp::socket socket_;
  udp::endpoint remote_endpoint_;
  boost::array<char, 1> recv_buffer_;
};
```

DAYTIME.7 - TCP 와 UDP 가 결합된 비동기적 서버

이 튜토리얼 프로그램은 우리가 앞서 만들었던 두개의 비동기적인 서버를 어떻게 하나의 서버 응용 프로그램에서 결합시키는지를 보여줍니다.

메인함수

```
int main()
{
  try
  {
    boost::asio::io_service io_service;
```

일단 TCP 클라이언트와의 연결을 억셉트 하기위한 서버를 생성하는것으로 시작합니다.

```
    tcp_server server1(io_service);
```

UDP 클라이언트와의 연결을 억셉트하는 서버객체도 만들어 주어야 겠지요.

```
    udp_server server2(io_service);
```

우리가 생성해 놓은 두가지 일은 boost::asio::io_service 객체가 처리하게 됩니다.

```
    io_service.run();
  }
  catch (std::exception& e)
  {
    std::cerr << e.what() << std::endl;
  }
```

```
  return 0;
}
```

tcp_connection 과 tcp_server 클래스

아래 두가지 클래스는 Daytime.3 에서 가져왔습니다.

```
class tcp_connection
  : public boost::enable_shared_from_this<tcp_connection>
{
public:
  typedef boost::shared_ptr<tcp_connection> pointer;

  static pointer create(boost::asio::io_service& io_service)
  {
    return pointer(new tcp_connection(io_service));
  }

  tcp::socket& socket()
  {
    return socket_;
  }

  void start()
  {
    message_ = make_daytime_string();

    boost::asio::async_write(socket_, boost::asio::buffer(message_),
        boost::bind(&tcp_connection::handle_write, shared_from_this()));
  }

private:
  tcp_connection(boost::asio::io_service& io_service)
    : socket_(io_service)
  {
  }

  void handle_write()
  {
  }

  tcp::socket socket_;
  std::string message_;
};

class tcp_server
{
public:
  tcp_server(boost::asio::io_service& io_service)
    : acceptor_(io_service, tcp::endpoint(tcp::v4(), 13))
  {
    start_accept();
  }

private:
  void start_accept()
  {
    tcp_connection::pointer new_connection =
      tcp_connection::create(acceptor_.io_service());
```

```
    acceptor_.async_accept(new_connection->socket(),
        boost::bind(&tcp_server::handle_accept, this, new_connection,
          boost::asio::placeholders::error));
  }

  void handle_accept(tcp_connection::pointer new_connection,
      const boost::system::error_code& error)
  {
    if (!error)
    {
      new_connection->start();
      start_accept();
    }
  }

  tcp::acceptor acceptor_;
};
```

udp_server 클래스

비슷하게, 다음의 클래스는 바로 앞의 튜토리얼 스텝에서 가져왔습니다.

```
class udp_server
{
public:
  udp_server(boost::asio::io_service& io_service)
    : socket_(io_service, udp::endpoint(udp::v4(), 13))
  {
    start_receive();
  }

private:
  void start_receive()
  {
    socket_.async_receive_from(
        boost::asio::buffer(recv_buffer_), remote_endpoint_,
        boost::bind(&udp_server::handle_receive, this,
          boost::asio::placeholders::error));
  }

  void handle_receive(const boost::system::error_code& error)
  {
    if (!error || error == boost::asio::error::message_size)
    {
      boost::shared_ptr<std::string> message(
          new std::string(make_daytime_string()));

      socket_.async_send_to(boost::asio::buffer(*message), remote_endpoint_,
          boost::bind(&udp_server::handle_send, this, message));

      start_receive();
    }
  }

  void handle_send(boost::shared_ptr<std::string> /*message*/)
  {
  }

  udp::socket socket_;
  udp::endpoint remote_endpoint_;
```

```
  boost::array<char, 1> recv_buffer_;
};
```

디자인 노트

[이론적인 근거(Rationale)]

Boost.Asio 라이브러리 디자인의 이론적인 근거입니다.

[비동기적인 오퍼레이션]

Boost.Asio 라이브러리의 비동기적인 오퍼레이션 제공은 프로액터 패턴을 통해서 이루어 집니다. 이 디자인 노트에서는 이 접근 방법의 장단점을 살펴봅니다.

[자체적인 메모리 할당]

Boost.Asio 에서 제공하는 자체적인 메모리 할당에 대해서 설명합니다.

[버퍼]

Scatter-Gatter 수행을 위해 사용된 asio 의 버퍼 추상화에 대한 검토입니다.

[왜 EOF 는 에러인가]

어째서 end-of-file 상태가 에러 코드가 될수있는지를 논의합니다.

[행 기반(Line-based) 프로토콜]

Boost.Asio 의 행기반 프로토콜 지원에 대해 간략히 알아봅니다.

[스레드]

Boost.Asio 의 개별 플랫폼에서의 구현에 있어서, 하나 이상의 추가적인 스레드가 비동기적으로 경쟁할 수도 있습니다. 이 디자인노트는 이러한 상황에서의 스레드 사용에 적용된 디자인 규칙에 대해서 논의합니다.

[스트랜드(Strands)]

동시적 프로그래밍을 쉽게해주고 다중 프로세서에서의 확장성을 제공하기위해 Boost.Asio 에서 제공하는 스트랜드(strand) 추상화에 대해서 설명합니다.

[특정 플랫폼에서의 구현]

이 디자인 노트는 기본적인 디멀티플렉싱 방식이나, 내부적으로 생성되는 스레드의 개수, 스레드가 생성될 때의 일 같은, 특정한 플랫폼 마다 다르게 구현된 세부 사항들을 나열합니다.

이론적인 근거 RATIONALE

Boost.Asio 라이브러리의 설계는 네트워킹이 종종 필요한 운영체제 기능에 접근하는 시스템 프로그래밍을 하는 c++ 프로그래머를 위해 의도되었습니다. 특히, 아래에 나오늘 것들을 목표로 합니다.

- **이식성(Portability).** 라이브러리는 일반적으로 사용되는 운영체제에서 동일하게 작동 해야 합니다.

o **확장성(Scalability).** 라이브러리는 수백에서 수천만의 동시접속을 지원하는 네트워크 응용프로그램의 개발을 지원해야 합니다. 사실은 장려하고 있습니다. 라이브러리는 각각의 운영체제가 지원하는 최상의 메커니즘을 지원하도록 구현되어야 합니다.

o **효율성(Efficiency).** 라이브러리는 sctter-gagher I/O 같은 메커니즘을 지원하고, 데이터의 복사를 최소화 할수 있는 프로토콜의 구현을 지원해야 합니다.

o **버클리 소켓 모델(Model Berkeley sockets).** 버클리 소켓 API 는 많은 문헌에서 언급되고, 널리 구현되고 이해되어 있습니다. 다른 프로그래밍 언어에서도 네트워킹 API 를 위해 종종 비슷한 인터페이스를 사용합니다.

o **사용하기 쉽게(Ease of use).** 프레임워크 보다는 툴킷을 이용하는 것이 새로운 사용자에게 진입 장벽을 낮추어 줍니다. 이것은 초기에 기본 규칙을 공부하는 데 드는 선행투자를 최소화 시켜 줍니다. 그리고, 라이브러리의 사용자는 단지 명세된 함수를 사용하는 법만 익히면 됩니다.

o **이후의 추상화에 대한 기초(Basis for further abstraction).** 라이브러리는 더높은 추상화 단계의 다른 라이브러리의 개발을 허락해야 합니다. 예를 들어 HTTP 같은 일반적으로 사용되는 프로토콜의 구현 같은 것 말입니다.

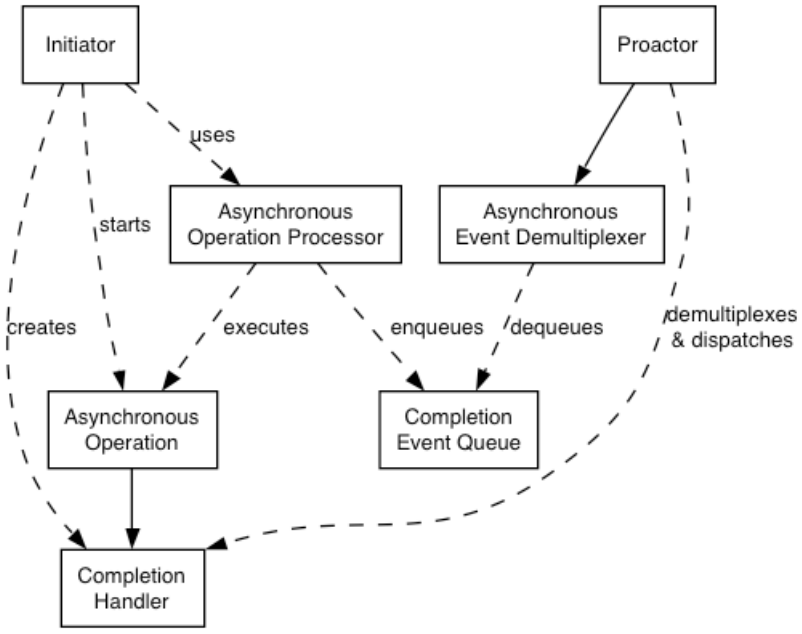
또한 현재의 Boost.Asio 의 구체화된 결과물은 첫째로 다른 운영체제의 파일과 같은 자원의 포함을 확장하는 비동기적인 입출력의 개념을 가진 네트워킹에 초점을 두고있습니다.

비동기적인 오퍼레이션

제안된 라이브러리는 동기적인면과 비동기적인면의 수행을 모두 제공합니다. 비동기적인 오퍼레이션의 지원은 프로액터 디자인 패턴에 기반합니다. 동기적으로만 사용되는 리액터 접근방법과 비교 할 때 보이는 장단점은 아래에 소개됩니다.

프로액터와 Boost.Asio

플랫폼 종속적인 세부사항은 제외하고, Boost.Asio 에서 어떻게 프로액터 디자인 패턴이 구현되는지 검토해 봅시다.



프로액터 디자인 패턴(POSA2)

–비동기적 오퍼레이션(Asynchronous Operation)

소켓에서의 비동기적인 읽기 쓰기와 같은,비동기적으로 실행되는 오퍼레이션을 정의합니다.

– 비동기적 오퍼레이션 처리기(Asynchronous Operation Processor)

오퍼레이션이 완료되었을 때, 비동기적인 오퍼레이션을 실행하거나, 완료 이벤트 큐에 이벤트를 넣습니다. 높은 차원의 관점에서, asio::stream_socket_service 같은 서비스들은 비동기적 오퍼레이션 처리기입니다.

– 완료 이벤트 큐(Completion Event Queue)

비동기적인 이벤트의 디멀티플렉서에의해 디큐(dequeue)될때까지 완료이벤트를 저장합니다.

– 완료 핸들러(Completion Handler)

비동기적인 오퍼레이션의 결과를 처리합니다. 이것은 종종 boost::bind()를 통해 함수 객체로 생성됩니다.

–비동기적인 이벤트의 디멀티플렉서(Asynchronous Event Demultiplexer)

완료 이벤트 큐에서 발생하는 이벤트를 블록대기하고, 완료된 이벤트를 자신의 호출자에게 반환합니다.

–프로액터(Proactor)

이벤트를 큐에서 꺼내기위해(dequeue) 비동기적인 이벤트의 디멀티플렉서를 호출하고, 그 이벤트와 연관된 완료 핸들러를 디스패치합니다.(이것은 함수객체의 실행을 의미합니다.) 이러한 추상화된 개념은 boost::asio_io_service 로 표현되어 있습니다.

–개시자(Initiator)

응용프로그램에서 명시된, 비동기적인 오퍼레이션을 시작하는 코드입니다. 개시자는 비동기적인 오퍼레이션 처리자와, asio::stream_socket_service 와 같은, 서비스에게 대리자 (delegates)를 반환하는 asio::basic_stream_socket 등의 상위레벨의 인터페이스를 통해서 상호작용합니다.

<u>리액터를 이용한 구현</u>

많은 플랫폼에서 Boost.Asio 는 select 나 epoll, kqueue 같은 리액터의 방식으로 프로액터를 구현합니다. 이러한 구현 방식은 아래와 같은 부분에서 프로액터 패턴과 일치합니다.

– 비동기적인 오퍼레이션 처리기

리액터는 select,epoll,또는 kqueue 를 통해서 구현됩니다. 리액터가 자원이 오퍼레이션을 수행할 준비가 되었다고 알려주면, 처리기는 비동기적인 오퍼레이션을 실행하고 연관된 완료 핸들러를 완료 이벤트 큐에 집어넣습니다.

– 완료 이벤트 큐

완료 핸들러(함수객체 등)의 연결 리스트 입니다.

– 비동기적인 이벤트의 디멀티플렉서

이것은 완료 핸들라가 완료 입벤트 큐에서 사용가능할 때 까지, 이벤트나 상태변수를 기다리는것으로 구현됩니다.

Windows Overlapped I/O 를 통한 구현

Windows NT, 2000, XP 에서, Boost.Asio 는 프로액터 디자인 패턴의 효율적인 구현에 있어서의 overlapped I/O 의 장점을 취합니다. 이 구현 방법은 아래와 같은 부분에서 프로액터 패턴과 일치합니다.

– 비동기적인 오퍼레이션 처리기

이것은 운영체제에 의해 구현되어 있습니다. 오퍼레이션은 AcceptEx 와 같은 오버랩드 함수를 호출함으로써 초기화됩니다.

– 완료 이벤트 큐

이것은 운영체제에 의해 구현되어있고 IOCP(I/O Completion port)와 연관되어 있습니다. 각각의 asio_io_service 인스턴스를 위한 한 개씩의 IOCP 가 존재합니다.

– 비동기적인 이벤트의 디멀티플렉서

큐에서 이벤트와 그들과 연결되어있는 완료 핸들러를 꺼내기위해서 boost.Asio 에의해 호출됩니다.

장점

– 이식성(Portability).

많은 운영체제는 높은 성능의 네트워크 응용프로그램 개발을 위한 준비된 옵션으로써, 자체적인 비동기 입출력 API 를 제공합니다. 제안된 라이브러리는 아마도 운영체제 자체의 비동기 입출력의 방식으로 구현될것입니다. 그러나, 만약 자체적인 지원이 되지 않는다면, 라이프로리는 또한 POSIX 의 select()같은 리액터 패턴 같은 비동기적인 이벤트의 디멀티플렉서를 사용하여 구현될것입니다.

– 동시성으로부터 스레딩을 분리(Decoupling threading from concurrency).

오랜기간, 오퍼레이션은 응용프로그램의 구현을 대신해서 비동기적으로 수행됩니다. 따라서 응용프로그램은 동시성을 높이기 위해 많은 스레드를 생성해낼 필요가 없습니다.

– 성능과 확장성(Performance and scalability).

연결마다 스레드를 만드는 구현 전략은(동기적인 접근이 필요함) CPU 간의 데이터 이동과 동기화에 따른 컨택스트 스위칭에 의해서, 시스템의 성능을 떨어뜨릴 수 있습니다.
비동기적인 수행을 하면, 운영체제 스레드의 개수를 최소화 하고(일반적으로 제한된 자원), 처리될 이벤트를 가지고있는 컨트롤의 논리적인 스레드만을 활성화 해서, 컨텍스트 스위칭의 비용을 피할 수 있습니다.

– 단순화 된 응용프로그램에서의 동기화 (Simplified application synchronisation).

비동기적인 오퍼레이션 완료 핸들러는 싱글 스레드 환경에서 있는 것 처럼 작성 되어질 수 있습니다. 그리고, 응용프로그램 로직도, 동기화에 대한 염려가 거의없이 개발될 수 있습니다.

– 함수의 합성(Function composition).

함수의 합성은 특정 포맷의 메시지를 보내는 것 같은 상위 레벨의 오퍼레이션을 제공하는 함수의 구현에 적용됩니다. 각각의 함수는 read 나 write 같은 하위 레벨 함수들을 호출하는 방식으로 구현되어있습니다.

예를 들어, 어떤 프로토콜이 있다고 가정해 봅시다. 이 프로토콜은 고정길이의 헤더와, 가변길이의 바디를 가지는 메시지 이고, 바디의 길이는 헤더에 정의되어있습니다. 이 가설에 근거한 read_message 오퍼레이션은 헤더(이 길이는 한번만 알면되겠죠)와 바디를 위한 두개의 하위레벨 read 로 이루어질 것입니다.

비동기적인 모델에서 함수를 합성하기 위해, 비동기적인 오퍼레이션은 서로 연결될 수 있습니다. 다시말해서, 하나의 오퍼레이션을 위한 완료 핸들러가 다음것을 초기화할 수 있다는거지요. 체인의 첫번째것만 호출함으로써, 상위 레벨의 오퍼레이션이 비동기적인 오퍼레이션 호출의 체엔이라는 사실을, 호출자가 굳이 신경쓰지 않아도 되도록 캡슐화 될 수 있습니다.

이런방법으로 새로운 오퍼레이션을 합성할수 있다는 것은 명세된 프로토콜을 지원하는 함수 같은, 네트워킹 라이브러리 위의 추상화된 상위 레벨의 개발을 단순화합니다.

단점

– 프로그램 복잡도(Program complexity).

비동기적인 메커니즘을 사용한 응용프로그램 개발은, 오퍼레이션의 초기화와 완료사이의 시공간적 분리로 인해 더욱 어렵습니다. 응용프로그램은 아마도 역전된 제어의 흐름(inverted flow of control)때문에 디버그하기도 힘들것입니다.

– 메모리 사용(Memory usage).

버퍼 공간은 읽고 쓰는 오퍼레이션을 위해서 계속적으로 무기한 할당되어야 합니다. 그리고, 분리된 버퍼는 각각의 동시적인 오퍼레이션을위해 필요합니다. 반면에 리액터 패턴은 소켓이 읽고 쓸 준비를 하는 동안에는 버퍼공간을 필요로하지 않습니다.

참고문헌

[POSA2] D. Schmidt et al, *Pattern Oriented Software Architecture, Volume 2*. Wiley, 2000.

자체적인 메모리 할당

많은 비동기적인 오퍼레이션은 그과 관련된 상태를 저장할 객체를 할당 해야 합니다. 예를들어, Win32 구현에서는 Win32 Api 함수를 전달하기 위한 OVERLAPPED 구동 객체가 필요합니다.

게다가, 프로그램은 일반적으로 쉽게 구별할 수 있는 비동기적인 오퍼레이션의 체인을 포함합니다. half duplex 프로토콜 구현(예. HTTP 서버)은 클라이언트 당 하나의 오퍼레이션 체인을 가지게 됩니다. full duplex 프로토콜은 병렬적으로 실행되는 두개의 체인을 가지게 될것입니다. 프로그램을 체인안의 모든 비동기적인 오퍼레이션을 위해 메모리를 재사용 할수 있도록 이 지식을 활용할 수 있어야 합니다.

사용자 정의 핸들러 객체 h 의 복사본이 있다고 가정합시다. 만약 구현상 이 핸들러와 연관된 메모리의 할당이 필요하다면 다음과 같은 코드를 실행하겠지요.

```
void* pointer = asio_handler_allocate(size, &h);
```

비슷하게, 메모리의 해제는 다음과같이 실행합니다.

```
asio_handler_deallocate(pointer, size, &h);
```

이 함수들은 인자에 따라서 검색을 사용하게 되어 있습니다. 이 구현은 asio namesapce 안의 앞에서 말한 함수들의 기본적인 구현을 제공합니다.

```
void* asio_handler_allocate(size_t, ...);
void asio_handler_deallocate(void*, size_t, ...);
```

이들은 ::operator new()와 ::operator delete()의 방식으로 각각 구현되었습니다.

이러한 구현은 연관된 핸들러가 호출되기 전에 메모리의 해제가 발생할 것이라는 것을 보장합니다. 이 말은, 메모리가 핸들러에의해 시작되는 다른 새로운 비동기적인 오퍼레이션을 위해 재사용될 준비가 되었다는 것을 의미합니다.

이 자체적인 메모리 할당 함수는 아마도 사용자가 생성한 어떠한 스레드에서라도 호출될 것입니다. 라이브러리 함수를 호출하고있는 스레드라면 말이죠. 이러한 구현은 라이브러리를 포함한 비동기적인 오퍼레이션에게, 그 구현이 핸들러를 위해 동시적으로 메모리 할당 함수를 호출하지 않을 것 이라는 것을 보장해 줍니다. 그러한 구현은 적절한 메모리의 경계를 추가하게 됩니다. 메모리 할당 함수가 다른 스레드에서 호출되도록 하는 올바른 메모리의 가시성을 위해서말이죠.

버퍼

효율적인 네트워크 응용프로그램의 개발을 위해, Boost,Asio 는 scatter-gatter 오퍼레이션의 지원을 포함합니다. 이 오퍼레이션은 하나 이상의 버퍼를 포함합니다.(각각의 버퍼는 연속된 메모리 영역을 말합니다.)

- scatter-read 는 여러 개의 버퍼로부터 데이터를 받습니다.
- gatter-write 는 여러 개의 버퍼로 전송합니다.

따라서 우리는 버퍼의 컬렉션을 표현하는 추상화된 무언인가가 필요합니다. Boost.Asio 에서 사용한 접근방법은 하나의 버퍼를 표현하는 하나의 타입을 정의 하는것입니다.(사실은 두개의 타입이죠.) 이것은 컨테이너에 저장될 수 있고, scatter-gather 오퍼레이션에 전달될 수 있습니다.

연속된 메모리 영역으로서의 버퍼는 주소와 바이트들의 크기(size in bytes)로 표현됩니다. 변경가능한 메모리(Boost.Asio 에서는 mutable 이라고 부릅니다.)와 변경 불가능한 메모리(후자는 상수의 저장 공간을 말함)에는 차이가 있습니다. 따라서 이 두가지 타입은 아래와같이 정의됩니다.

```
typedef std::pair<void*, std::size_t> mutable_buffer;
typedef std::pair<const void*, std::size_t> const_buffer;
```

여기서, mutable_buffer 는 const_buffer 로 변환될수 있지만, 반대로는 안됩니다.

그러나, Boost,Asio 는 위와 같은 정의를 사용하지 않습니다. 대신, mutable_buffer 와 const_buffer 라는 두개의 클래스를 정의합니다. 이러한 이유는, 다음과 같은 상황에서 연속된 메모리를 불투명하게 표현하기 위해서입니다.

- 이 타입들은 변환에 있어서 std::pair 처럼 동작합니다. 이것은, muable_buffer 는 const_buffer 로 변환할수 있지만, 그 반대는 안된다는 것이지요.

- 여기에는 버퍼오버런을 방지하기위한 장치가 있습니다. 버퍼 인스턴스가 하나 있다고 할 때, 유저는 또다른 버퍼를 오직 같은 영역의 메모리나, 그 영역 내부의 영역에만 표현할 수 있습니다. 좀더 안전하게 하기 위해, 라이브러리는 자동으로 배열(boost:array, POD 엘리먼트의 std::vector, std::string 등)의 크기를 결정하는 메커니즘을 포함합니다.

- 타입 안정성 위반은 명시적으로 buffer_cast 에 의해서 요청됩니다. 일반적으로 응용프로그램은 이렇게 할 필요가 없습니다만, 라이브러리 구현에서는 날 메모리(raw memory)를 운영체제 함수에 넘겨주기 위해 필요합니다.

마지막으로, 다중버퍼는 버퍼 객체를 컨테이너에 넣음으로써, scatter-gather 오퍼레이션에 전달될수 있습니다.(boost::asio::read 또는 boost::asio::write) MutableBufferSequence 와 ConstBufferSequence 의 개념은 std::vector 나 std::list, boost::array 같은 컨테이너들이 사용될수 있도록 정의되어있습니다.

왜 EOF 는 에러인가

- 스트림의 끝은 read,async_read,read_until 같은 함수가 실패하도록 합니다. 예를들어 N 바이트를 읽는 것은 EOF 때문에 미리 끝나게 될 것 입니다.
- EOF 에러는 사이즈 0 을 성공적으로 읽었을 때와 스트림의 끝을 구별하는 데 사용할 수있습니다.

행 기반 프로토콜

보통 사용되는 많은 인터넷 프로토콜들은 행 기반입니다. 이 말은, "\r\n" 문자열로 구별되는 프로토콜 요소를 가지고 있다는 말이지요. HTTP 나 SMTP,FTP 같은것이 그 예입니다.

행기반 프로토콜이나 다른 구분자를 사용하는 프로토콜의 구현을 좀더 쉽게 하기 위해서, Boost.Asio 는 read_until()함수와 async_read_until()함수를 포함합니다.

아래의 예는 HTTP 서버에서 클라이언트로부터 받은 HTTP 요청의 첫번째 행을 받기 위해서 async_read_until()함수를 사용하는 것을 보여줍니다.

```
class http_connection
{
  ...

  void start()
  {
    asio::async_read_until(socket_, data_, "\r\n",
        boost::bind(&http_connection::handle_request_line, this, _1));
  }
```

```
  void handle_request_line(boost::system::error_code ec)
  {
```

```
    if (!ec)
    {
      std::string method, uri, version;
      char sp1, sp2, cr, lf;
      std::istream is(&data_);
      is.unsetf(std::ios_base::skipws);
      is >> method >> sp1 >> uri >> sp2 >> version >> cr >> lf;
      ...
    }
  }

  ...

  asio::ip::tcp::socket socket_;
  asio::streambuf data_;
};
```

스레드

스레드 안정성

일반적으로 별개의 객체를 동시적으로 사용하는 것은 안전하지만, 하나의 객체를 동시에 사용하는 것은 안전하지 않습니다. 그러나, **io_service** 는 한 개의 오프젝트를 동시적으로 안전하게 사용하는을 강력하게 보장해줍니다.

내부 스레드

이 라이브러리에 대한 특정 플랫폼의 구현은 비동기적으로 에뮬레이트 되는 하나 이상의 내부 스레드를 유용하게 사용할수 있습니다. 가능한한, 이 스레드들은 라이브러리의 유저에게 보이지 않아야 합니다. 특히, 스레드는

- 유저의 코드를 직접 불러서는 안됩니다.
- 모든 시그널을 블록해야 합니다.

이러한 접근은 다음과 같은 보장사항을 통해 보완되었습니다.

- 비동기적 완료 핸들러는 오직 현재 **io_service::run()**를 호출중인 스레드에 의해서만 호출될 수 있습니다.

따라서, 알림(**notifications**)이 전달될 모든 스레드를 생성하고 관리하는 것은 라이브러리 유저의 책임이 됩니다.

이러한 접근의 이유는 다음과 같습니다.

- 하나의 스레드로부터만 **io_service::run()**을 호출하기 때문에, 사용자의 코드는 동기화와 관련된 개발 복잡성을 피할 수 있습니다. 예를들어, 라이브러리의 사용자는 단일 스레드기반의 확장성있는 서버를 구현할 수 있습니다.(유저의 관점에 따라)
- 라이브러리 유저는 스레드에서 스레드가 시작된 직후 그리고, 다른 코드가 실행되지 전에 초기화를 필요로 할 수도 있습니다. 예를들어, 사용자의 **MS Com** 은 다른 **COM** 오퍼레이션이 스레드에서 호출되기 전에 **CoInitializeEx** 를 호출해야만 합니다.
- 라이브러리의 인터페이스는 스레드의 생성과 관리 인터페이스와는 독립되어 있고, 스레드가 불가능한 플렛폼에서의 구현을 허락합니다.

스트랜드 Strands

스트랜드는 "이벤트 핸들러의 엄격한 순차적 호출" 이라고 정의됩니다.(즉, 동시적인 호출이 안된다는겁니다.) 스트랜드를 사용함으로써, 코드의 실행은 명시적인 락킹(locking)이 필요 없어지게 됩니다.(예를들어 뮤텍스 같은.)

스트랜드는 다음에서 설명하는 접근법 처럼 암시적이거나 명시적일 있습니다.

- 오직 하나의 스레드에서 io_service::run()을 호출한다는 것은 모든 이벤트 핸들러가 암시적인 스트랜드안에서 실행된다는 것을 의미합니다. io_service 는 핸들러가 오직 run()을 통해서만 호출되는 것을 보장해주기 때문이죠.
- 커넥션과 연관되어있는 비동기적인 오퍼레이션의 체인은 하나뿐이기 때문에(예를들어, HTTP 와 같은 half duplex 프로토콜 구현에서) 핸들러의 동시적인 실행 가능성이 없습니다. 이것이 암시적인 스트랜드 입니다.
- 명시적인 스트랜드는 io_service::strand 의 인스턴스입니다. 모든 이벤트 핸들러 함수 객체는 io_service::strand::wrap()함수로 감싸지거나, io_service::strand 객체를 통해 보내지고(posted) 디스패치 되어야 합니다.

async_read()나 async_read_until()같은 비동기적 오퍼레이션으로 구성된 경우에는, 만약 완료 핸들러가 스트랜드를 통한다면, 모든 중간의 핸들러는 같은 스트랜드를 통해서 수행되어야 합니다. 이것은 호출자와, 구성된 수행사이에 공유된 모든 객체에 대한 스레드 세이프한 접근을 보장합니다.(async_read()의 경우에는 그것은 소켓이되고, 호출자는 오퍼레이션을 취소하기 위해 cancel()을 호출할 수 있습니다.) 이것은 최종 핸들러와 연관된 사용자 지정 가능한 훅(hook)의 호출을 발송하는 모든 중간 핸들러를 위한 훅 함수를 가지고 있는 것으로 이루어집니다.

```
struct my_handler
{
  void operator()() { ... }
};

template<class F>
void asio_handler_invoke(F f, my_handler*)
{
  // 여기서 사용자 지정 호출을 합니다.
  // 기본구현은 f()를 호출하는 것입니다.
}
```

io_service::strand::wrap()함수는 함수객체가 스트랜드를 통해 실행 되도록 asio_handler_invoke 로 정의된 새로운 완료 핸들러를 생성합니다.

Platform-Specific Implementation (이부분은 따로 번역하지 않겠습니다.)

This design note lists platform-specific implementation details, such as the default demultiplexing mechanism, the number of threads created internally, and when threads are created.

Linux Kernel 2.4

Demultiplexing mechanism:

- Uses select for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed FD_SETSIZE.

Threads:

- Demultiplexing using select is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

Linux Kernel 2.6

Demultiplexing mechanism:

- Uses epoll for demultiplexing.

Threads:

- Demultiplexing using epoll is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

Solaris

Demultiplexing mechanism:

- Uses /dev/poll for demultiplexing.

Threads:

- Demultiplexing using /dev/poll is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

QNX Neutrino

Demultiplexing mechanism:

- Uses select for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed FD_SETSIZE.

Threads:

- Demultiplexing using select is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

Mac OS X

Demultiplexing mechanism:

- Uses kqueue for demultiplexing.

Threads:

- Demultiplexing using kqueue is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().

- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

FreeBSD

Demultiplexing mechanism:

- Uses kqueue for demultiplexing.

Threads:

- Demultiplexing using kqueue is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

AIX

Demultiplexing mechanism:

- Uses select for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed FD_SETSIZE.

Threads:

- Demultiplexing using select is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

HP-UX

Demultiplexing mechanism:

- Uses select for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed FD_SETSIZE.

Threads:

- Demultiplexing using select is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

Tru64

Demultiplexing mechanism:

- Uses select for demultiplexing. This means that the number of file descriptors in the process cannot be permitted to exceed FD_SETSIZE.

Threads:

- Demultiplexing using select is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

Windows 95, 98 and Me

Demultiplexing mechanism:

- Uses select for demultiplexing.

Threads:

- Demultiplexing using select is performed in one of the threads that calls io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().

Windows NT, 2000, XP, 2003 and Vista

Demultiplexing mechanism:

- Uses overlapped I/O and I/O completion ports for all asynchronous socket operations except for asynchronous connect.
- Uses select for emulating asynchronous connect.

Threads:

- Demultiplexing using I/O completion ports is performed in all threads that call io_service::run(), io_service::run_one(), io_service::poll() or io_service::poll_one().
- An additional thread per io_service is used for the select demultiplexing. This thread is created on the first call to async_connect().
- An additional thread per io_service is used to emulate asynchronous host resolution. This thread is created on the first call to either ip::tcp::resolver::async_resolve() or ip::udp::resolver::async_resolve().