

A close-up photograph of an owl's face, focusing on its large, dark eye with a yellow ring around the pupil. The feathers are detailed, showing shades of brown, white, and black.

Hanbit eBook

Realtime 27

Boost.Asio를 이용한 네트워크 프로그래밍

최홍배 지음

Boost.Asio를 이용한 네트워크 프로그래밍

지은이_최홍배

2003년부터 현재에 이르기까지, 보드 게임부터 MMORPG 게임을 아우르는 다양한 온라인 게임의 서버 프로그램을 만들어온 개발자다. 게임 개발자로서 프로그래밍 언어 중 C++을 주 언어로, C#을 보조 언어로 사용하고 있다. 요즘은 C++11 프로그래밍과 Linux 플랫폼 프로그래밍, 심도 있는 .NET 기술, JavaScript에 대해 공부하고 있다. 기술이나 경험을 공유하는 것을 좋아하여 게임 개발자 커뮤니티나 세미나 강연을 통해 다른 프로그래머와 활발히 교류한다. 웹이 대중화되기 전부터 프로그래밍 공부를 해와서 그런지 여전히 새로운 기술을 배울 때는 책을 더 선호하여, 지금도 매달 새로운 프로그래밍 관련 책을 읽으며 연구하고 있다. 현재 티쓰리 엔터테인먼트 삼국지천 팀에서 근무 중이다.

Boost.Asio를 이용한 네트워크 프로그래밍

초판발행 2013년 6월 4일

지은이 최홍배 / 펴낸이 김태현

펴낸곳 한빛미디어(주) / 주소 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

전화 02-325-5544 / 팩스 02-336-7124

등록 1999년 6월 24일 제10-1779호

ISBN 978-89-6848-611-1 15000 / 정가 9,900원

책임편집 배용석 / 기획 김병희 / 교정 안선희

디자인 표지 여동일, 내지 스튜디오 [밈], 조판 김현미

마케팅 박상용, 박주훈, 정민하

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.

한빛미디어 홈페이지 www.hanb.co.kr / 이메일 ask@hanb.co.kr

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2013 최홍배 & HANBIT Media, Inc.

이 책의 저작권은 최홍배와 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanb.co.kr)로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 서문

필자가 처음 온라인 게임 서버를 만들 때만 해도, 이해하기 어려운 IOCP 방식보다 기존의 BSD Socket 방식이 네트워크 프로그래밍(Windows 플랫폼에서)에 더 좋다는 논의로 프로그래머 커뮤니티 사이트에서 격렬한 논쟁이 벌어지기도 했었다. 그러나 지금은 고성능 네트워크 프로그래밍에 당연하다는 듯이 IOCP를 사용하는 비동기 IO 방식을 사용한다.

Windows 플랫폼에서 고성능 네트워크 프로그램을 만들 때 필수적이기는 하지만, 사용 방법이 간단하지 않고 다른 플랫폼(가령, Linux)에서 사용할 수 없다는 단점이 있다(타 플랫폼의 비동기 프로그래밍 방식도 비슷하다). 네트워크 프로그래밍 경험이 적은 경우라면 학습하는 데 적지 않은 시간이 소요되고, IOCP로 만든 프로그램이 실제로 올바르게 동작하지 않을 수도 있다. 또 요즘은 방대한 유저를 수용하기 위해 많은 서버 컴퓨터를 운영하고 있는데, OS 라이센스 문제로 무료 플랫폼인 Linux의 사용 역시 증가하는 추세다. 하지만 Linux에서는 IOCP로 만든 프로그램을 사용할 수 없으므로 프로그램을 새로 만들어야 한다는 문제가 있다.

위에서 밝힌 문제들을 해결할 수 있는 방법의 하나가 C++ 오픈 소스 라이브러리로 유명한 Boost 라이브러리의 ‘Asio’를 사용하는 것이다. Asio를 사용하면 비동기 IO 프로그래밍을 손쉽게 사용할 수 있고, 코드가 간결해져서 유지보수가 좋아진다. 또한, Boost가 멀티 플랫폼을 지원하기 때문에 Windows 플랫폼에서 만든 프로그램을 거의 수정 없이 Linux로 전환할 수 있다.

C++ 프로그래머라면 대부분 Boost 라이브러리를 알고 있을 테지만, 아직 이에 관한 국내 출판물은 없는 상태다. 그래서 Boost 라이브러리의 일부만 알고 있는 경우가 많으며, 아직은 Asio에 관해서도 잘 모르는 사람이 많은 것 같다. 이 책을 통해서 Asio를 쉽게 공부하여 하루빨리 고성능 네트워크 프로그램을 만들 수 있게 되길

바란다.

이 책은 전자책으로 출판되는 만큼 배포도 쉬울 것이라 생각한다. 필자는 이러한 전자책의 장점을 잘 활용하여, 책에싣지 못한 Asio에 대한 정보나 팁 등을 출판 이후에도 가능하면 지속적으로 업데이트해 나가려고 한다.

책을 준비하며 도움을 준 사람들에게 감사의 마음을 전한다. 먼저, 결혼 이후에도 집에서 프로그래밍 공부를 할 수 있도록 많이 배려해주고 언제나 건강을 챙겨주는 사랑하는 아내 선민 씨에게 고마움을 전하고 싶다.

또 ‘온라인서버제작자모임’을 통해서 온라인 게임 서버 개발에 관한 정보를 공유해 주고 커뮤니티를 관리하느라 수고가 많은 임영기님, 이지현님, 이육진님, 혀승욱님, 최우영군에게 고마움을 전한다. 마지막으로 게임 개발자 지망생 시절부터 같이 공부해왔고, 현재 게임 업계에서 같이 일하고 있으며, 평소 싫은 내색 하나 없이 필자의 부탁을 잘 들어주는 조현진군에게 고맙다는 말을 전하고 싶다.

집필을 마무리하며

지은이 **최홍배**

도움을 주신 분들

Technical Reviewer

김승혁

현재 블루사이드 Kingdom under fire 2 online team에서 서버 프로그래머로 근무 중이다. ‘우공이산’을 신조로 빠르게 진화하는 IT기술에 도태되지 않으려, 노력하는 프로그래머다.

구승모

현재 nhn next에서 게임서버 전공 교수로 테라의 전투시스템 구현 및 서버 기반 제작 참여하였다.

김승혁

NCsoft Lineage Eternal 팀 서버 프로그래머로 유연한 구조와 최소한의 코드를 선호하는 서버 개발자다.

Beta Reader

박동섭, 송형주, 이근호, 이정재

대상 독자 및 소스 코드

초급

초·중급

중급

중·고급

고급

이 책은 C++ 프로그래밍의 기본과 BSD Socket을 사용한 아주 기본적인 네트워크 프로그래밍에 대해서는 알고 있는 데 반해, 네트워크 프로그래밍 경험이 적고 고성능 네트워크 프로그래밍을 위한 비동기 IO 프로그래밍에 대해서는 잘 모르는 개발자를 대상으로 한다.

또한, 기존 방식 대신 Boost.Asio를 사용한 고성능 멀티 플랫폼 대응의 네트워크 프로그램을 만드는 방법은 물론, 비동기 프로그래밍을 일반적인 프로그래밍에서 사용하는 방법 역시 다루고 있기 때문에, IOCP 등의 비동기 IO 네트워크 프로그래밍 경험이 있는 중급 이상의 개발자에게도 도움이 될 것이다.

이론적인 설명보다는 Asio를 사용하는 실용적인 프로그래밍에 초점을 맞추고 있는 만큼, 책에서는 이에 관한 모든 부분을 세세하게 다루지 않는다. 그러나 책을 읽고 난 후 즉시 Asio를 사용하여 프로그래밍하는 것이 가능하도록 구성했다.

필자의 경험상 프로그래밍 관련 책에서 익힌 것은 반드시 직접 코딩해보고, 배운 기술을 사용하여 직접 프로그램을 만들어보아야 자기 것으로 만들 수 있다. 그러니 그냥 눈으로만 보지 말고 꼭 프로그램을 직접 만들어보기 바란다.

소스 코드는 다음 링크를 통해서 다운받을 수 있다.

- <http://www.hanbit.co.kr/exam/2611>

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 편리한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정 보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

차례

0 1	Boost 라이브러리	1
1.1	Boost 라이브러리 설치	2
1.2	멀티 플랫폼 지원	3
0 2	Boost.Asio	5
2.1	멀티 플랫폼 지원	5
2.2	신뢰성	6
2.3	성능	7
2.4	편의성 및 범용성	8
0 3	간단한 Echo 서버, 클라이언트 프로그램 만들기	11
3.1	Boost.Asio를 사용하기 위한 준비	11
3.2	동기 I/O 방식의 TCP/IP Echo 서버	12
3.3	동기 I/O 방식의 TCP/IP Echo 클라이언트	18
3.4	관련 Boost.Asio API	24
0 4	비동기 I/O를 사용한 Echo 서버, 클라이언트 프로그램 만들기	34
4.1	비동기 I/O 프로그래밍의 특징	34
4.2	비동기 I/O 방식의 TCP/IP Echo 서버	36
4.3	비동기 방식의 TCP/IP Echo 클라이언트	49
4.4	관련 Boost.Asio API	57

0 5	채팅 프로그램 만들기	61
	5.1 채팅 서버	61
	5.2 채팅 클라이언트	70
	5.3 개선할 점	76
0 6	비동기 I/O를 사용한 UDP Echo 서버, 클라이언트 만들기	78
	6.1 UDP로 데이터 보내고 받기	78
	6.2 관련 Boost.Asio API	80
0 7	Boost.Asio의 Timer 사용하기	82
	7.1 기본적인 타이머	82
	7.2 반복하는 타이머	85
	7.3 설정한 타이머 취소하기	87
0 8	Boost.Asio를 사용한 백그라운드 메시지 처리	93
0 9	Boost.Asio의 기타 기능들	100
	9.1 Boost.Asio와 스레드	101
	9.2 Windows에서 비동기로 파일 읽기	112
	9.3 resolver를 사용하여 도메인 네임을 IP 주소로 변환하기	114
1 0	참고 자료	119

1 | Boost 라이브러리

Boost 라이브러리를 알고 있는가? 아직 모르고 있다면, C++ 프로그래머로서 적지 않은 손해를 보고 있는 셈이다. Boost 라이브러리에 있는 유용한 라이브러리를 사용하면, 개발에 필요한 기능들을 직접 만들지 않아도 될뿐더러 효율성 또한 좋아서 C++ 프로그래밍의 생산성을 크게 올릴 수 있기 때문이다.

Boost 라이브러리는 C++ 프로그래머를 위한 유용한 오픈 소스 C++ 라이브러리의 모음으로, 수많은 C++ 고급 프로그래머들이(C++ 표준 위원회 멤버 등) 개발에 참여하고 있다. 그래서 Boost 라이브러리는 기능 면에서 실용적이며 안정성 또한 높다.

그림 1-1 Boost 공식 홈페이지(www.boost.org)

The screenshot shows the official website for Boost.org. At the top, there's a green banner with the Boost logo and the text "...one of the most highly regarded and expertly designed C++ library projects in the world." Below the banner, the page has a dark header with the word "boost" in white. The main content area is divided into several sections:

- WELCOME TO BOOST.ORG!**: A brief introduction stating that Boost provides free peer-reviewed portable C++ source libraries.
- DOWNLOADS**: A section for downloading the current release (Version 1.53.0), release notes, and documentation. It also includes links for more downloads and RSS feeds.
- NEWS**: A list of recent news items, such as "Boost.Locale security notice" and "Important information for users of boost::result_of...".
- SEARCH**: A search bar with a "Search" button and a dropdown menu containing links like "Getting Started", "Download", "Libraries", "Macros", "Reporting and Fixing Bugs", "Wiki", "INTRODUCTION", "COULDNOTY", "DEVELOPMENT", and "DOCUMENTATION".

아직까지 Boost 라이브러리를 사용해본 적이 없다면 지금 바로 사용하기를 권한다. 아직도 일부에서는 Boost 라이브러리의 안정성을 의심하기도 하는데, 이제 무의미한 의심은 그만해도 될 것 같다. 한국의 게임 개발 회사 상당수가 이미 Boost

라이브러리를 사용하고 있으니 말이다. 무슨 설명이 더 필요한가!

2011년 8월에 표준 작업이 끝난 C++11 버전에는 많은 Boost 라이브러리가 새로운 표준 라이브러리로 추가되었다. 예를 들어 Visual Studio(이하 VS) 2008 SP를 설치하면 사용할 수 있는 tr1 라이브러리 대부분이 Boost 라이브러리다. 그래서 C++11 버전을 지원하지 않는 VC를 사용하더라도, Boost 라이브러리를 사용하면 C++의 새로운 기능을 적지 않게 사용할 수 있다.

Boost.Asio의 사용 방법과 장점을 알아보기 전에, 우선 Boost 라이브러리 설치 방법과 특징을 간략하게 살펴보자.

1.1 Boost 라이브러리 설치

Boost 라이브러리는 공식 웹 사이트와 Boost 라이브러리 컨설팅 BoostPro에서 다운받을 수 있다.

- Boost 공식 웹 사이트 : http://www.boost.org/users/history/version_1_52_0.html
- BoostPro 웹 사이트 : <http://www.boostpro.com/download/>

Boost 라이브러리 중 OS^{Operating System}, 운영체제의 커널 기능을 사용하는 라이브러리(thread, filesystem 등)는 직접 빌드해야 한다. VS에서 BoostPro를 사용한다면, 이미 빌드된 lib이나 dll 파일을 얻을 수 있다(VS 2003버전 ~ VS 2010 버전을 사용한다면, 시스템 종류(32비트나 64비트 운영체제에 상관없이) BoostPro를 실행할 수 있다).

다만 BoostPro를 이용하면 Boost 라이브러리의 최신 버전을 공식 사이트에서 받는 것보다 상당히 늦게 받을 수밖에 없다는 단점이 있다. 또한 Windows 이외

의 플랫폼을 지원하지도 않으며, 최신 버전의 VS를 지원하지 않을 수도 있다⁰¹(VS 2012를 지원하지 않는다).

Boost 라이브러리를 사용하기 위해 본인이 직접 라이브러리를 빌드해야 하는 것 이 꽤 까다롭게 여겨질 수도 있다. 하지만 Boost 라이브러리의 대부분은 헤더 파일을 포함하는 것으로 사용할 수 있으며, Boost 라이브러리에서 제공하는 툴을 사용하면 빌드하는 것도 아주 간단하다.

빌드하는 방법에 대해 자세히 알고 싶다면 다음의 웹 사이트를 참고하기 바란다(이 외에도 네이버나 구글에서 검색하면 다양한 Boost 라이브러리 빌드 방법을 찾을 수 있다).

- Boost 라이브러리 빌드하기: <http://jacking.tistory.com/986>

1.2 멀티 플랫폼 지원

필자는 게임 개발자로서 경력을 쌓고 있는데, 몇 년 전까지만 해도 게임은 대부분 Windows 플랫폼으로 개발했다. 때문에 멀티 플랫폼이라는 단어는 별 의미가 없었다. 그러나 지금은 모바일 플랫폼이 무서운 기세로 성장하는 추세라서 iOS나 Android 등 비(非) Windows 플랫폼을 무시할 수 없게 되었다.

Boost 라이브러리의 장점 중 하나가 멀티 플랫폼 지원이다. 이처럼 Boost 라이브러리가 Windows 이외에도 Linux OS, iOS, Android 등을 지원하기 때문에 모바일 플랫폼에 적절히 대응할 수 있다. 또한 거의 모든 플랫폼에 유연하게 대응할 수 있기 때문에, Boost 라이브러리를 사용하여 얻은 지식과 경험의 가치는 쉽게 바래지 않으며 오랫동안 이용 가능하다.

01 이 책을 집필하던 2013년 2월 초, BoostPro의 설립자가 다른 회사로 이직하면서 회사 문을 닫았다. 때문에 이후 1.5.2 버전부터는 빌드된 라이브러리를 제공하지 않을 것 같다.

iOS나 Android에서 Boost 라이브러리를 사용하는 방법을 알고 싶다면, 구글에서 ‘boost ios’와 ‘boost android’를 각각 검색해보기 바란다. 다양한 자료를 찾아볼 수 있을 것이다(Boost 라이브러리나 iOS, android는 다양한 버전이 있으니 필요 할 때 직접 검색하기 바란다).

이상으로 Boost 라이브러리에 대해 짧게 설명했다. 핵심 부분은 이해했을 것이니, 이제 본격적인 주제로 들어가보자.

2 | Boost.Asio

Boost.Asio는 Boost 라이브러리 중 하나로, 네트워크 및 비동기 프로그래밍과 관련된 라이브러리다. Boost.Asio를 사용하면 어렵고 복잡한 고성능 멀티 플랫폼 네트워크 프로그래밍과 비동기 프로그래밍을 아주 쉽게 할 수 있다. Boost.Asio의 특징은 다음과 같다.

- 네트워크 프로그래밍에서 주로 사용한다.
- Asynchronous I/O(비동기 입출력)를 지원하여, 상대적으로 많은 시간이 걸리는 I/O 작업을 빠르게 처리할 수 있다.
- 파일 입출력, 시리얼 포트 입출력, 범용적인 비동기 프로그래밍에 사용할 수 있다.
- 멀티 플랫폼을 지원한다.

2.1 멀티 플랫폼 지원

멀티 플랫폼을 지원하기 위해 해당 플랫폼의 특수한 기능을 지원하지 않는 라이브러리도 있지만, Boost.Asio는 각 플랫폼에서 지원하는 특정 기능을 대부분 지원한다. 그래서 Boost.Asio는 멀티 플랫폼을 지원하면서도 각 플랫폼의 전용 API보다 성능이 떨어지지 않는다.

Boost.Asio의 OS 플랫폼별 지원 기능 중에서 특히나 중요한 것이 비동기 입출력(Asynchronous I/O) 지원 여부다. Boost.Asio를 사용하여 고성능 네트워크 프로그래밍을 하려면 비동기 입출력 지원이 필수이기 때문이다. 플랫폼별 Boost.Asio의 비동기 입출력 기능 구현 방법은 표 2-1에서 확인할 수 있다.

표 2-1 Boost.Asio 구현을 위해 플랫폼에서 사용한 API

플랫폼	Boost.Asio 구현
Linux Kernel 2.4	비동기를 지원하지 않으므로 select만 사용. FD_SIZE 크기에 연결 수 제한
Linux Kernel 2.6 이상	epoll 사용
FreeBSD	Kqueue 사용
Mac OS X	Kqueue 사용
Solaris	/dev/poll 사용
Windows 2000 이상	Ovelapped I/O와 IOCP 사용

여기서 잠깐_ 비동기 입출력(Asynchronous I/O)

- 네트워크 프로그래밍에서의 blocking, non-blocking, synchronous, asynchronous에 관한 설명
<http://devsw.tistory.com/142>
- 비동기 I/O를 이용한 Boost 애플리케이션 실행에 관한 설명
<https://www.ibm.com/developerworks/linux/library/l-async/>

2.2 신뢰성

오픈 소스 라이브러리를 사용할 때 가장 신경 쓰이는 부분이 신뢰성이다. 신뢰성 때문에 오픈 소스를 사용하지 않고 직접 만들어서 사용하는 경우도 있는데, 다행히 Boost.Asio는 신뢰성에 관한 한 걱정하지 않아도 된다.

현재 Boost.Asio는 많은 곳에서 사용되고 있다. 국내 한 유명 IT 회사는 회사 내의 C++ 네트워크 라이브러리로 Boost.Asio를 사용하고 있으며, 이미 출시된 유명 온라인 게임에서도 사용하고 있다. 필자가 일했던 곳을 포함한 몇몇 회사는 온라인 P2P용 캐주얼 게임과 MMORPG에서도 Boost.Asio를 사용 중이다. Boost.Asio의 신뢰성이 낮다면 기업들이 사용하겠는가. 답은 이미 나와 있다.

이처럼 Boost.Asio는 오픈 소스답게 많은 곳에서 사용 중이며, 버그가 있었다면 이미 발견하여 수정했을 확률이 아주 높다. 그리고 C++11 이후의 새로운 C++ 표준 STL에 네트워크 라이브러리로 포함될 확률도 꽤 높다. 반면 우리가 직접 만든 것은 우리만 사용하기 때문에, 사용 시간이나 사용처가 Boost.Asio에 비해 확연히 적다. 어떤 의미에서는 우리가 직접 만든 것에 버그가 있을 확률이 더 높다.⁰¹

2.3 성능

필자는 PC 온라인 게임의 서버 프로그래머로 일하고 있어서, 멀티 플랫폼 대응보다는 성능적인 측면을 더 중요하게 생각한다. 서버 프로그램은 대부분 플랫폼 하나를 정해서 개발하고, 클라이언트를 최대한 많이 연결할 수 있어야 하며, 빠르게 데이터를 주고받을 수 있어야 하기 때문에 필자 같은 서버 프로그래머는 필연적으로 성능에 민감하다. Boost.Asio와 ‘직접 만든 네트워크 라이브러리’의 성능 차이를 그림 2-1의 자동차 사진을 통해 비교해보겠다.

그림 2-1 네트워크 라이브러리 성능 차 비교



01 Boost 라이브러리 1.4 버전 이후의 Asio에서는 안정성을 넘어서 성능 투닝이 많이 되었다.

네트워크 라이브러리를 직접 만들 경우, 특별한 OS와 프로그램에 특화되게 만들 수 있어서, 이론적으로 보자면 사진 (A)의 F1 머신 같이 엄청난 성능을 낼 수 있다. Boost.Asio는 다양한 OS와 범용적인 곳에서 사용하기 때문에 (A)보다는 좀 떨어지는 (C)의 스포츠카 정도의 성능을 낼 수 있다. 그런데 실제로 우리가 직접 만들어 보면 아주 잘 만들어야 (A)와 같은 성능을 낼 수 있다. 누구나 F1 머신 수준의 성능을 낼 수 있을까? 이 책을 읽고 있는 독자라면 답을 알고 있을 것이다. 지식이나 경험에 부족하면 사실 사진 (B)의 일반 자동차 정도의 성능도 내기 어렵다.

아주 특별한 경우가 아니라면, Boost.Asio는 고성능을 요구하는 네트워크 프로그래밍에도 사용할 수 있다(고성능 네트워크 프로그램을 만드는 곳이 온라인 게임인데, 여기서도 문제없이 잘 사용하고 있다).

2.4 편의성 및 범용성

아무리 좋은 것이라도 사용하기 어렵다면 효율성이 떨어진다(업무에 필요 없다면 사용을 꺼릴 것이다).

그림 2-2 개발 편의도 비교



필자는 예전에 건강을 위해 양파즙을 구입하려고 한 적이 있는데, 사진 (a)와 같이 즙으로 만들어 판매하는 것을 사면 간단할뿐더러 신뢰할 만한 것을 구할 수 있었다. 그러나 (a)를 구매해서는 필자에게 어울리는, 좀 다른 양파즙을 얻기 어렵다는 문제가 있었다. 즉, 필자가 (a)에 맞추어야 했다. 필자에게 꼭 맞는 양파즙을 구하려면 사진 (b)와 같이 직접 생양파를 사서 처음부터 만들어야 했는데, 그러면 필자가 직접 해야 할 일이 너무 많고 질 좋은 양파를 구하는 것도 문제였다. 이에 반해 사진 (c)처럼 어느 정도 손질되어 있는 양파를 구매하면 (a)와 (b)의 장점을 취할 수 있었다. 그래서 필자는 (c)를 구매해서, 스스로에게 꼭 맞는 양파즙을 만들어 먹고 기운을 차렸다.

이 예를 네트워크 프로그램을 만드는 것에 대입하면 (a)는 상용 라이브러리를 사용하는 경우고, (b)는 자신이 직접 네트워크 라이브러리를 만드는 경우며, (c)는 Boost.Asio를 사용한 경우다.

Boost.Asio를 사용하여 네트워크 프로그래밍하면 네트워크 라이브러리를 직접 만드는 것에 비해 개발 속도와 성능, 신뢰도가 높아지는 것은 물론, 자신이 원하는 기능을 추가하는 데 좀 더 많은 시간을 쓸 수 있다.

네트워크 프로그래밍이나 비동기 프로그래밍은 일반적으로 쉽게 접할 수 있는 것이 아니라서, Boost 라이브러리는 알아도 그에 포함되어 있는 Asio에 대해서는 잘 모르는 분들이 많으리라 생각한다. 필자가 지금까지 주절주절 설명한 것도 그 때문이다. 앞서 밝혔듯이 이 책은 실전에 도움되는 것을 목표로 하고 있으니 Boost.Asio에 대한 이론적인 설명은 이것으로 끝내겠다. 다음 장부터는 간단한 소켓 프로그래밍 예제를 통해서 ‘고성능 비동기 네트워크 프로그래밍’과 ‘범용적 비동기 프로그래밍’에 대해 알아보겠다.

NOTE_ Boost.Asio는 Boost 라이브러리와 별도로 만들어졌다가 이후에 Boost 라이브러리에 편입되었다. 그래서 지금도 Boost 라이브러리와 별도로 사이트가 운영되고 있는데, 여기서도 Asio 라이브러리를 다운받을 수 있다.

- Asio C++ library : <http://think-async.com/Asio>

The screenshot shows the homepage of the Asio C++ library website. The header includes a navigation bar with links like 'asio Home', 'Download', 'Documentation', 'Source Repository', 'SourceForge Page', 'Mailing List', 'Blog', and 'Support'. Below the header, there's a search bar with 'Jump' and 'Search' buttons. The main content area has several sections:

- Who is using Asio?**: A section encouraging users to add their project to a list.
- Latest Stable Release**: Information about Asio version 1.4.8, with links to 'Download', 'Release notes', 'Documentation (non-Boost)', and 'Documentation (Boost)'.
- Asio and Boost.Asio**: A note stating that Asio comes in two variants: (non-Boost) Asio and Boost Asio, with a 'More...' link.
- License**: A note that Asio is released under the [Boost Software License](#).
- Supported Platforms**: A list of platforms and compilers tested, including Win32 and Win64 using Visual C++ 7.1, 8.0, 9.0 and 10.0, Win32 using MinGW, Win32 using Cygwin (with `_USE_VSOCKETS` defined), Linux (2.4 or 2.6 kernels) using g++ 3.3 or later, Solaris using g++ 3.3 or later, and Mac OS X 10.4 and later using g++ 3.3 or later.
- Latest Development Release**: Information about Asio version 1.5.3, with links to 'Download', 'Release notes', 'Documentation (non-Boost)', and 'Documentation (Boost)'.
- Other Notes**: A section listing platforms that may also work, such as Win32 using Borland C++ Builder 2007, AIX 5.3 using XL C/C++ v9, HP-UX 11.0 using patchlevel ac++ A.05.14, QNX Neutrino 6.3 using g++ 3.3 or later, Solaris using Sun Studio 11 or later, and Tru64 v4.1 using Common C++ v7.1.

3 | 간단한 Echo 서버, 클라이언트 프로그램 만들기

이제 Boost.Asio를 사용하여 네트워크 프로그래밍을 어떻게 하는지 본격적으로 알아보자. 이전에 OS의 API(Windows 혹은 Linux의 Socket API)를 사용하여 네트워크 프로그래밍을 해본 적이 있다면, Boost.Asio를 이용해서 네트워크 프로그래밍을 하면 얼마나 간단해지는지 알고 있을 것이다.

여기서는 Boost.Asio 사용 방법을 설명하는 것이 핵심이므로, 사용 방법에 집중할 수 있도록 예제를 최대한 간단하게 만들었다. 실용적인 예제는 아니지만 코드가 짧고 간단해서 Boost.Asio 사용 방법을 배우는 데 좋을 것이다.

3.1 Boost.Asio를 사용하기 위한 준비

Boost.Asio를 사용하려면, 먼저 Boost 라이브러리를 빌드하여(“1.1 Boost 라이브러리 설치” 참고) Boost.Asio와 관련된 lib 파일을 만들어야 한다. lib 파일이 준비되었다면, 다음과 같은 헤더 파일 하나만 추가하여 Boost.Asio를 사용할 수 있다.

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

이것으로 Boost.Asio를 사용할 준비는 끝났다. 이제 사용만 하면 된다. 정말 간단하지 않은가? 참고로, lib 파일 없이 헤더 파일만 추가하는 간단한 작업으로 Boost.Asio를 사용할 수도 있다.

예제 프로그램은 Boost.Asio를 사용한 간단한 네트워크 프로그래밍으로, 동기 I/O 버전의 TCP 프로토콜을 사용하는 Echo 서버와 클라이언트다.

3.2 동기 I/O 방식의 TCP/IP Echo 서버

예제 3-1은 TCP/IP 프로토콜을 서버 프로그램으로 사용하여 하나의 클라이언트 접속을 받은 후 클라이언트가 보내는 메시지를 받으면, 그 메시지를 다시 클라이언트에 보내주는 일종의 Echo 서버 프로그램⁰¹이다.

예제 3-1 SynchronousTCPServer

```
#include <SDKDDKVer.h>
// 'SDKDDKVer.h'는 VS 2012에서 프로젝트 만들면 자동으로 넣어주는
// 헤더 파일이다. 없어도 되는 헤드 파일이다.

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

const char SERVER_IP[] = "127.0.0.1";
const unsigned short PORT_NUMBER = 31400;

int main()
{
    boost::asio::io_service io_service;
    boost::asio::ip::tcp::endpoint endpoint( boost::asio::ip::tcp::v4(),
        PORT_NUMBER );
    boost::asio::ip::tcp::acceptor acceptor( io_service, endpoint );

    boost::asio::ip::tcp::socket socket(io_service);
    acceptor.accept(socket);
    std::cout << "클라이언트 접속" << std::endl;

    for (;;)
```

01 예제는 에코(Echo) 서버/클라이언트다. 상대방이 보낸 것을 그대로 돌려주는 에코의 뜻 그대로, 서버는 클라이언트가 보낸 데이터를 다시 그대로 클라이언트에 보낸다.

```
{  
    std::array<char, 128> buf;  
    buf.assign(0);  
    boost::system::error_code error;  
    size_t len = socket.read_some(boost::asio::buffer(buf), error);  
  
    if( error )  
    {  
        if( error == boost::asio::error::eof )  
        {  
            std::cout << "클라이언트와 연결이 끊어졌습니다" << std::endl;  
        }  
        else  
        {  
            std::cout << "error No: " << error.value() << " error Message: "  
                << error.message() << std::endl;  
        }  
        break;  
    }  
  
    std::cout << "클라이언트에서 받은 메시지: " << &buf[0] << std::endl;  
    char szMessage[128] = {0,};  
    sprintf_s( szMessage, 128-1, "Re:%s", &buf[0] );  
    int nMsgLen = strnlen_s( szMessage, 128-1 );  
  
    boost::system::error_code ignored_error;  
    socket.write_some(boost::asio::buffer(szMessage, nMsgLen), ignored_error);  
    std::cout << "클라이언트에 보낸 메시지: " << szMessage << std::endl;  
}  
getchar();  
return 0;  
}
```

예제 3-1은 코드가 짧아서 전체가 한눈에 들어올 것이다. 한눈에 들어오지 않는다고 해도, 필자가 하나하나씩 설명할 테니 편안한 마음으로 잘 따라오기 바란다. 이 예제를 제대로 이해하면 Boost.Asio로 간단한 네트워크 프로그램을 만들 수 있으며, 앞으로 살펴볼 기능이 추가된 예제 역시 쉽게 이해할 수 있다(네트워크 프로그래밍에 관한 경험이 부족하다고 생각하는 개발자라면 2장과 3장의 예제 코드를 모두 외워서 사용하기 바란다. 코드가 짧아서 생각보다 외우기 쉬우며, 코드를 외워 두면 자신감도 생기고 이해가 더 잘 될 것이다).

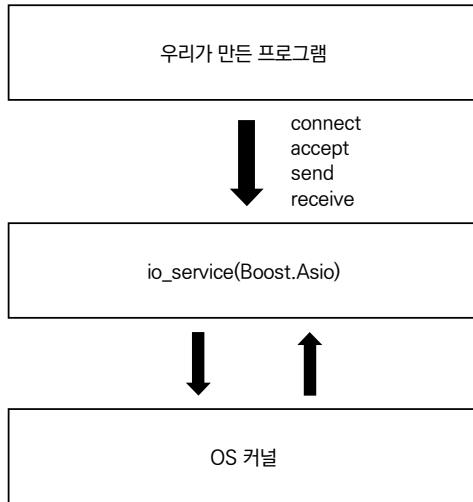
3.2.1 클라이언트 접속 준비

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

boost::asio::io_service는 Boost.Asio의 핵심 중의 핵심이다. io_service는 커널에서 발생한 I/O 이벤트를 ‘디스패치⁰²’ 해주는 클래스로, io_service를 통해서 커널에서 발생한 네트워크상의 접속 받기, 접속하기, 데이터 받기, 데이터 보내기 이벤트를 알 수 있다. 그래서 Boost.Asio에서 socket과 같은 객체를 생성할 때 io_service를 인자로 넘겨줘야 한다.

02 다중 태스킹 환경에서 우선순위가 높은 작업이 수행될 수 있도록 시스템 자원을 할당하는 것을 말한다.

그림 3-1 동기 버전에서의 io_service



```
boost::asio::ip::tcp::endpoint endpoint( boost::asio::ip::tcp::v4(),
    PORT_NUMBER );
boost::asio::ip::tcp::acceptor acceptor( io_service, endpoint )
```

endpoint는 네트워크 주소를 설정한다(이 주소로 클라이언트가 접속한다). 서버와 클라이언트는 endpoint 설정 방식이 다른데, 서버는 IP 주소(IPv4 또는 IPv6)와 포트 번호를 사용한다. 예제에서는 IPv4 방식의 주소 체계를 사용했다.

acceptor 클래스는 클라이언트의 접속을 받아들이는 역할을 하는데, io_service와 endpoint를 인자로 사용했다.

3.2.2 클라이언트의 접속 받기

```
boost::asio::ip::tcp::socket socket(io_service);
acceptor.accept(socket);
```

예제 3-1은 TCP/IP 프로토콜을 사용하므로 접속한 클라이언트에 할당할 ‘tcp::socket’을 만든다. 그리고 만든 socket을 통해서 클라이언트가 보낸 메시지를 주고받아야 하므로 io_service를 할당한다.

위에서 만든 acceptor의 accept 멤버 함수에 클라이언트용 socket 객체를 넘겨주면 서버가 클라이언트를 접속받을 준비는 끝난다. 이 예제는 동기형 방식⁰³을 사용하고 있어서, accept 멤버 함수를 호출한 후 클라이언트의 접속이 완료될 때까지 대기 상태가 된다.

3.2.3 클라이언트가 보낸 메시지 받기

```
| std::array<char, 128> buf;  
| buf.assign(0);
```

클라이언트가 보낸 메시지를 담을 버퍼를 만든다(std::array<char, 128> buf;). 예제 3-1에서는 char형 배열과 호환되는 STL의 array 컨테이너를 사용했다. 물론 array가 아닌 char형 배열을 사용해도 괜찮다.

```
| boost::system::error_code error;
```

error_code 클래스는 시스템에서 발생하는 에러 코드를 ‘랩핑wrapping’⁰⁴한 클래스로, 에러가 발생하면 에러 코드와 에러 메시지를 얻을 수 있다.

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

socket 클래스의 read_some 멤버 함수를 사용하여 클라이언트가 보낸 데이터를 받는다. 인자로 데이터를 담을 버퍼와 에러 코드를 받을 error_code를 사용하며 데이터를 받으면 받은 데이터 크기를 반환한다. 예제 3-1은 동기형 방식이므로 데

03 요청 후 답변을 받을 때까지 더 이상 진행하지 않고 기다리는 방식이다. 답변을 받으면 다음 단계를 진행한다.

04 기본형 자료를 객체형으로 바꿀 때 사용하는 클래스다.

이터를 다 받을 때까지 대기 상태에 들어간다.

클라이언트에서 보낸 데이터를 받으면 에러가 발생했는지 확인하기 위해 error_code를 조사해야 한다. 클라이언트의 접속이 끊어지는 경우도 read_some을 통해서 알 수 있는데, 이때는 error_code에 에러 값(boost::asio::error::eof)이 설정된다. 때문에 에러가 발생할 경우, 그것이 진짜 에러인지 아니면 접속이 끊어져 발생한 문제인지 꼭 조사해보아야 한다.

```
if( error )
{
    if( error == boost::asio::error::eof )
    {
        std::cout << "클라이언트와 연결이 끊어졌습니다" << std::endl;
    }
    else
    {
        std::cout << "error No: " << error.value() << " error Message: "
                << error.message() << std::endl;
    }
    break;
}
```

3.2.4 클라이언트에 메시지 보내기

```
char szMessage[128] = {0,};
sprintf_s( szMessage, 128-1, "Re:%s", &buf[0] );
int nMsgLen = strnlen_s( szMessage, 128-1 );

boost::system::error_code ignored_error;
socket.write_some(boost::asio::buffer(szMessage, nMsgLen),
    ignored_error);
```

3.2.3에서는 데이터를 받을 때 버퍼로 std::array를 사용했는데, 이번에는 일반적으로 자주 사용하는 char형 배열을 버퍼로 사용하였다. Boost.Asio는 C++ STL과 호환이 잘되어서 std::vector와 std::string도 버퍼로 사용할 수 있다.

메시지를 보낼 때는 socket 클래스의 write_some 멤버 함수를 사용한다. 인자는 받을 때와 비슷하게 buffer 클래스와 error_code 클래스를 사용한다. 단, 받을 때는 buffer 클래스에서 buffer 생성자에게 보낼 데이터의 양을 지정하여, szMessage 배열에 있는 데이터를 모두 보내지 않고 지정한 양만큼 보낸다.

이것으로 Echo 서버가 완성되었다. Windows나 Linux의 OS API를 사용한 네트워크 프로그램에 비해 소스 코드가 아주 짧다는 것이 확연하다. 한편, 예제 3-1은 Windows뿐만 아니라 Linux나 Mac OS 등 다양한 플랫폼에서 코드 하나 바꾸지 않고 그대로 사용할 수 있다.

NOTE_ Winsock API를 사용하여 예제 3-1과 비슷한 기능을 가진 서버를 만든 예

<http://blog.naver.com/liccorob/10155384326>

3.3 동기 I/O 방식의 TCP/IP Echo 클라이언트

클라이언트와 서버 간에 서로 공통 부분(Asio를 초기화하는 부분과 데이터 받기/

보내기는 서버와 클라이언트가 같은 방식을 사용한다)이 있어서 예제 3-1을 이해 했다면, 예제 3-2의 클라이언트 코드 역시 쉽게 이해할 수 있을 것이다.

예제 3-2 SynchronousTCPClient

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

const char SERVER_IP[] = "127.0.0.1";
const unsigned short PORT_NUMBER = 31400;

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

    boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::address::
        from_string(SERVER_IP), PORT_NUMBER);

    boost::system::error_code connect_error;
    boost::asio::ip::tcp::socket socket(io_service);
    socket.connect(endpoint, connect_error);

    if (connect_error)
    {
        std::cout << "연결 실패. error No: " << connect_error.value()
            << ", Message: " << connect_error.message() << std::endl;
        getchar();
        return 0;
    }
    else
    {
```

```

    std::cout << "서버에 연결 성공" << std::endl;
}

for (int i = 0; i < 7; ++i )
{
    char szMessage[128] = {0,};
    sprintf_s( szMessage, 128-1, "%d - Send Message", i );
    int nMsgLen = strlen_s( szMessage, 128-1 );

    boost::system::error_code ignored_error;
    socket.write_some( boost::asio::buffer(szMessage, nMsgLen), ignored_error);

    std::cout << "서버에 보낸 메시지: " << szMessage << std::endl;

    std::array<char, 128> buf;
    buf.assign(0);
    boost::system::error_code error;
    size_t len = socket.read_some(boost::asio::buffer(buf), error);

    if( error )
    {
        if( error == boost::asio::error::eof )
        {
            std::cout << "서버와 연결이 끊어졌습니다" << std::endl;
        }
        else
        {
            std::cout << "error No: " << error.value() << " error Message: "
                  << error.message().c_str() << std::endl;
        }
        break;
    }
}

```

```
    std::cout << "서버로부터 받은 메시지: " << &buf[0] << std::endl;
}

if( socket.is_open() )
{
    socket.close();
}

getchar();
return 0;
}
```

3.3.1 서버에 접속하기

```
boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip
::address::from_string(SERVER_IP), PORT_NUMBER);
```

서버(예제 3-1)의 endpoint 설정과 조금 다르다. 클라이언트는 접속할 서버의 IP 주소를 지정하는데, boost::asio::ip::address::from_string 클래스를 사용하여 문자열로 된 IP 주소를 Boost.Asio에서 사용하는 IP 주소로 변환한다.

```
boost::system::error_code connect_error;
boost::asio::ip::tcp::socket socket(io_service);

socket.connect(endpoint, connect_error);

if (connect_error)
{
    std::cout << "연결 실패. error No: " << connect_error.value()
        << ", Message: " << connect_error.message() << std::endl;
    getchar();
}
```

```
        return 0;
    }
else
{
    std::cout << "서버에 연결 성공" << std::endl;
}
```

socket 클래스의 connect 멤버 함수를 사용하여 서버에 접속을 시도하며, 접속이 성공하거나 실패할 때까지 대기 상태가 된다. error_code의 값을 조사하면 연결 성공 및 실패 여부를 알 수 있다.

3.3.2 서버에 데이터 보내기, 받기, 접속 끊기

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

예제 3-1과 같이 socket 클래스의 write_some과 read_some 멤버 함수를 사용하여 서버에 데이터를 보내고 받는다.

```
if( socket.is_open() )
{
    socket.close();
}
```

socket 클래스의 is_open() 멤버 함수를 사용하면 네트워크에 연결된 상태인지 아닌지를 알 수 있다. 'is_open()'이 true를 반환하면 연결된 상태다. 서버와 연결을 끊을 때는 close() 멤버 함수를 사용한다.

그림 3-2 서버와 클라이언트 실행 결과

```
C:\exam\boost\WSynchronousTCP\Server\Debug\WSynchronousTCPServer.exe
클라이언트 접속
클라이언트에서 받은 메시지: 0 - Send Message
클라이언트에 보낸 메시지: Re:0 - Send Message
클라이언트에서 받은 메시지: 1 - Send Message
클라이언트에 보낸 메시지: Re:1 - Send Message
클라이언트에서 받은 메시지: 2 - Send Message
클라이언트에 보낸 메시지: Re:2 - Send Message
클라이언트에서 받은 메시지: 3 - Send Message
클라이언트에 보낸 메시지: Re:3 - Send Message
클라이언트에서 받은 메시지: 4 - Send Message
클라이언트에 보낸 메시지: Re:4 - Send Message
클라이언트에서 받은 메시지: 5 - Send Message
클라이언트에 보낸 메시지: Re:5 - Send Message
클라이언트에서 받은 메시지: 6 - Send Message
클라이언트에 보낸 메시지: Re:6 - Send Message
클라이언트와 연결이 끊어졌습니다
```

(서버)

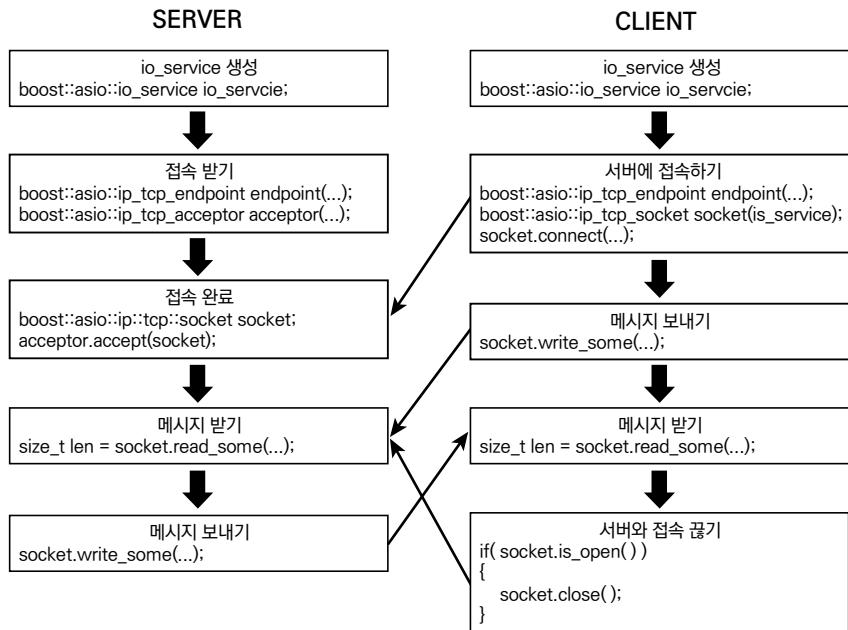
```
C:\exam\boost\WSynchronousTCPClient\Debug\WSynchronousTCPClient.exe
서버에 연결 성공
서버에 보낸 메시지: 0 - Send Message
서버로부터 받은 메시지: Re:0 - Send Message
서버에 보낸 메시지: 1 - Send Message
서버로부터 받은 메시지: Re:1 - Send Message
서버에 보낸 메시지: 2 - Send Message
서버로부터 받은 메시지: Re:2 - Send Message
서버에 보낸 메시지: 3 - Send Message
서버로부터 받은 메시지: Re:3 - Send Message
서버에 보낸 메시지: 4 - Send Message
서버로부터 받은 메시지: Re:4 - Send Message
서버에 보낸 메시지: 5 - Send Message
서버로부터 받은 메시지: Re:5 - Send Message
서버에 보낸 메시지: 6 - Send Message
서버로부터 받은 메시지: Re:6 - Send Message
```

(클라이언트)

예제 3-1과 예제 3-2에서 적은 양의 소스 코드로 서버와 클라이언트 프로그램을 만들 수 있었던 이유는 Boost.Asio를 사용하여 진행했기 때문이었다. 여기까지는 대부분 이해했으리라 생각하지만, 이어지는 내용을 이해하려면 지금까지 설명한 내용을 반드시 알아야 한다. 그림 3-3은 지금까지 설명한 내용의 전체적인 흐름을

나타낸 것이다. 이를 통해 내용을 정리해보고 만약 이해되지 않는 부분이 있다면 관련 내용을 다시 한 번 정독하기 바란다.

그림 3-3 동기 버전 Echo 서버/클라이언트 흐름도



이번 장을 통해 Boost.Aasio를 사용한 네트워크 프로그래밍을 간단히 맛보았다. Windows나 Linux의 OS에서 제공하는 API로 프로그래밍하는 것보다 더 간단하고 체계적이지 않은가.

3.4 관련 Boost.Aasio API

예제에서 사용한 주요 Boost.Aasio API를 살펴보자. 핵심적인 부분만 간략하게 설명했으니, 관련 내용을 숙지하여 사용하기 바란다.

| **boost::asio::ip::tcp::endpoint** 클래스

프로그램에서 사용할 네트워크 주소를 설정한다. 생성자 버전은 네 가지가 있다. 3장의 예제에서 서버는 두 번째 버전(②), 클라이언트는 세 번째 버전(③)을 사용했다.

① basic_endpoint():

② basic_endpoint(const InternetProtocol & internet_protocol, unsigned short port_num);

예) TCP 프로토콜의 IPv6 주소 형식, 포트 번호는 14567

boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::tcp::v6(),
14567);

예) UDP 프로토콜의 IPv4 주소 형식, 포트 번호는 24567

boost::asio::ip::udp::endpoint endpoint(boost::asio::ip::udp::v4(),
24567);

③ basic_endpoint(const boost::asio::ip::address & addr, unsigned short port_num);

예) TCP 프로토콜의 IPv4 주소 형식에 주소는 127.0.0.1, 포트 번호는 14567

boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::address::from_string("127.0.0.1"), 14567);

④ basic_endpoint(const basic_endpoint & other);

• internet_protocol: 프로토콜 타입 및 주소 형식을 지정

• addr: (접속할 서버의) IP 주소

• port_num: 포트 번호

| **boost::asio::ip::tcp::acceptor** 클래스

클라이언트의 접속을 받아들이기 위한 클래스로, 생성자 버전은 다섯 가지가 있다.
이 중 두 번째와 세 번째 버전을 가장 많이 사용한다.

- ① basic_socket_acceptor(boost::asio::io_service & io_service);
 - ② basic_socket_acceptor(boost::asio::io_service & io_service, const protocol_type & protocol);
 - ③ basic_socket_acceptor(boost::asio::io_service & io_service, const endpoint_type & endpoint, bool reuse_addr = true);
 - ④ basic_socket_acceptor(boost::asio::io_service & io_service, const protocol_type & protocol, const native_handle_type & native_acceptor);
 - ⑤ basic_socket_acceptor(basic_socket_acceptor && other);
- protocol: 사용할 프로토콜 TCP(v4, v6), UDP(v4, v6)
 - endpoint_type: 접속을 받아들일 주소
 - reuse_addr: 만약 다른 프로그램에서 이미 사용 중인 주소라면 재설정한다.⁰⁵

세 번째 버전은 두 번째 버전을 더 쉽게 사용하기 위해 업데이트된 버전으로, 두 번째 버전에서 직접 설정해야 했던 것들을 자동으로 설정해준다. 예제 3-1에서 다음과 같이 세 번째 버전을 사용했다.

```
| boost::asio::ip::tcp::acceptor acceptor( io_service, endpoint );
```

이것을 첫 번째 버전으로 바꾸면 다음과 같다.

```
boost::asio::ip::tcp::acceptor acceptor(io_service);
```

05 주소를 사용한 프로그램이 종료되었지만 커널에서 정리되지 않았을 수 있다. 이때 앞서 사용한 것을 무시하고 주소를 재설정한다.

```
acceptor.open(endpoint.protocol());  
  
if (reuse_addr)  
    acceptor.set_option(socket_base::reuse_address(true));  
  
acceptor.bind(endpoint);  
acceptor.listen(listen_backlog);
```

세 번째 버전을 사용하면 일반적으로 추천되는 값을 사용하여 acceptor를 생성하고 설정하므로, 두 번째 코드에 있는 boost::asio::ip::tcp::acceptor acceptor 이후의 코드가 필요 없다. 두 번째 버전을 사용하면, listen의 '백로그'⁰⁶ 값을 원하는 값으로 설정할 수 있다.

| boost::asio::ip::tcp::acceptor 클래스의 accept 함수

클라이언트의 연결 요청을 받아들인다. 새로운 연결 요청이 있을 때까지 대기하며, accept 과정이 끝나야 연결된 클라이언트와 데이터를 주고받을 수 있다. 생성자 버전은 네 가지가 있는데, 첫 번째와 두 번째 버전을 많이 사용한다. 첫 번째와 두 번째 버전의 차이는 인자로 에러 코드를 사용하느냐 하지 않느냐의 여부다. 인자로 에러 코드를 사용하는 것과 사용하지 않는 버전이 있는 경우, 에러 코드를 사용하지 않는 버전에서는 보통 에러 발생 시 예외가 발생하므로 예외 처리를 꼭 해줘야 한다.

① template<typename SocketService>

```
void accept(basic_socket< protocol_type, SocketService > & peer);
```

② template<typename SocketService>

```
boost::system::error_code accept(basic_socket< protocol_type,
```

06 <http://blog.naver.com/lig1111/100157952465>

```
SocketService > & peer, boost::system::error_code & ec);
```

③ template<typename SocketService>

```
void accept(basic_socket< protocol_type, SocketService > & peer,  
endpoint_type & peer_endpoint);
```

④ template<typename SocketService>

```
boost::system::error_code accept(basic_socket< protocol_type,  
SocketService > & peer, endpoint_type & peer_endpoint, boost::  
system::error_code & ec);
```

- peer: 새로 연결된 클라이언트에게 할당할 소켓 클래스
- ec: 에러 코드
- peer_endpoint: 새로 연결된 클라이언트의 주소

예) TCP 프로토콜을 사용하는 클라이언트의 연결을 받아들인다.

```
boost::asio::ip::tcp::socket socket(io_service);  
acceptor.accept(socket);
```

| boost::asio::ip::tcp::socket 클래스의 connect 함수

서버와 네트워크로 연결하기 위해 접속 요청을 하며, 서버에 접속될 때까지 더는 진행하지 않고 기다린다. 생성자 버전은 두 가지가 있으며 에러 코드 사용 여부가 차이 난다.

① void connect(const endpoint_type & peer_endpoint);

② boost::system::error_code connect(const endpoint_type
& peer_endpoint, boost::system::error_code & ec);

- peer_endpoint: 접속할 서버의 주소

- ec: 에러 코드

다음은 서버에 접속 요청을 하는 예다. 에러가 발생했다면(접속 실패) 에러 처리를 한다.

```
boost::system::error_code connect_error;
boost::asio::ip::tcp::socket socket(io_service);

socket.connect(endpoint, connect_error);

if (connect_error)
{
    // 에러 처리
}
```

| [boost::asio::ip::tcp::socket 클래스의 read_some 함수](#)

연결된 곳에서 보내는 데이터를 받으면, 에러가 발생하거나 데이터를 받을 때까지 대기한다. 두 가지 버전이 있는데, 첫 번째 버전은 에러가 발생하거나 연결이 끊어지면 예외가 발생한다. 두 번째 버전은 예제 코드에서 사용한 것으로, 데이터를 받다가 에러가 발생하면 ec를 통해서 어떤 에러인지 알 수 있다. 주로 두 번째 버전을 많이 사용한다.

① template<typename MutableBufferSequence>

```
std::size_t read_some(const MutableBufferSequence & buffers);
```

② template<typename MutableBufferSequence>

```
std::size_t read_some(const MutableBufferSequence & buffers,
boost::system::error_code & ec);
```

- buffers: 데이터를 받을 버퍼. char 배열이나 std::array, std::vector, std::string를 사용할 수 있다.
- ec: 에러 코드. 연결이 끊어졌을 때, 에러 코드는 ‘boost::asio::error::eof’이다.

| `boost::asio::ip::tcp::socket` 클래스의 `write_some` 함수

연결한 곳에 데이터를 보내며 데이터를 다 보낼 때까지 대기한다. 두 가지 버전이 있는데, 첫 번째 버전은 에러가 발생하면 예외가 발생한다. 두 번째 버전이 앞서 예제 코드에서 사용한 것으로, 데이터를 보내다가 에러가 발생할 경우 ec를 통해 어떤 에러인지 알 수 있다. 주로 두 번째 버전을 사용한다.

- ① `template<typename ConstBufferSequence>`
`std::size_t write_some(const ConstBufferSequence & buffers);`
 - ② `template<typename ConstBufferSequence>`
`std::size_t write_some(const ConstBufferSequence & buffers,
boost::system::error_code & ec);`
- buffers: 데이터를 받을 버퍼. char 배열이나 std::array, std::vector를 사용할 수 있다.
 - ec: 에러 코드

| `boost::asio::ip::tcp::socket` 클래스의 `close` 함수

연결한 곳과의 접속을 끊는다. 두 가지 버전이 있는데, 첫 번째 버전의 경우 에러가 발생하면 예외가 발생한다. 두 번째 버전이 앞서 예제 코드에서 사용한 것으로, 접속을 끊을 때 에러가 발생하면 ec를 통해 어떤 에러인지 알 수 있다.

- ① `void close();`
- ② `boost::system::error_code close(boost::system::error_code & ec);`

- **ec:** 예제 코드

여기서 잠깐_ Boost.Asio와 BSD Socket 비교

대부분의 OS는 네트워크 프로그래밍을 위해 Socket API를 제공한다. 주로 BSD Socket과 비슷한 API를 제공하는데, 보통 처음 네트워크 프로그래밍을 공부할 때 BSD Socket API를 사용한 예제를 접하게 되고, 예제를 직접 만들어보기도 한다.

BSD Socket에 대해 잘 알고 있는 경우라면, 다음의 ‘Boost.Asio와 BSD Socket 비교표’를 통해 Boost.Asio를 좀 더 쉽게 이해할 수 있을 것이다.

- http://www.boost.org/doc/libs/1_53_0/doc/html/boost_asio/overview/networking/bsd_sockets.html

여기서 잠깐_ IPv6 주소 사용하기

현재 우리가 사용하는 대부분의 IP 주소는 IPv4 방식의 주소다. 그러나 IPv4 방식의 주소는 이미 대부분 고갈되었다. 그래서 작년부터 대형 IT 회사들은 서서히 IPv6 방식 주소를 사용하기 위해 준비 중이다. 아마 몇 년 후에는 IPv6로 완전히 넘어가야 할 텐데, 그러면 기존에 만든 프로그램도 수정해야 한다.

만약 Boost.Asio를 사용하여 네트워크 프로그램을 만들었다면 IPv6로 변환하는 데 아주 적은 수고만 들이면 된다. 가령, 예제 3-1의 서버에서 endpoint를 설정할 때 다음과 같이 IPv4 주소를 사용하였다.

```
boost::asio::ip::tcp::endpoint endpoint( boost::asio::ip::tcp::v4( ), PORT_NUMBER );
```

이것을 IPv6 주소로 변경하고 싶다면, 다음과 같이 수정하면 된다.

```
boost::asio::ip::tcp::endpoint endpoint( boost::asio::ip::tcp::v6( ), PORT_NUMBER );
```

이처럼 간단히 변경한 것만으로 프로그램은 IPv6 주소를 사용할 수 있게 되었다. 이런 부분도 Boost.Asio를 사용했기 때문에 얻을 수 있는 혜택이다.

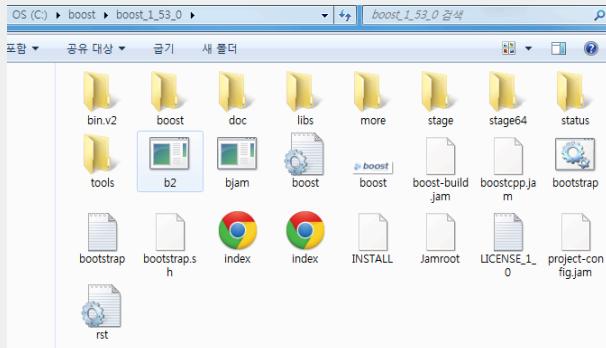
- 참고 자료: IPv6 이론과 소켓 프로그래밍

<http://www.slideshare.net/zone0000/ipv6-7542249>

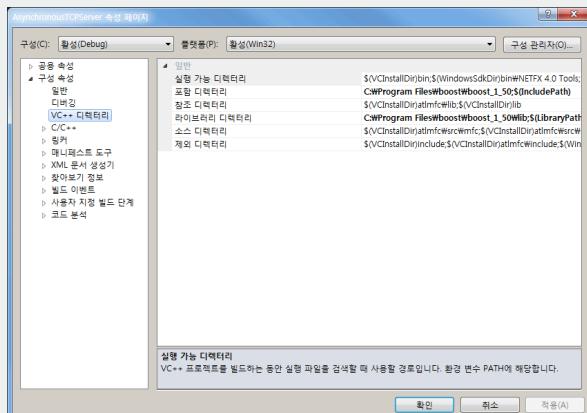
07 IPv4 주소는 32비트 길이의 식별자로, 0.0.0.0부터 255.255.255.255까지 숫자의 조합으로 이루어진다.
따라서 IPv4 주소는 총 2^{32} 개의 주소를 사용할 수 있다.

[여기서 잠깐_ Visual C++ 프로젝트에서 Boost 라이브러리 설정하기](#)

boost 라이브러리가 다음의 그림과 같은 위치에 설치되었다고 가정해보자.



Visual Studio의 속성 매니저에서 Boost의 헤더 파일과 라이브러리 위치를 지정한다. 그러면 아래와 같이 된다.



위 방식은 Visual C++의 모든 프로젝트에 다 적용한 경우다. Visual C++(VC 10 이후)에서 외부 라이브러리를 모든 프로젝트에 적용하는 방법에 대해 알고 싶다면 다음의 웹 사이트를 참고하기 바란다

- <http://iacking.tistory.com/987>

여기서 잠깐_ 런타임 라이브러리 형식 맞추기

프로젝트의 ‘런타임 라이브러리’ 형식마다 Boost 라이브러리의 lib 파일을 빌드해야 한다. VC10 이후부터는 프로젝트를 만들면 기본적으로 ‘런타임 라이브러리’ 형식은 ‘다중 스레드 디버그/릴리즈 DLL(MDd 혹은 MD)’이다. MDd나 MD 형식의 Boost 라이브러리의 lib 파일을 빌드하기 위해서 다음 명령어로 Boost 라이브러리를 빌드한다.

```
b2 --stagedir=stage --toolset=msvc-11.0 runtime-link=shared
```

4 | 비동기 I/O를 사용한 Echo 서버, 클라이언트 프로그램 만들기

3장에서는 동기 I/O 기능을 사용하여 만든 서버와 클라이언트 프로그램을 살펴보았다. 사실 Boost.Asio를 사용하는 이유는 동기 I/O 방식이 아닌 비동기 I/O 방식을 사용하는 프로그램을 만들기 위해서다. 3장에서 Boost.Asio를 이용해 어떻게 네트워크 프로그래밍을 하는지 큰 줄기를 보여주었다면, 이번 장에서는 그것을 토대로 ‘동기 I/O 기능’을 ‘비동기 I/O 기능’으로 바꾸는 방법을 보여줄 것이다. Boost.Asio의 새로운 기능을 배우는 데 집중하기 위해, 이번 장에서도 예제 코드는 아주 단순하게 만들었다. 이것을 토대로 쓸만한 프로그램을 만드는 과정에 대해서는 다음 장에서 알아볼 것이니, 너무 조급해하지 말고 이번장을 잘 따라오기 바란다.

4.1 비동기 I/O 프로그래밍의 특징

병렬 프로그래밍과 근래의 모바일 프로그래밍에서 I/O와 관련된 작업(네트워크나 파일 조작)을 할 때, 비동기 방식을 사용한다는 말을 자주 들었을 것이다. 비동기의 특징은 가령, A라는 작업을 요청하면 작업이 완료될 때까지 기다리지 않고 다음 작업을 진행하다가, 요청한 작업 A가 끝나면 완료 통보를 받은 후 관련된 작업을 하는 방식이다. 그래서 보통 비동기 기능을 사용하는 함수에는 ‘async’를 표기하고 요청한 작업이 끝났을 때 불러줄 함수를 설정한다. 다음과 같은 방식으로 설정하면 된다.

```
async_XXX(요청 작업에 사용할 값, 요청한 작업(async_read)이 끝나면 호출할 함수)  
async_read(Buffer, Session::OnRead)
```

위의 코드는 비동기로 데이터 읽기를 요청하는데, 읽은 데이터는 Buffer라는 곳에

저장하고 데이터 읽기가 끝나면 Session 클래스의 OnRead라는 멤버 함수를 호출한다.

여기서 잠깐_ 동기와 비동기

동기와 비동기를 카페에서 커피를 주문하는 것과 비교를 하면 다음 그림과 같다.



(a) 동기



(b) 비동기

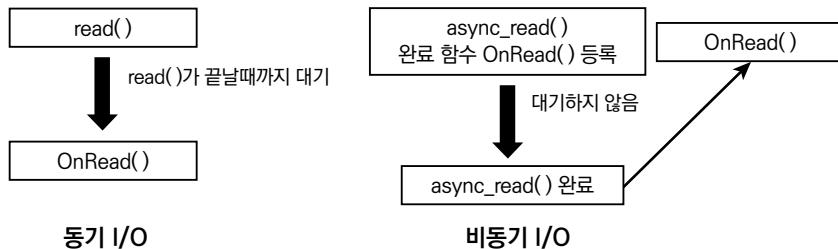
(a) 그림의 가게에서는 동기식으로 커피 주문을 받아서 손님이 주문 후 커피가 나올 때까지 그 자리에서 기다려야 한다. 그러나 (b) 그림의 가게에서는 비동기식으로 주문을 받아서 손님은 주문 후 자리로 돌아가서 볼일을 보고, 커피가 완성된 후 점원이 불러주면 가서 커피를 받아오면 된다.

(b) 가게처럼 비동기식으로 커피 주문을 받으면 (a)가게에 비해 한번에 더 많은 커피 주문을 받을 수 있고, 손님은 주문을 하고 끝날 때까지 멍하니 기다릴 필요가 없어서 자신의 시간을 더 효율적으로 사용할 수 있다.

프로그램도 이와 같이 비동기식으로 작업을 처리하면 작업 흐름에 대기가 없이 신속하고 효율적으로 작업을 처리할 수 있다.

async_read 함수를 호출하면 바로 다음 코드를 진행하며, 다른 작업을 하는 도중 async_read 작업이 완료되면 async_read 작업을 처리한 곳에서 OnRead 함수를 호출한다.

그림 4-1 동기 I/O(블록킹)와 비동기 I/O의 차이



4.2 비동기 I/O 방식의 TCP/IP Echo 서버

3장의 예제 3-1과 기능은 거의 같으나, 네트워크 I/O를 비동기 방식으로 처리하기 위해 Boost.Asio의 비동기 함수를 사용한다는 점이 다르다. 그래서 필자는 예제 3-1과 같은 부분은 넘기고 Boost.Asio의 비동기 함수 사용 부분을 집중적으로 살펴볼 것이다.

예제 4-1 AsynchronousTCPServer

```
#include <SDKDDKVer.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <list>
#include <boost/bind.hpp>
#include <boost/asio.hpp>

class Session
```

```

{
public:
    Session(boost::asio::io_service& io_service)
        : m_Socket(io_service)
    {
    }

    boost::asio::ip::tcp::socket& Socket()
    {
        return m_Socket;
    }

    void PostReceive()
    {
        memset( &m_ReceiveBuffer, '\0', sizeof(m_ReceiveBuffer) );
        m_Socket.async_read_some
        (
            boost::asio::buffer(m_ReceiveBuffer),
            boost::bind( &Session::handle_receive, this,
                        boost::asio::placeholders::error,
                        boost::asio::placeholders::bytes_transferred )
        );
    }
}

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

    void handle_receive( const boost::system::error_code& error,
                        size_t bytes_transferred )

```

```

{
    if( error )
    {
        if( error == boost::asio::error::eof )
        {
            std::cout << "클라이언트와 연결이 끊어졌습니다" << std::endl;
        }
        else
        {
            std::cout << "error No: " << error.value() << " error Message: "
                << error.message() << std::endl;
        }
    }
    else
    {
        const std::string strRecvMessage = m_ReceiveBuffer.data();
        std::cout << "클라이언트에서 받은 메시지: " << strRecvMessage
            << ", 받은 크기: " << bytes_transferred << std::endl;

        char szMessage[128] = {0,};
        sprintf_s( szMessage, 128-1, "Re:%s", strRecvMessage.c_str() );
        m_WriteMessage = szMessage;

        boost::asio::async_write(m_Socket, boost::asio::buffer(m_WriteMessage),
            boost::bind( &Session::handle_write, this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred )
        );
    }
}

```

```

        boost::asio::ip::tcp::socket m_Socket;
        std::string m_WriteMessage;
        std::array<char, 128> m_ReceiveBuffer;
    };

const unsigned short PORT_NUMBER = 31400;

class TCP_Server
{
public:
    TCP_Server( boost::asio::io_service& io_service )
        : m_acceptor(io_service,
            boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(),
                PORT_NUMBER))
    {
        m_pSession = nullptr;
        StartAccept();
    }

    ~TCP_Server()
    {
        if( m_pSession != nullptr )
        {
            delete m_pSession;
        }
    }
}

private:
    void StartAccept()
    {
        std::cout << "클라이언트 접속 대기....." << std::endl;
    }
};

```

```

    m_pSession = new Session(m_acceptor.get_io_service());
    m_acceptor.async_accept( m_pSession->Socket(),
        boost::bind(&TCP_Server::handle_accept,
            this,
            m_pSession,
            boost::asio::placeholders::error)
    );
}

void handle_accept(Session* pSession, const boost::system::error_code& error)
{
    if (!error)
    {
        std::cout << "클라이언트 접속 성공" << std::endl;

        pSession->PostReceive();
    }
}

int m_nSeqNumber;
boost::asio::ip::tcp::acceptor m_acceptor;
Session* m_pSession;
};

int main()
{
    boost::asio::io_service io_service;
    TCP_Server server(io_service);
    io_service.run();

    std::cout << "네트워크 접속 종료" << std::endl;
}

```

```
    getchar();
    return 0;
}
```

예제 4-1의 코드는 크게 Session 클래스와 TCP_Server 클래스로 나뉜다. Session 클래스는 서버에 접속한 클라이언트를 뜻하며, 클라이언트가 보낸 데이터를 받고 클라이언트에 데이터를 보내기도 한다. TCP_Server는 Session 클래스를 가지고 있으면서 클라이언트의 접속을 받아들이는 기능을 담당한다.

예제 4-1은 3장의 예제 3-1보다 코드가 3배 정도 많지만, 전체적으로 크게 달라지거나 더 복잡해진 것은 없다. 이 둘의 차이는 단지 Boost.Asio의 동기 함수를 사용했느냐 비동기 함수를 사용했느냐 여부다.

보통 서버의 네트워크 기능은 클라이언트 접속 받기, 데이터 받기, 데이터 보내기 등이다. 예제 4-1에서는 Boost.Asio를 사용하여 이런 서버의 네트워크 기능을 비동기로 처리했다.

4.2.1 클라이언트 접속 받기

```
void StartAccept()
{
    .....
    m_acceptor.async_accept( m_pSession->Socket(),
        boost::bind(&TCP_Server::handle_accept,
            this,
            m_pSession,
            boost::asio::placeholders::error)
    );
}
```

Boost.Asio의 acceptor 클래스의 멤버 중 async_accept(예제 3-1에서 사용한 accept의 비동기 버전)를 사용하여 비동기로 접속을 처리한다. async_accept의 첫 번째 인자는 접속한 클라이언트에 할당할 소켓 클래스다. 두 번째 인자는 Boost 라이브러리의 bind 함수를 사용하여, async_accept가 완료되면 호출할 함수를 맵핑하였다.

async_accept를 호출하면 접속이 끝날 때까지 대기하지 않고 바로 다음 단계로 넘어간다. async_accept 작업이 끝나면 두 번째 인자로 넘겨준 함수가 호출되어 handle_accept가 호출된다. 즉, handle_accept가 호출되면 접속 처리 작업이 완료되었다는 뜻이다.

```
void handle_accept(Session* pSession, const boost::system::error_code& error)
{
    if (!error)
    {
        std::cout << "클라이언트 접속 성공" << std::endl;
        pSession->PostReceive();
    }
}
```

handle_accept에서 주의할 점이 있다. 접속이 성공했을 때 접속한 클라이언트가 보내는 패킷을 받을 수 있도록, accept된 받은 소켓(Session 클래스의 m_Socket)에 받기 작업을 요청해야 한다는 것이다.

pSession->PostReceive()

예제 4-1은 Session 클래스의 PostReceive() 함수를 통해서 받기 작업을 요청한다. 이 작업을 하지 않으면 지금 접속한 클라이언트가 보내는 데이터를 받지 못한다.

4.2.2 클라이언트가 보내는 데이터 받기

```
void PostReceive()
{
    ....
    m_Socket.async_read_some( boost::asio::buffer(m_ReceiveBuffer), // ❶
        boost::bind( &Session::handle_receive, this, // ❷
            boost::asio::placeholders::error,
            boost::asio::placeholders::bytes_transferred )
    );
}
```

Boost.Asio의 socket 클래스의 멤버 중 async_read_some(예제 3-1에서 사용한 read_some의 비동기 버전)를 사용하여, 접속한 클라이언트가 보내는 데이터를 비동기로 받는다.

async_read_some의 첫 번째 인자(①)는 클라이언트가 보낸 데이터를 받을 버퍼고, 두 번째 인자(②)는 async_read_some이 완료되면 호출할 함수다. 앞에서 설명한 async_accept와 방식이 같다. 즉, async_read_some 역시 호출하면 데이터를 받을 때까지 기다리지 않고 다음으로 넘어간 후, 데이터를 받으면 두 번째 인자로 지정한 함수를 호출한다.

```
void handle_receive( const boost::system::error_code& error, size_t
    bytes_transferred )
{
    if( error )
    {
        ....
```

```
    }
else
{
    .....
    PostReceive();
}
}
```

handle_receive는 에러 코드를 조사하여 에러가 발생했다면(접속이 끊긴 것도 포함) 에러 처리를 하고, 에러가 없으면 받은 데이터를 처리한다. 여기서도 역시 PostReceive() 함수를 호출하여 다시 async_read_some 함수를 호출한다는 점을 잊지 말아야 한다. async_read_some 함수를 호출해야 클라이언트가 보내는 데이터를 다시 받을 수 있다.

접속 받기와 데이터 받기를 요청해야 클라이언트의 접속 요청을 받아들이고 클라이언트가 보낸 데이터를 받을 수 있다. 앞의 요청이 완료되면 또다시 새로운 요청을 해야 한다.

4.2.3 클라이언트에게 데이터 보내기

void handle_receive()를 보면 클라이언트에서 받은 패킷을 다시 보낸다.

```
void handle_receive( const boost::system::error_code& error, size_t
bytes_transferred )
{
    if( error )
    {
        ....
    }
    else
```

```

{
    ....
    boost::asio::async_write( m_Socket, boost::asio::buffer(m_WriteMessage),
        boost::bind( &Session::handle_write, this,
        boost::asio::placeholders::error,
        boost::asio::placeholders::bytes_transferred )
    );
}

.....
}
}

```

Boost.Asio의 함수 중 전역함수인 `async_write`를(예제 3-1에서 사용한 `write_some`의 비동기 버전) 사용하여, 데이터를 보낸 클라이언트에게 비동기로 데이터를 보낸다.

그림 4-2 `async_write` 함수의 인자 값 설명

boost::asio::async_write(m_Socket,	인자 1. 클라이언트 소켓
boost::asio::buffer(m_WriteMessage),	인자 2. 보낼 데이터를 담은 버퍼
boost::bind(&Session::handle_write, this, boost::asio::placeholders::error, boost::asio::placeholders::bytes_transferred));	인자 3. 전송이 완료되면 호출할 함수

`async_write`의 첫 번째 인자는 클라이언트 소켓, 두 번째 인자는 클라이언트에 보낼 데이터를 담은 버퍼, 세 번째 인자는 `async_write`가 완료되면 호출될 함수다.

그림 4-3 bind 함수의 인자 값 설명

```
boost::bind(&Session::handle_write,  
           this,  
           boost::asio::placeholders::error,  
           boost::asio::placeholders::bytes_transferred)  
  
인자 1. 완료하면 호출할 함수  
인자 2. 함수를 멤버로 가지는 클래스(Session)의 인스턴스  
인자 3. handle_write 함수에 넘길 첫 번째 인자. 에러 코드  
인자 4. handle_write 함수에 넘길 두 번째 인자.  
보낸 데이터 크기  
  
void handle_write(const boost::system::error_code& error, size_t bytes_transferred)
```

지금 살펴보고 있는 예제에서는 데이터를 보낸 후 함수만 호출할 뿐 다른 작업은 하지 않는데, 좀 더 쓸만한 프로그램을 만들고자 한다면 여기서도 몇 가지 해야 할 일이 있다. 이 내용은 5장에서 다시 설명하겠다.

이상으로 예제 4-1의 핵심 기능을 살펴보았다. 그런데 비동기 함수를 사용하는 경우 요청한 작업(접속 받기, 데이터 받기, 데이터 보내기)이 완료되면 지정한 완료 함수를 누가 호출하는지 궁금하지 않은가? 이것은 Boost.Aasio의 심장이라고 할 수 있는 io_service가 한다. io_service는 Boost.Aasio의 동기 버전에서는 존재감이 거의 없지만, 비동기 버전에서는 io_service야말로 주인공이라 할 수 있다.

4.2.4 io_service.run()

예제 4-1의 main() 함수 부분을 살펴보자.

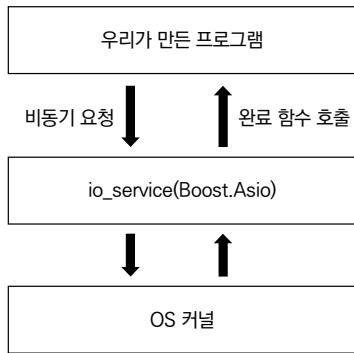
```
int main()  
{  
    boost::asio::io_service io_service;  
    TCP_Server server(io_service);  
    io_service.run();  
    std::cout << "네트워크 접속 종료" << std::endl;  
    getchar();  
    return 0;  
}
```

여기서 3장에서 살펴본 동기 방식의 서버와 다른 점을 하나 찾을 수 있다. 바로 ‘io_service.run()’이다. io_service 클래스의 run() 함수를 호출하면 main() 함수에서 무한 대기 상태가 되고, 우리가 사용한 Boost.Asio의 비동기 함수 작업이 끝나면 비동기 함수와 연결된 완료 함수를 호출한다. run() 함수는 비동기 요청이 있을 경우 요청이 끝날 때까지 무한히 대기하다가, 다음 요청이 있는지 보고 없으면 run() 함수를 빠져나와서 다음 작업을 진행한다. run() 함수의 동작을 의사 코드로 표현하면 다음과 같다.

```
void run()
{
    while(true)
    {
        if( CompleteQueue.empty() )
            break;
        .....
        if(CompleteQueue.IsComplete ) // 완료되었다면
        {
            등록된 완료 함수 호출
        }
        .....
    }
}
```

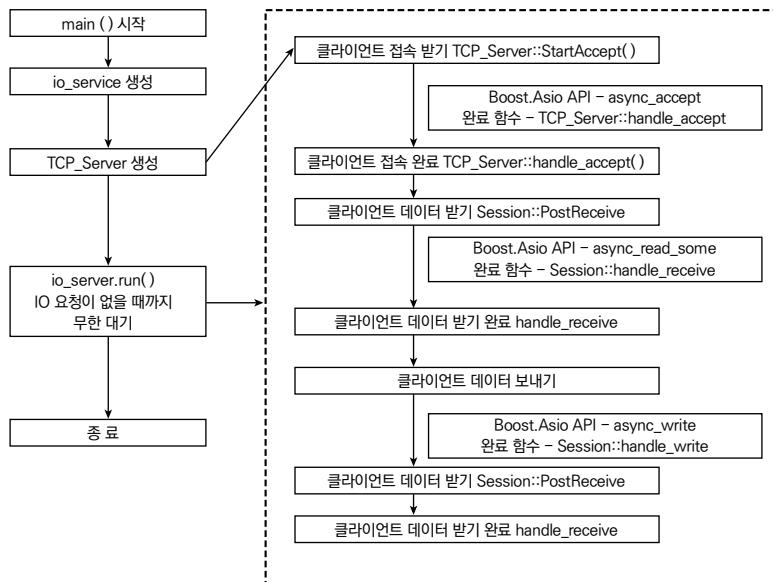
그래서 비동기 요청을 하기 전에 io_service의 run() 함수를 호출하면 run() 함수는 곧바로 종료된다. 또한 비동기 요청이 완료되었을 때 새로운 비동기 요청을 하지 않으면 run 함수는 종료된다. 그림 4-4는 io_service의 비동기 요청과 처리의 흐름도다.

그림 4-4 io_service의 비동기 요청/완료 처리



예제 4-1의 이해를 돋기 위해 전체적인 흐름을 정리했으니, 아직도 소스 코드가 이해되지 않는다면 그림 4-5의 흐름도를 순서대로 따라가보기 바란다.

그림 4-5 비동기 Echo 서버 흐름도



4.3 비동기 방식의 TCP/IP Echo 클라이언트

예제 4-1의 소스 코드를 이해했다면, 지금부터 살펴볼 클라이언트의 소스 코드는 사실 특별한 것이 거의 없다. 서버에 접속해서 데이터를 보내고 받는 기능적인 부분밖에 없으니 말이다(예제 3-2와 같다). 클라이언트도 Boost.Asio의 비동기 함수를 사용하는데, 서버에 접속하는 함수만 다를 뿐 데이터를 보내고 받는 것은 서버와 같다.

예제 4-2 AsynchronousTCPClient

```
#include <SDKDDKVer.h>
#include <iostream>
#include <boost/bind.hpp>
#include <boost/asio.hpp>

const char SERVER_IP[] = "127.0.0.1";
const unsigned short PORT_NUMBER = 31400;

class TCP_Client
{
public:
    TCP_Client(boost::asio::io_service& io_service)
        : m_io_service(io_service),
        m_Socket(io_service),
        m_nSeqNumber(0)
    {}

    void Connect( boost::asio::ip::tcp::endpoint& endpoint )
    {
        m_Socket.async_connect( endpoint,
            boost::bind(&TCP_Client::handle_connect,
            this,

```

```

                boost::asio::placeholders::error)
            );
        }

private:
    void PostWrite()
    {
        if( m_Socket.is_open() == false )
        {
            return;
        }

        if( m_nSeqNumber > 7 )
        {
            m_Socket.close();
            return;
        }

        ++m_nSeqNumber;

        char szMessage[128] = {0,};
        sprintf_s( szMessage, 128-1, "%d - Send Message", m_nSeqNumber );

        m_WriteMessage = szMessage;

        boost::asio::async_write(m_Socket, boost::asio::buffer(m_WriteMessage),
            boost::bind( &TCP_Client::handle_write, this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred
            );
    }

    PostReceive();
}

```

```

void PostReceive()
{
    memset( &m_ReceiveBuffer, '\0', sizeof(m_ReceiveBuffer) );

    m_Socket.async_read_some( boost::asio::buffer(m_ReceiveBuffer),
        boost::bind( &TCP_Client::handle_receive, this,
                    boost::asio::placeholders::error,
                    boost::asio::placeholders::bytes_transferred )
    );
}

void handle_connect(const boost::system::error_code& error)
{
    if (error)
    {
        std::cout << "connect failed : " << error.message() << std::endl;
    }
    else
    {
        std::cout << "connected" << std::endl;
        PostWrite();
    }
}

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

void handle_receive( const boost::system::error_code& error, size_t
bytes_transferred )
{
}

```

```

    if( error )
    {
        if( error == boost::asio::error::eof )
        {
            std::cout << "서버와 연결이 끊어졌습니다" << std::endl;
        }
        else
        {
            std::cout << "error No: " << error.value() << " error Message: "
                << error.message() << std::endl;
        }
    }
    else
    {
        const std::string strRecvMessage = m_ReceiveBuffer.data();
        std::cout << "서버에서 받은 메시지: " << strRecvMessage
            << ", 받은 크기: " << bytes_transferred << std::endl;
        PostWrite();
    }
}

boost::asio::io_service& m_io_service;
boost::asio::ip::tcp::socket m_Socket;
int m_nSeqNumber;
std::array<char, 128> m_ReceiveBuffer;
std::string m_WriteMessage;
};

int main()
{
    boost::asio::io_service io_service;
    boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::
        address::from_string(SERVER_IP), PORT_NUMBER);

```

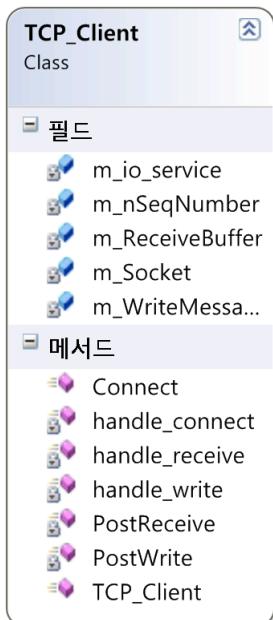
```

TCP_Client client(io_service);
client.Connect( endpoint );
io_service.run();
std:: cout << "네트워크 접속 종료" << std::endl;
getchar();
return 0;
}

```

그림 4-6은 Echo 비동기 Echo 클라이언트에서 구현한 TCP_Client 클래스의 전체적인 구성이다.

그림 4-6 TCP_Client 클래스의 구성



4.3.1 서버에 접속하기

클라이언트는 서버에 접속하기 위해서 TCP_Client 클래스의 Connect 함수를 사용한다.

```
void Connect( boost::asio::ip::tcp::endpoint& endpoint )
{
    m_Socket.async_connect( endpoint,
        boost::bind(&TCP_Client::handle_connect,
            this,
            boost::asio::placeholders::error)
    );
}
```

비동기 접속을 위해서는 Boost.Asio의 async_connect(예제 3-2에서 사용한 connect의 비동기 버전)를 사용한다.

```
void Connect(boost::asio::ip::tcp::endpoint& endpoint)
{
    m_Socket.async_connect( endpoint,      // 인자 1. 접속할 서버 주소
        // 인자 2. 접속 완료 후 호출할 함수
        boost::bind(&TCP_Client::handle_connect,
            this,
            boost::asio::placeholders::error)
    );
}
```

async_connect의 첫 번째 인자는 접속할 서버의 주소, 두 번째 인자는 서버 접속을 완료하면 호출할 함수다.

```
boost::bind(&TCP_Client::handle_connect, // 인자 2. 접속 완료 후 호출할 함수  
           this,  
           boost::asio::placeholders::error)  
void handle_connect(const boost::system::error_code& error)
```

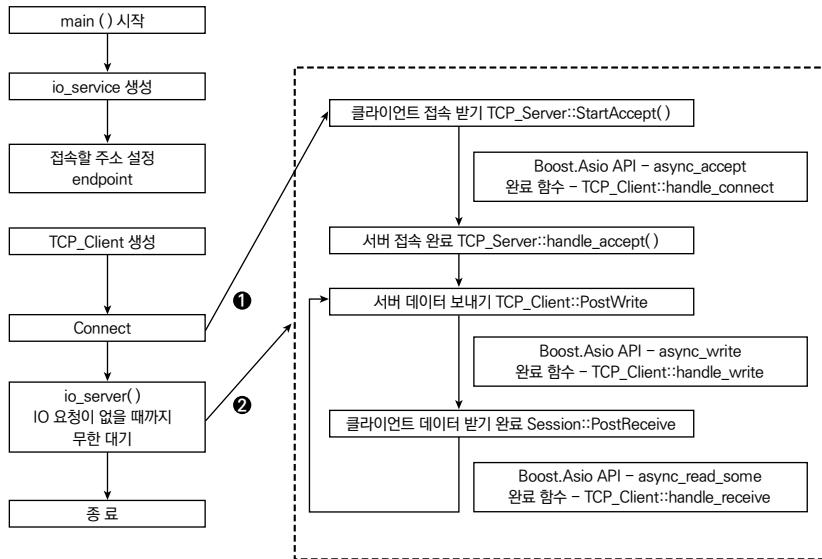
```
boost::bind( &TCP_Client::handle_connect,  
            this,  
            boost::asio::placeholders::error )  
인자 1. 원료하면 호출할 함수  
인자 2. 함수를 멤버로 가지는 클래스(TCP_Client)의 인스턴스  
인자 3. 에러 코드  
void handle_connect(const boost::system::error_code& error)
```

async_connect는 비동기 함수이기 때문에 접속을 시도하면 대기하지 않고 바로 다음으로 넘어가며, 접속이 완료되면 async_connect에서 두 번째 인자로 넘겨준 함수가 호출된다. 이 예제에서는 완료 함수인 handle_connect에서 접속이 성공할 경우 서버에 데이터를 보낸다.

예제 4-2에서 사용할 Boost.Asio 기능의 대부분은 이미 예제 4-1에서 다 설명해 서 더 이상 할 것이 없다. 다만, 앞서도 이야기했지만 io_service.run()을 호출하기 전에 비동기 요청이 있어야 한다는 점은 주의한다. 이 때문에 예제 4-2에서는 io_service.run()을 호출하기 전에 async_connect 함수를 호출했다는 점을 잘 기억하기 바란다.

예제 4-2의 전체적인 흐름을 보다 잘 이해할 수 있도록 프로그램의 흐름을 정리하였으니, 참고하기 바란다.

그림 4-7 비동기 Echo 클라이언트 흐름도



- ① Connect 후 io_service_run을 호출해야, 서버와 접속될 때 Asio에서 TCP_Client 클래스의 Connect 함수를 호출한다.
- ② Asio에서 각각의 비동기 요청에(접속, 데이터 받기/보내기) 등록한 함수를 호출 한다.

예제 4-1과 예제 4-2는 프로그램으로서는 쓸만하지 않지만, Boost.Asio의 비동기 기능은 잘 보여준다. 예제가 간단한 만큼, 꼭 확실히 이해하고 다음 장으로 넘어가기 바란다.

그림 4-8 서버와 클라이언트 실행 결과

The image shows two windows side-by-side. The left window is titled 'C:\exam_boost\AsynchronousTCPserver\Debug\AsynchronousTCPserver.exe' and contains the following text:
클라이언트 접속 대기.....
클라이언트 접속 성공
클라이언트에서 받은 메시지: 1 - Send Message, 받은 크기: 16
클라이언트에서 받은 메시지: 2 - Send Message, 받은 크기: 16
클라이언트에서 받은 메시지: 3 - Send Message, 받은 크기: 16
클라이언트에서 받은 메시지: 4 - Send Message, 받은 크기: 16
클라이언트에서 받은 메시지: 5 - Send Message, 받은 크기: 16
클라이언트에서 받은 메시지: 6 - Send Message, 받은 크기: 16
클라이언트에서 받은 메시지: 7 - Send Message, 받은 크기: 16
클라이언트에서 받은 메시지: 8 - Send Message, 받은 크기: 16
클라이언트와 연결이 끊어졌습니다
네트워크 접속 종료

The right window is titled 'C:\exam_boost\AsynchronousTCPClient\Debug\AsynchronousTCPClient.exe' and contains the following text:
connected
서버에서 받은 메시지: Re:1 - Send Message, 받은 크기: 19
서버에서 받은 메시지: Re:2 - Send Message, 받은 크기: 19
서버에서 받은 메시지: Re:3 - Send Message, 받은 크기: 19
서버에서 받은 메시지: Re:4 - Send Message, 받은 크기: 19
서버에서 받은 메시지: Re:5 - Send Message, 받은 크기: 19
서버에서 받은 메시지: Re:6 - Send Message, 받은 크기: 19
서버에서 받은 메시지: Re:7 - Send Message, 받은 크기: 19
서버에서 받은 메시지: Re:8 - Send Message, 받은 크기: 19
네트워크 접속 종료

4.4 관련 Boost.Asio API

예제에서 사용한 주요 Boost.Asio API를 살펴보자. 핵심적인 부분만 간략히 설명 했으니, 관련 내용을 숙지하여 사용하기 바란다.

| `boost::asio::ip::tcp::acceptor` 클래스의 `async_accept`

새로운 연결을 받아들인다. 새로운 연결이 있을 때까지 대기하지 않고 접속이 완료 되면 완료 함수를 호출한다. 두 가지 버전이 있다.

```

① template<typename SocketService, typename AcceptHandler>
void async_accept(basic_socket< protocol_type, SocketService >
& peer, AcceptHandler handler);

② template<typename SocketService, typename AcceptHandler>
void async_accept(basic_socket< protocol_type, SocketService > & peer,
endpoint_type & peer_endpoint, AcceptHandler handler);

```

- peer : 새로 연결된 클라이언트에 할당할 소켓 클래스
- peer_endpoint : 새로 연결된 클라이언트의 주소
- handler : 연결이 완료되면 호출할 함수

| boost::asio::ip::tcp::socket 클래스의 async_connect

서버에 접속한다. 연결할 때까지 대기하지 않으며, 연결이 완료되면 완료 함수를 호출한다.

```

template<typename ConnectHandler>
void async_connect(const endpoint_type & peer_endpoint,
ConnectHandler handler);

```

- peer_endpoint : 접속할 서버의 주소
- handler : 연결이 완료되면 호출할 함수

| boost::asio::ip::tcp::socket 클래스의 async_read_some

연결된 곳에서 보낸 데이터를 받는다. 에러가 발생하거나 데이터를 받을 때까지 대기하지 않는다. 에러가 발생하거나 데이터를 받으면 완료 함수를 호출한다.

```

template<typename MutableBufferSequence, typename ReadHandler>
void async_read_some(const MutableBufferSequence & buffers, ReadHandler

```

```
    handler);
```

- buffers: 데이터를 받을 버퍼. char 배열이나 std::array, std::vector, std::string 를 사용할 수 있다.
- handler: 완료되면 호출할 함수

| **boost::asio 전역함수 async_write**

연결된 곳에 데이터를 보낸다. 데이터를 다 보낼 때까지 대기하지 않으며, 데이터를 다 보내면 완료 함수를 호출한다. Boost.Aasio의 전역함수로 네 가지 버전⁰¹이 있는데, 이번 예제에서 사용한 것은 다음 버전이다.

```
template<typename AsyncWriteStream, typename ConstBufferSequence,
          typename WriteHandler>
void async_write(AsyncWriteStream & s, const ConstBufferSequence
& buffers, WriteHandler handler);
```

- s : 데이터를 보낼 곳
- buffers : 보낼 데이터를 담은 버퍼
- handler : 완료되면 호출할 함수

async_write에서 주의해야 할 것이 있다. async_write가 완료될 때까지(즉, 완료 함수가 호출될 때까지) buffers의 데이터를 잘 보관해야 한다는 점이다. async_write 호출 즉시 데이터가 보내진 것이 아니라, 완료 함수가 호출되어야 데이터가 다 보내진 것이기 때문이다.

async_read_some은 socket 클래스의 멤버를 사용했는데 async_write는 전역 함수를 사용했다. socket 클래스의 비동기 write 함수인 async_write_some의

01 여기서 설명한 것 이외의 버전은 거의 사용하지 않아서 따로 설명하지 않았다.

경우 보내려는 데이터를 다 보내지 못해도 완료 함수가 호출될 수 있기 때문이다
(거의 이런 일은 발생하지 않지만, 네트워크 상황이 아주 극심한 경우 발생한다).
그에 의해 전역함수인 `async_write`는 완료 함수가 호출되면 100퍼센트 보냈음을
보증해준다.

5 | 채팅 프로그램 만들기

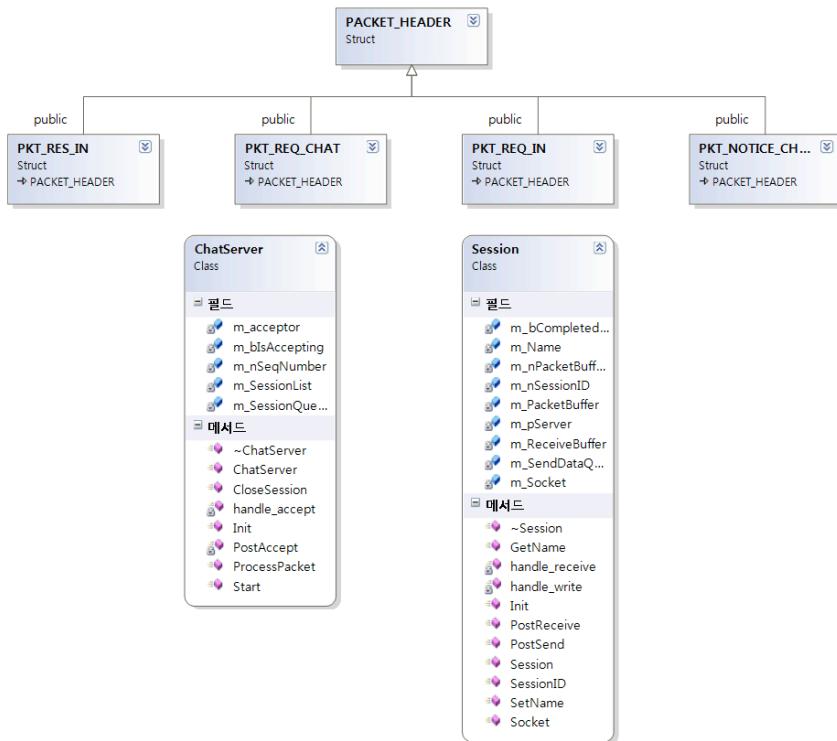
3장과 4장에서는 Boost.Asio의 사용법에 집중했던 만큼, Boost.Asio를 실제로 활용하는 방법에 대한 설명은 부족했을 것이다. 실제 네트워크 프로그램을 만들면, 서버는 수많은 클라이언트의 접속을 받고 클라이언트의 요청을 처리해야 한다. 그래서 이번 장에서는 4장에서 배운 바를 기반으로, Boost.Asio를 어떻게 실용적으로 사용하는지에 대해 예제를 중심으로 살펴볼 것이다.

예제는 네트워크 프로그램의 특징을 가장 쉽게 잘 보여주는 채팅 프로그램이다. 3장과 4장의 예제 코드에 비해 코드가 몇 배나 더 길고 중복되는 부분도 많아서, 소스 코드 전체를 보여줄 수는 없다. 여기서는 특징적인 부분을 중심으로 살펴볼 것이다(단, 전체 소스 코드는 꼭 확인해보기 바란다).

5.1 채팅 서버

예제 코드에서 채팅 서버의 프로젝트 이름은 ‘ChattingTCPServer’이다. 그림 5-1은 구현할 채팅 서버의 클래스 구성이다.

그림 5-1 채팅 서버의 클래스 다이어그램



5.1.1 대량의 클라이언트 접속 관리

서버는 클라이언트가 접속할 때마다 Session 클래스의 인스턴스를 할당한다. ChattingServer.h 파일의 ChatServer 클래스는 “`std::vector< Session* > m_SessionList;`”를 멤버로 가지고 있으며, ChatServer 클래스의 `Init()` 함수를 통해서 초기화한다.

```

void Init( const int nMaxSessionCount )
{

```

```
for( int i = 0; i < nMaxSessionCount; ++i )
{
    Session* pSession = new Session( i, m_acceptor.get_io_service(), this );
    m_SessionList.push_back( pSession );
    m_SessionQueue.push_back( i );
}
}
```

Session 클래스를 최대 접속 수만큼 미리 할당해서 m_SessionList에 담아 놓는다. 그리고 새로운 접속이 있을 때마다 사용하지 않는 Session을 할당하기 위해 아직 사용하지 않는 Session의 인덱스 번호를 m_SessionQueue에 저장해 놓는다.

접속 받기 요청을 할 때마다, PostAccept 함수⁰¹를 이용하여 m_SessionQueue에서 사용하지 않는 세션 번호를 가져와 async_accept에 사용한다.

```
bool PostAccept()
{
    if( m_SessionQueue.empty() )
    {
        m_bIsAccepting = false;
        return false;
    }

    m_bIsAccepting = true;
    int nSessionID = m_SessionQueue.front();
    m_SessionQueue.pop_front();

    m_acceptor.async_accept( m_SessionList[nSessionID]->Socket(),
                           boost::bind(&ChatServer::handle_accept,
```

01 · PostAccept()는 하나의 스레드에서만 호출하므로 공유 객체에 lock이 필요 없다

```
        this,
        m_SessionList[nSessionID],
        boost::asio::placeholders::error)
);

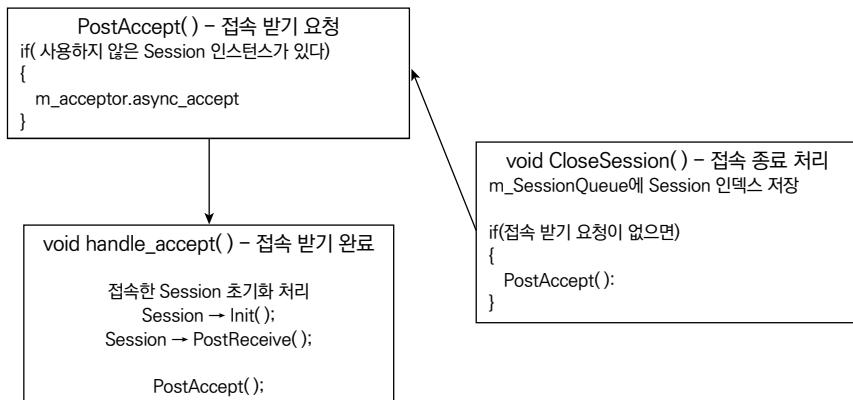
return true;
}
```

클라이언트의 접속이 끊기면 CloseSession() 함수를 사용하여 해당 Session 인덱스 번호를 다시 m_SessionQueue에 저장하고, 현재 접속 받기 요청을 하고 있지 않다면 접속 받기 요청을 한다.

```
void CloseSession( const int nSessionID )
{
    std::cout << "클라이언트 접속 종료. 세션 ID: " << nSessionID << std::endl;
    m_SessionList[ nSessionID ]->Socket().close();
    m_SessionQueue.push_back( nSessionID );

    if( m_bIsAccepting == false )
    {
        PostAccept();
    }
}
```

그림 5-2 비동기로 Accept 처리 흐름



5.1.2 클라이언트의 요청 처리

클라이언트에서 보낸 데이터를 받기 위해 Session 클래스의 `PostReceive()` 멤버 함수를 호출한다. `PostReceive()`는 Asio의 `async_read_som` 함수를 사용하여 데이터 받기 완료 시 호출될 Session 클래스의 `handle_receive` 멤버 함수를 등록 한다. Asio가 `handle_receive` 함수를 호출하면, 클라이언트가 보낸 데이터를 받아서 어떤 데이터를 보냈는지 분석하여 우리가 정의한 프로토콜별로 클라이언트의 요청을 처리한다.

`handle_receive`에서는 먼저 에러가 발생했는지 조사한 다음, 에러가 발생한 경우 ChatServer 클래스의 `CloseSession` 함수를 호출하여 ‘접속 종료 처리’를 한다. 에러가 없는 경우라면 받은 데이터에서 서버와 클라이언트 간에 정한 프로토콜에 맞춰 데이터를 처리한다.

네트워크의 특성상 클라이언트에서 서버로 데이터 보내기 요청을 동시에 여러 번 하면, 서버에서는 클라이언트에서 보내는 단위대로 받지 않는다. 한꺼번에 모두 받을 수도 있고 여러 번 나눠서 받을 수 있다(즉, 클라이언트는 `write`를 두 번 했는데,

서버에서는 receive가 한 번만 발생한다). 이런 경우를 처리하기 위해서 먼저 받은 데이터를 m_PacketBuffer에 저장한 후 클라이언트에서 동시에 여러 번 요청하면 서버는 한 번에 다 받으므로, 각 요청별로 나누어서 처리한다. 그리고 클라이언트가 보낸 데이터 중 일부만 도착한 경우, 우선은 처리하지 않고 남겨놓았다가 다음에 받은 데이터와 연결하여 처리한다.

```
void Session::handle_receive( const boost::system::error_code& error,
    size_t bytes_transferred )
{
    if( error )
    {
        ...
        m_pServer->CloseSession( m_nSessionID ); //에러가 있으면 연결 종료
    }
    else
    {
        // 받은 데이터를 패킷 버퍼에 저장
        memcpy( &m_PacketBuffer[ m_nPacketBufferMark ], m_ReceiveBuffer.data(),
            bytes_transferred );

        int nPacketData = m_nPacketBufferMark + bytes_transferred;
        int nReadData = 0;

        while( nPacketData > 0 ) // 받은 데이터를 모두 처리할 때까지 반복
        {
            if( nPacketData < sizeof(PACKET_HEADER) )
                // 남은 데이터가 패킷 헤더보다 작으면 중단
            {
                break;
            }
        }
    }
}
```

```

PACKET_HEADER* pHeader = (PACKET_HEADER*)&m_PacketBuffer[nReadData];

if( pHeader->nSize >= nPacketData )
    //처리할 수 있는 만큼 데이터가 있다면 패킷을 처리
{
    m_pServer->ProcessPacket( m_nSessionID, &m_PacketBuffer[nReadData] );
    nPacketData -= pHeader->nSize;
    nReadData += pHeader->nSize;
}
else
{
    break; // 패킷으로 처리할 수 있는 만큼이 아니면 중단
}
}

if( nPacketData > 0 ) // 남은 데이터는 m_PacketBuffer에 저장
{
    ...
}

// 남은 데이터 양을 저장하고 데이터 받기 요청
m_nPacketBufferMark = nPacketData;
PostReceive();
}
}

```

5.1.3 클라이언트에 데이터 보내기

4장의 예제에서는 비동기로 데이터를 보낸 후 완료 함수에서는 아무것도 하지 않았다. 설명을 간략하게 하기 위해서 그랬던 것이고, 또한 예제 프로그램 특성상 그 정도만 해도 충분했다. 그러나 사실 비동기로 데이터 보내기를 할 때는 꼭 지켜야

하는 것이 있다. 비동기 완료가 떨어질 때까지 보내기로 한 데이터를 잘 보관해야 한다는 점이다.

비동기는 요청이 끝날 때까지 대기하지 않는다. 즉, 데이터가 모두 다 보내졌다는 것은 비동기 완료 함수가 호출되었음을 의미한다. 그러므로 `async_write` 호출 후 `handle_write` 함수가 호출될 때까지 보낸 데이터를 보관하지 않으면, 데이터를 일부만 보낼 수 있다(4장의 API 설명에서도 언급했던 내용이다).

```
boost::asio::async_write( m_Socket,
    boost::asio::buffer( pSendData, nSize ),
    boost::bind( &Session::handle_write, this,
        boost::asio::placeholders::error,
        boost::asio::placeholders::bytes_transferred )
);
```

위와 같이 데이터 보내기를 한 후에 데이터 보내기가 완료되어 `Session` 클래스의 `handle_write`가 호출될 때까지는 보낸 데이터인 `pSendData`를 잘 보존하고 있어야 한다. `handle_write`가 호출될 때까지는 데이터가 다 보내진 게 아니기 때문이다.

그래서 채팅 서버 예제에 있는 `Session` 클래스의 `PostSend()`와 `handle_write`를 보면, `PostSend()`에서는 보낼 데이터를 저장하고 이전에 보내기 요청을 한 것이 완료되지 않았을 경우 곧바로 보내지 않고 보관한다. `handle_write`에서는 보낸 데이터를 제거하고 `PostSend()`에서 `async_write`로 보내지 못한 것이 있으면 마저 보낸다. 비동기 보내기를 할 때 실수하기 쉬운 것 중의 하나가 보내기 요청을 한 후 데이터를 바로 삭제하여 ‘보내기 실패’가 발생하는 경우다. 반드시 완전히 다 보내기 전까지는 데이터를 보관하고, 다 보낸 다음에 삭제해야 한다. 정말 중요하다!

```
void Session::PostSend( const bool bImmediately, const int nSize, char* pData )
{
    char* pSendData = nullptr;
    if( bImmediately == false )
    {
        pSendData = new char[nSize];
        memcpy( pSendData, pData, nSize );
        m_SendDataQueue.push_back( pSendData );
    }
    else
    {
        pSendData = pData;
    }

    if (bImmediately == false && m_SendDataQueue.size() > 1)
    {
        return;
    }

    boost::asio::async_write( m_Socket, boost::asio::buffer( pSendData, nSize ),
        boost::bind( &Session::handle_write, this,
        boost::asio::placeholders::error,
        boost::asio::placeholders::bytes_transferred )
    );
}

void Session::handle_write(const boost::system::error_code& error, size_t
bytes_transferred)
{
    delete[] m_SendDataQueue.front();
    m_SendDataQueue.pop_front();
    if( m_SendDataQueue.empty() == false )
```

```

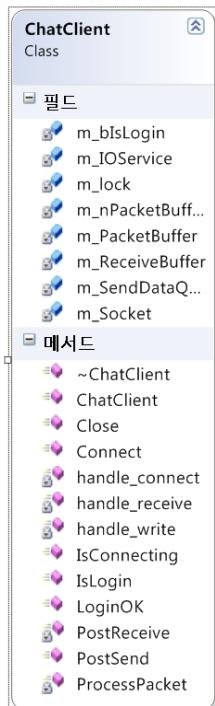
    {
        char* pData = m_SendDataQueue.front();
        PACKET_HEADER* pHeader = (PACKET_HEADER*)pData;
        PostSend( true, pHeader->nSize, pData );
    }
}

```

5.2 채팅 클라이언트

예제에서 채팅 클라이언트의 프로젝트 이름은 ‘ChattingTCPClient’다. 그럼 5-2는 구현할 채팅 클라이언트의 클래스 구성이다.

그림 5-3 채팅 클라이언트의 클래스 다이어그램



5.2.1 클라이언트의 워커 스레드 사용

채팅 클라이언트는 서버에 접속한 후 콘솔 창에서 텍스트를 입력받아서 서버에 보낸다. 서버와 클라이언트 모두 Boost.Asio 함수를 사용한다는 점에서는 특별히 다를 것이 없다. 그러나 클라이언트의 경우, 서버와 달리 메인 스레드가 아닌 워커 스레드_{Worker Thread}에서 io_service.run()을 실행한다.

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

    auto endpoint = boost::asio::ip::tcp::endpoint(
        boost::asio::ip::address::from_string("127.0.0.1"),
        PORT_NUMBER);

    ChatClient Client( io_service );
    Client.Connect( endpoint );

    boost::thread thread( boost::bind(&boost::asio::io_service::run,
        &io_service) );
    .....
}
```

워커 스레드는 Boost 라이브러리의 ‘thread 라이브러리’를 사용했다. 지금까지 본 것 중에서 가장 특이한 점은 바로, 클라이언트의 main()에서 메인 스레드가 아닌 워커 스레드를 만들어 io_service.run()을 호출한 것이다.

아마 그 이유를 짐작하는 개발자도 있을 것이다. 3장에서 이야기했는데, io_service.run()을 호출하면 비동기 요청이 끝날 때까지 ‘무한 대기 상태’에 들어가기 때문이다. 만약 메인 스레드에서 io_service.run()을 호출하면 채팅할 때 입력

을 할 수 없다.

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

    auto endpoint = boost::asio::ip::tcp::endpoint(
        boost::asio::ip::address::from_string("127.0.0.1"),
        PORT_NUMBER);

    ChatClient Client( io_service );
    Client.Connect( endpoint ); // 서버에 접속 요청

    // 워커 스레드를 만든 후 io_service.run 호출
    boost::thread thread( boost::bind(&boost::asio::io_service::run,
        &io_service) );

    char szInputMessage[MAX_MESSAGE_LEN * 2] = {0,};

    while( std::getline( szInputMessage, MAX_MESSAGE_LEN ) )
    {
        ...

        if( Client.IsLogin() == false )
        {
            ...
            // 채팅을 종료할 때까지 입력 데이터를 서버에 보낸다.
            Client.PostSend( false, SendPkt.nSize, (char*)&SendPkt );
        }
        else
        {
            ...
        }
    }
}
```

```

        Client.PostSend( false, SendPkt.nSize, (char*)&SendPkt );
    }
}

io_service.stop();

Client.Close(); // io_service를 중단하고(접속을 끊고),
thread.join(); // 스레드가 종료할 때까지 기다린다.

std::cout << "클라이언트를 종료해 주세요" << std::endl;
return 0;
}

```

워커 스레드를 만들어서 `io_service.run()`을 호출하면 프로그램이 중단되는 것을 막을 수 있지만, 스레드가 두 개 이상이 되면 새로운 문제가 생긴다. 바로 동기화다.

5.2.2 데이터 보내기 동기화

앞서 채팅 서버에서 데이터 보내기를 설명할 때, 데이터를 보낸 후 보내기가 완료될 때까지는 데이터를 삭제하면 안 된다고 했다. 서버는 하나의 스레드이기 때문에 큐를 사용하여 보낼 데이터를 저장한 다음 삭제하면 된다. 그러나 클라이언트는 워커 스레드를 사용해서 데이터를 보내므로, 메인 스레드^{Main Thread}와 보내기 완료는 워커 스레드에서 호출된다.

즉, `ChatClient` 클래스의 `PostSend()`는 메인 스레드에서 사용하고 `handle_write()`는 워커 스레드에서 사용한다. 따라서 이 두 함수가 같이 사용하는 공유 리소스는 동기화를 잘해야 한다.

```
void PostSend( const int nSize, char* pData )
{
    char* pSendData = nullptr;
    EnterCriticalSection(&m_lock); ←
    if(bImmediately == false )
    {
        char* pSendData = new char[nSize];
        memcpy( pSendData, pData, nSize );
        m_SendDataQueue.push_back( pSendData );
    }
    else
    {
        pSendData = pData;
    }

    if( m_SendDataQueue.size() < 2 )
    {
        boost::asio::async_write( m_Socket,
            boost::asio::buffer( pSendData, nSize ),
            boost::bind( &ChatClient::handle_write,
                this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred )
        );
    }
    LeaveCriticalSection(&m_lock); ←
}

void handle_write(const boost::system::error_code& error,
    size_t bytes_transferred)
{
    EnterCriticalSection(&m_lock); ←
}
```

공유 리소스인
m_SendDataQueue의
충돌을 보호하기 위해
크리티컬 сек션으로
동기화

```

delete[] m_SendDataQueue.front();
m_SendDataQueue.pop_front();

char* pData = nullptr;

if( m_SendDataQueue.empty() == false )
{
    pData = m_SendDataQueue.front();
}

LeaveCriticalSection(&m_lock); ←
}

if( pData != nullptr )
{
    PACKET_HEADER* pHeader = (PACKET_HEADER*)pData;
    PostSend( pHeader->nSize, pData );
}
}

```

채팅 서버와 클라이언트의 소스 코드는 3장의 예제에 비해서 많지만, 위에서 설명한 것 이외에는 거의 같다. 채팅 서버 고유의 기능을 구현하기 위한 소스 코드가 더 해졌을 뿐이다(그리고 중복된 소스 코드도 있어서 지금보다 더 줄일 수 있다). 코드 분석 역시 앞서 했던 것처럼 Boost.Asio 함수를 중심으로 따라가면 쉽게 이해할 수 있을 것이다. 로씨 Boost.Asio를 이용하여, 비동기 IO를 사용하는 채팅 서버를 어떻게 만드는지 알아보았다. 이것을 토대로 콘텐츠 부분만 바꾸면 얼마든지 게임이나 비즈니스 프로그램으로 바꿀 수 있다. 네트워크 프로그램이 가져야 하는 기본 기능은 다 갖추었기 때문이다.

그림 5-4 채팅 서버와 클라이언트 실행 결과

The image displays three separate windows, each representing a different component of a chat application. The top window is titled 'C:\exam_boost\ChattingTCPServer\Debug\ChattingTCPServer.exe' and contains the following text:
서버 시작.....
클라이언트 접속 성공. SessionID: 0
클라이언트 접속 성공. SessionID: 1
클라이언트 로그인 성공 Name: hanbit01
클라이언트 로그인 성공 Name: hanbit02

The middle window is titled 'C:\exam_boost\ChattingTCPClient\Debug\ChattingTCPClient.exe' and contains the following text:
서버 접속 성공
이름을 입력하세요!!
hanbit01
클라이언트 로그인 성공 ?: 1
우하하
hanbit01: 우하하
hanbit02: Boost 최고

The bottom window is also titled 'C:\exam_boost\ChattingTCPClient\Debug\ChattingTCPClient.exe' and contains the following text:
서버 접속 성공
이름을 입력하세요!!
hanbit02
클라이언트 로그인 성공 ?: 1
hanbit01: 우하하
Boost 최고
hanbit02: Boost 최고

5.3 개선할 점

5장의 예제는 나름 쓸만한 프로그램으로 만들어졌지만, 프로그램의 성능이나 소스 코드의 유지 보수 측면에서는 아직 손볼 점이 있다.

우선 성능 부분을 살펴보자. 클라이언트에서 `io_service.run()`을 워커 스레드로

사용하여 네트워크 I/O에서 성능을 높였듯, 서버에서도 하나 이상의 워커 스레드를 만들어서 `io_service.run()`을 사용하면 네트워크 I/O 부분에서 성능을 높일 수 있다(클라이언트보다는 서버에서 성능을 더 높일 수 있다). 다만 이런 경우 공유 리소스의 동기화 문제가 발생하니 조심해야 한다. 그러니 성능에 별다른 문제가 없다면, 개발 편의성과 안전성을 위해서 `io_service.run()`은 하나의 워커 스레드에 서만 사용하는 것도 좋다. 한편, 데이터를 받거나 보낼 때 복사나 동적 할당을 자주 하는데, 이 역시 잘 다듬으면 성능을 높일 수 있는 방법이 된다.

소스 코드 측면에서 보자면, Boost.Asio 함수를 사용하는 부분은 클라이언트와 서버 모두 같은 코드를 사용할 수 있는 부분이 많다. 클라이언트와 서버가 함께 사용하는 소스 코드 부분은 따로 클래스로 묶어서 사용하는 것이 좋다.

이것으로 Boost.Asio를 사용하여 TCP 환경에서 네트워크 통신하는 방법의 핵심을 모두 설명했다. 이제 TCP와 쌍벽을 이루는 UDP에 대해서 살펴보자.

6 | 비동기 I/O를 사용한 UDP Echo 서버, 클라이언트 만들기

4장과 5장의 내용을 잘 이해했다면, Boost.Asio의 UDP 네트워크 프로그래밍 역시 쉽게 이해할 수 있다. 이미 대부분의 내용을 배웠기 때문이다. 앞에서 배운 것과 다른 부분은 UDP 프로토콜의 특성상 accept와 connect가 필요 없고, UDP용 함수를 사용하여 write와 read를 구현한다는 점이다. UDP Echo 서버와 클라이언트는 3장의 예제와 기능적으로 같다. 그러므로 차이가 있는 UDP용 write(보내기)와 read(받기) 함수만을 살펴보기로 한다(예제의 전체 소스 코드는 꼭 찾아보기 바란다).

6.1 UDP로 데이터 보내고 받기

UDP를 사용하려면 Boost.Asio에서 UDP용 boost::asio::ip::udp::socket 클래스와 boost::asio::ip::udp::endpoint를 사용해야 한다.

6.1.1 데이터 보내기

```
m_Socket.async_send_to(
    //인자 1. 클라이언트가 보낸 데이터를 저장할 버퍼
    boost::asio::buffer(m_WriteMessage),

    //인자 2. 데이터를 보낼 주소
    boost::asio::ip::udp::endpoint( boost::asio::ip::address::
        from_string(UDP_IP), CLIENT_PORT_NUMBER),

    //인자 3. 완료하면 호출할 함수
    boost::bind( &UDP_Server::handle_write,
        this,
```

```
        boost::asio::placeholders::error,
        boost::asio::placeholders::bytes_transferred )
);
```

데이터는 boost::asio::ip::udp::socket 클래스의 `async_send_to` 함수를 사용하여 보낸다. TCP용 `async_write_some`과 크게 차이 나는 것은 endpoint 부분이다. UDP는 특성상 연결되지 않은 상태에서 데이터를 주고받기 때문에, 데이터를 보낼 때마다 보낼 곳의 주소를 지정해야 한다.

위 코드에서 주의해야 할 점이 하나 있다. 바로 `boost::asio::ip::address::from_string` 사용 부분이다. `from_string`은 에러 코드를 사용하는 버전도 있는데, 위의 코드에서처럼 에러 코드를 사용하지 않으면 입력 값이 IP 주소 문자열이 아닐 경우 예외가 발생한다. 따라서 서비스 정책이나 그 밖에 다른 이유로 IP 주소가 틀릴 수도 있다면 ‘에러 코드 사용 버전’을 사용하기 바란다. 다음은 에러 코드 사용 버전이다.

```
| asio::error_code error;
| asio::ip::address::from_string(UDP_IP, error);
```

6.1.2 데이터 받기

```
m_Socket.async_receive_from(
    boost::asio::buffer(m_ReceiveBuffer, 128), // 인자 1. 받은 데이터를 저장할 버퍼
    m_SenderEndpoint,                         // 인자 2. 데이터를 보낸 곳의 주소
    boost::bind( &UDP_Server::handle_receive, // 인자 3. 완료하면 호출할 함수
                this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred )
);
```

boost::asio::ip::udp::socket 클래스에 있는 `async_receive_from` 함수를 사용하여 데이터를 보낸다. TCP용 `async_read_some`과 크게 차이가 나는 부분은 두 번째 인자인 ‘`m_SenderEndpoint(boost::asio::ip::udp::endpoint 클래스)`’다. UDP의 특성상 연결하지 않은 상태에서 데이터를 주고받기 때문에, 어디서 데이터를 보냈는지 알 수 없다. 그래서 두 번째 인자에서 보낸 곳의 주소를 얻는다.

앞서 이야기했듯이 4장을 잘 이해했다면, 이번 장에서는 UDP용 함수를 어떻게 사용하는지 살펴보기만 하면 된다. 그러니 위에서 설명하지 않은 부분은 편한 마음으로 예제 `AsynchronousUDPServer`와 `AsynchronousUDPClient`를 쭉 훑어보기 바란다.

6.2 관련 Boost.Asio API

| `boost::asio::ip::udp::socket` 클래스의 `async_send_to`

UDP 프로토콜로 데이터를 보낸다. 비동기 방식이므로 데이터를 다 보낼 때까지 대기하지 않는다. 두 가지 버전이 있다.

```
① template<typename ConstBufferSequence, typename WriteHandler>
void async_send_to(const ConstBufferSequence & buffers, const endpoint_
type & destination, WriteHandler handler);

② template<typename ConstBufferSequence, typename WriteHandler>
void async_send_to(const ConstBufferSequence & buffers, const endpoint_
type & destination, socket_base::message_flags flags, WriteHandler
handler);
```

- `buffers` : 보낼 데이터를 저장한 베퍼
- `destination` : 데이터를 보낼 곳의 주소

- flags : 어떤 방식으로 보낼지 지정⁰¹

- handler : 완료 함수

| `boost::asio::ip::udp::socket` 클래스의 `async_receive_from`

UDP 프로토콜로 데이터를 받는다. 비동기 방식이므로 데이터를 다 받을 때까지 대기하지 않는다. 두 가지 버전이 있다.

① `template<typename MutableBufferSequence, typename ReadHandler>`

```
void async_receive_from(const MutableBufferSequence & buffers,  
                        endpoint_type & sender_endpoint, ReadHandler handler);
```

② `template<typename MutableBufferSequence, typename ReadHandler>`

```
void async_receive_from(const MutableBufferSequence & buffers,  
                        endpoint_type & sender_endpoint, socket_base::message_flags flags,  
                        ReadHandler handler);
```

- buffers : 보낼 데이터를 저장한 버퍼
- destination : 데이터를 보낼 곳의 주소
- flags : 어떤 방식으로 보낼지 지정⁰²
- handler : 완료 함수

이것으로 Boost.Asio를 사용한 네트워크 프로그래밍에 대한 설명을 마무리하려고 한다. 지금까지 설명한 것만 올바르게 이해했다면 Boost.Asio의 핵심적인 부분에 대해서는 다 배웠다고 할 수 있다. 다음 장부터는 일반적인 프로그래밍에서 Boost.Asio의 비동기 기능을 사용하는 방법에 대해 살펴볼 것이다.

01 linux의 Socket 프로그래밍과 관련된 기능으로, “man7.org/linux/man-pages/man2/send.2.html”의 설명을 참고하기 바란다.

02 linux의 Socket 프로그래밍과 관련된 기능으로, “man7.org/linux/man-pages/man2/recv.2.html”의 설명을 참고하기 바란다.

7 | Boost.Asio의 Timer 사용하기

보통 네트워크 프로그래밍에만 Boost.Asio를 사용하는 것으로 아는 사람이 많은데, Boost.Asio는 사실 생각보다 활용도가 높다. 비동기 기능을 사용하면 네트워크 프로그래밍뿐만 아니라 일반적인 프로그래밍에도 활용할 수 있다.

이번 장에서는 Boost.Asio의 기능 중 타이머 기능에 대해 살펴볼 것이다.

7.1 기본적인 타이머

알람 시계처럼 지정한 시간이 되면 특정 함수를 호출하는 기능이 필요할 때가 있다. Windows에서는 타이머 기능을 위해 Win32 API의 WM_TIMER라는 메시지를 사용하며, 좀 더 정확한 시간이 필요할 때는 CreateTimerQueueTimer를 사용한다.

Boost.Asio에서는 이와 비슷한 기능을 `boost::asio::steady_timer` 클래스로 지원한다. 타이머 방식에는 `deadline_timer`, `high_resolution_timer`, `system_timer`가 있다.

여기서 잠깐_ `deadline_timer`, `high_resolution_timer`, `system_timer`

`deadline_timer`, `high_resolution_timer`, `system_timer` 등은 C++11의 chrono에 있는 타임 클래스와 비슷하다. 다음은 타이머별 특징이다.

- `system_clock` : 가장 일반적으로 사용하며, 시스템 시간을 표현하기 위한 시간이다.
- `steady_clock` : 물리적인 시간처럼 역행하지 않는 시간을 나타내기 위한 시간이다.
- `high_resolution_clock` : 해당 플랫폼(Windows 또는 Linux 등)에서 가장 정밀한 단위의 시간이다.

예제 7-1은 Boost.Asio를 사용하여 2초가 지나면 onTimer 함수를 호출하는 프로그램이다.

예제 7-1 Timer1

```
#include <SDKDDKVer.h>
#include <iostream>
#include <boost/asio.hpp>
#include <boost/asio/steady_timer.hpp>

void OnTimer( const boost::system::error_code& error )
{
    if( !error )
    {
        std::cout << "Call OnTimer!!! " << time(NULL) << std::endl;
    }
    else
    {
        std::cout << "error No: " << error.value() << " error Message: "
              << error.message() << std::endl;
    }
}

int main()
{
    std::cout << "시작: " << time(NULL) << std::endl;
    boost::asio::io_service io_service;

    boost::asio::steady_timer timer(io_service);
    timer.expires_from_now( boost::chrono::milliseconds(2000) );
    timer.async_wait( OnTimer );
}
```

```
    io_service.run();

    std::cout << "종료: " << time(NULL) << std::endl;
    return 0;-
}
```

그림 7-1 실행 결과 화면



7.1.1 타이머 사용하기

타이머를 사용하기 위해서 설정해야 할 것은 다음과 같다.

1. 타이머를 사용하려면 사용하는 타이머 클래스의 헤더를 포함해야 한다. steady_timer는 다음의 헤더 파일을 포함해야 한다.
`#include <boost/asio/steady_timer.hpp>`
 2. io_service와 timer 클래스를 생성한다.
 3. 타이머가 동작할 시간과 호출할 함수를 지정한다.
 4. io_service.run()을 호출한다.
-

```
boost::asio::io_service io_service;
boost::asio::steady_timer timer(io_service);
```

```
// 타이머가 동작할 시간을 설정한다.  
// boost::chrono를 사용하여 다양한 시간 단위로 설정할 수 있다.  
// 예제에서는 1/1000초 단위를 사용하여 2초 후에 동작하도록 설정했다.  
timer.expires_from_now( boost::chrono::milliseconds(2000) );  
timer.async_wait( OnTimer );  
io_service.run();
```

7.2 반복하는 타이머

예제 7-1의 타이머는 한 번 실행한 후 종료된다. 하지만 타이머를 사용할 경우 대부분 반복적으로 타이머 기능을 사용한다. 이번에는 타이머를 반복적으로 사용하는 방법에 대해서 살펴보자. 예제를 보면 알겠지만, Boost.Asio 네트워크 프로그래밍의 비동기 데이터 읽기에서 사용한 방법과 비슷하다. 예제 7-2는 타이머를 다섯 번까지 반복적으로 사용하는 코드다.

예제 7-2 Timer2

```
#include <SDKDDKVer.h>  
#include <iostream>  
#include <boost/bind.hpp>  
#include <boost/asio.hpp>  
#include <boost/asio/steady_timer.hpp>  
  
void OnTimer( const boost::system::error_code& error,  
              boost::asio::steady_timer* pTimer );  
int nCount = 0;  
  
void SetTimer( boost::asio::steady_timer* pTimer )  
{  
    std::cout << "SetTimer: " << time(NULL) << std::endl;  
    pTimer->expires_from_now( boost::chrono::milliseconds(1000));
```

```

pTimer->async_wait( boost::bind( OnTimer,
                                boost::asio::placeholders::error,
                                pTimer )
                     );
}

void OnTimer( const boost::system::error_code& error,
              boost::asio::steady_timer* pTimer )
{
    if( !error )
    {
        ++nCount;
        std::cout << "OnTimer: " << nCount << std::endl;

        if( nCount < 5 )
        {
            SetTimer( pTimer );
        }
    }
    else
    {
        std::cout << "error No: " << error.value() << " error Message: "
              << error.message() << std::endl;
    }
}

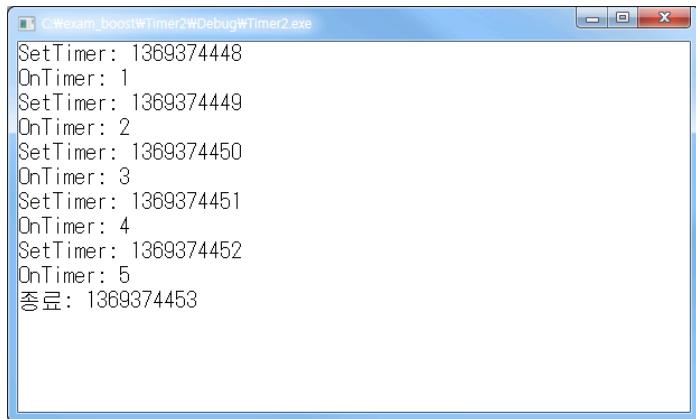
int main()
{
    boost::asio::io_service io_service;
    boost::asio::steady_timer timer(io_service);
    SetTimer( &timer );
}

```

```
    io_service.run();

    std::cout << "종료: " << time(NULL) << std::endl;
    return 0;
}
```

그림 7-2 실행 결과 화면



7.3 설정한 타이머 취소하기

설정한 타이머를 취소하고 싶을 수도 있다. 그럴 때는 타이머 클래스의 `cancel()` 함수를 사용하면 된다. 단, `cancel()` 함수를 사용할 경우 해당 인스턴스를 통해서 설정한 모든 타이머가 취소된다는 점을 주의하도록 한다. 예제 7-3은 타이머 두 개를 설정한 후 `cancel()` 함수를 호출하는 코드다.

예제 7-3 Timer3

```
#include <SDKDDKVer.h>
#include <iostream>
#include <boost/asio.hpp>
#include <boost/asio/steady_timer.hpp>

void OnTimer1( const boost::system::error_code& error )
{
    if( !error )
    {
        std::cout << "Call OnTimer1 !!! " << time(NULL) << std::endl;
    }
    else
    {
        std::cout << "OnTimer1. error No: " << error.value() << " error Message: "
              << error.message() << std::endl;
    }
}

void OnTimer2( const boost::system::error_code& error )
{
    if( !error )
    {
        std::cout << "Call OnTimer2 !!! " << time(NULL) << std::endl;
    }
    else
    {
        std::cout << "OnTimer2. error No: " << error.value() << " error Message: "
              << error.message() << std::endl;
    }
}
```

```

int main()
{
    std::cout << "시작: " << time(NULL) << std::endl;

    boost::asio::io_service io_service;
    boost::asio::steady_timer timer(io_service);

    timer.expires_from_now( boost::chrono::milliseconds(2000) );

    timer.async_wait( OnTimer1 );
    timer.async_wait( OnTimer2 );

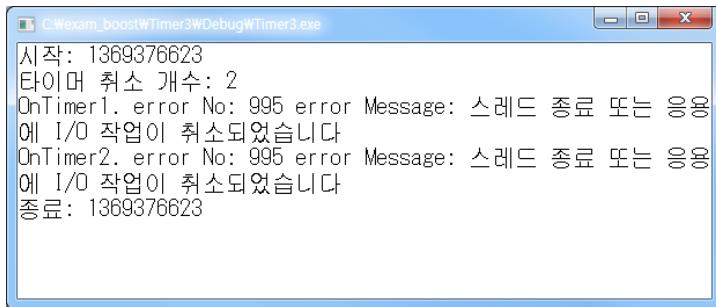
    const int count = timer.cancel();
    std::cout << "타이머 취소 개수: " << count << std::endl;

    io_service.run();

    std::cout << "종료: " << time(NULL) << std::endl;
    return 0;
}

```

그림 7-3 Timer3의 실행 결과 화면



cancel() 함수를 사용하여 기존에 설정한 타이머를 취소할 수 있지만, 설정한 모든 타이머가 취소된다는 점은 아쉽다. 다수의 타이머를 설정한 경우 이처럼 특정한 타이머를 취소할 수는 없지만, 첫 번째 타이머만 취소하는 방법은 있다. 이때 사용하는 것이 cancel_one() 함수다.

예제 7-4는 타이머 세 개를 설정한 후, cancel_one() 함수를 사용하여 첫 번째 타이머를 취소하는 코드다.

예제 7-4 Timer4

```
#include <SDKDDKVer.h>
#include <iostream>
#include <boost/asio.hpp>
#include <boost/asio/steady_timer.hpp>

void OnTimer1( const boost::system::error_code& error )
{
    if( !error )
    {
        std::cout << "Call OnTimer1 !!! " << time(NULL) << std::endl;
    }
    else
    {
        std::cout << "OnTimer1. error No: " << error.value() << " error Message: "
              << error.message() << std::endl;
    }
}

void OnTimer2( const boost::system::error_code& error )
{
    if( !error )
    {
```

```

        std::cout << "Call OnTimer2 !!! " << time(NULL) << std::endl;
    }
else
{
    std::cout << "OnTimer2. error No: " << error.value() << " error Message: "
        << error.message() << std::endl;
}
}

void OnTimer3( const boost::system::error_code& error )
{
    if( !error )
    {
        std::cout << "Call OnTimer3 !!! " << time(NULL) << std::endl;
    }
else
{
    std::cout << "OnTimer3. error No: " << error.value()
        << " error Message: " << error.message() << std::endl;
}
}

int main()
{
    std::cout << "시작: " << time(NULL) << std::endl;
    boost::asio::io_service io_service;
    boost::asio::steady_timer timer(io_service);
    timer.expires_from_now( boost::chrono::milliseconds(2000) );

    timer.async_wait( OnTimer1 );
    timer.async_wait( OnTimer2 );
    timer.async_wait( OnTimer3 );
}

```

```
const int count = timer.cancel_one();
std::cout << "타이머 취소 개수: " << count << std::endl;

io_service.run();

std::cout << "종료: " << time(NULL) << std::endl;
return 0;
}
```

그림 7-4 실행 결과 화면



8 | Boost.Asio를 사용한 백그라운드 메시지 처리

네트워크 통신을 하거나 사용자의 요청을 처리하는 데 시간이 많이 걸리는 프로그램의 경우, 요청을 한 후 처리가 완료될 때까지 사용자에게 보이는 화면이 멈춘 것처럼 보일 수 있다. UX 관점에서 보면 상당히 불편한 프로그램이다. 특히 모바일 프로그램에서는 PC보다 훨씬 더 불편한데, 이것 때문에 앱^{App}을 지우고 싶어질 정도다. 필자는 매일 모바일 스트리밍으로 음악을 듣는데, 필자가 사용하는 프로그램은 네트워크 환경이 조금만 안 좋아도 한동안 앱이 먹통이 되어서 무척 불편하다(대체 서비스만 있다면 언제라도 지울 준비가 되어 있다!).

이렇게 특정 작업이 끝날 때까지 프로그램이 멈춤 상태가 되는 이유는 대부분 동기 방식으로 프로그래밍을 했기 때문이다. 시간이 많이 걸리는 작업이나 네트워크를 사용하는 작업을 수행할 때는 대기 상태가 발생할 수 있으므로 최대한 비동기 방식을 사용해야 한다.

지금까지는 우리는 Boost.Asio를 사용하여 비동기 프로그래밍을 배웠는데 이것을 네트워크나 타이머가 아닌 범용적인 방식의 비동기 프로그래밍에도 사용할 수 있다. 현재로서는 Boost.Asio을 범용적인 비동기 프로그래밍에 사용하는 예를 찾아보기 힘들 텐데, 지금부터 설명하는 것을 잘 기억해두었다가 Boost.Asio를 좀 더 폭넓게 사용하기 바란다. 많이 유용할 것이다.

예제 8-1은 멀티 스레드를 사용하여 사용자의 요청을 백그라운드로 처리하는 코드다. 예제는 Function 함수와 TEST 클래스의 Function 멤버 함수를 io_service의 post 함수로 호출하여, 비동기로 실행한다. 이런 방식을 사용하면 시간이 오래 걸리는(몇 초 이상 걸리는) 요청을 해도 프로그램이 먹통이 되지 않아서, 요청을 중간에 중단하거나 다른 요청을 할 수 있다.

예제 8-1 WorkerThread

```
#include <SDKDDKVer.h>
#include <iostream>
#include <string>
#include <boost/asio.hpp>
#include <boost/thread.hpp>
#include <boost/bind.hpp>

class BackGroundJobManager
{
    boost::asio::io_service& m_io_service;
    boost::shared_ptr<boost::asio::io_service::work> m_Work;
    boost::thread_group m_Group;

public:
    BackGroundJobManager( boost::asio::io_service& io_service, std::size_t size )
        : m_io_service(io_service)
    {
        m_Work.reset( new boost::asio::io_service::work(m_io_service) );

        for (std::size_t i = 0; i < size; ++i)
        {
            m_Group.create_thread( boost::bind(&boost::asio::io_service::run,
                &m_io_service) );
        }
    }

    ~BackGroundJobManager()
    {
        m_Work.reset();
        m_Group.join_all();
    }
}
```

```

template <class F>
void post(F f)
{
    m_io_service.post(f);
}
};

boost::mutex g_mutex;

void Function( int nNumber )
{
    char szMessage[128] = {0,};
    sprintf_s( szMessage, 128-1, "%s(%d) | time(%d)", __FUNCTION__, nNumber,
               time(NULL) );
    {
        boost::mutex::scoped_lock lock(g_mutex);
        std::cout << "워커 스레드 ID: " << ::GetCurrentThreadId()
               << ". " << szMessage << std::endl;
    }

    ::Sleep(1000);
}

class TEST
{
public:
    TEST() { }

    void Function( int nNumber )
    {
        ::Function( nNumber );
    }
}

```

```

    }

};

int main()
{
    std::cout << "메인 스레드 ID: " << ::GetCurrentThreadId() << std::endl;

    boost::asio::io_service io_service;
    BackGroundJobManager JobManager( io_service, 3 );

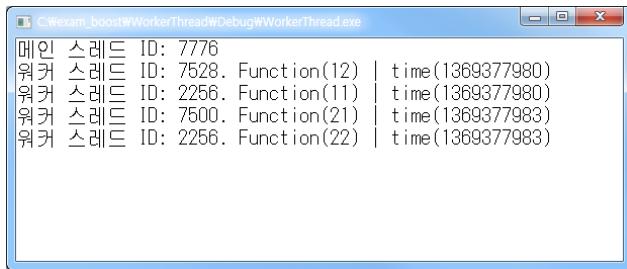
    JobManager.post( boost::bind(Function, 11) );
    JobManager.post( boost::bind(Function, 12) );
    ::Sleep(3000);

    TEST test;
    JobManager.post(boost::bind(&TEST::Function, &test, 21));
    JobManager.post(boost::bind(&TEST::Function, &test, 22));
    ::Sleep(3000);

    return 0;
}

```

그림 8-1 실행 결과 화면



예제 8-1은 BackGroundJobManager 클래스에 작업을 요청하면, 미리 만들어 놓은 스레드 풀(thread pool)을 사용하여 워커 스레드로 작업을 처리한다.

```
class BackGroundJobManager
{
    boost::asio::io_service& m_io_service;
    boost::shared_ptr<boost::asio::io_service::work> m_Work;
    boost::thread_group m_Group;

public:
    BackGroundJobManager( boost::asio::io_service& io_service, std::size_t size )
        : m_io_service(io_service)
    {
        // boost::asio::io_service::work를 사용하면, io_service에 비동기
        // 요청을 하기 전에 run 함수를 실행해도 run 함수가 종료되지 않는다.
        m_Work.reset( new boost::asio::io_service::work(m_io_service) );

        for (std::size_t i = 0; i < size; ++i)
        {
            // boost::thread_group를 사용하여 지정한 개수만큼 스레드를 생성한 후
            // io_service() 실행한다.
            m_Group.create_thread( boost::bind(&boost::asio::io_service::run,
                &m_io_service) );
        }
    }

    ~BackGroundJobManager()
    {
        m_Work.reset();          // io_service.run을 종료
        m_Group.join_all();     // 생성한 스레드가 종료될 때까지 대기한다.
    }
}
```

```

    }

template <class F>
void post(F f)
{
    m_io_service.post(f); // post 함수를 사용하여 비동기로 실행할 함수를
                          // io_service에 요청
}

};

boost::mutex g_mutex; // 콘솔 출력 시 동기화를 위해 사용하는 동기화 객체

void Function( int nNumber )
{
    // 함수 이름과 인자 값, 시간, 이 함수를 실행하는
    // 스레드의 ID를 콘솔에 출력한다.
    char szMessage[128] = {0,};
    sprintf_s( szMessage, 128-1, "%s(%d) | time(%d)", __FUNCTION__,
               nNumber, time(NULL) );
    {
        boost::mutex::scoped_lock lock(g_mutex);
        std::cout << "워커 스레드 ID: " << ::GetCurrentThreadId() << ". "
                  << szMessage << std::endl;
    }

    ::Sleep(1000);
}

```

| boost::asio::io_service 클래스의 post

io_service에 지정한 핸들러를 호출하도록 요구한다. post는 대기하지 않고 바로 반환한다. 사용하는 방법은 다음과 같다.

```
template<typename CompletionHandler>
void post(CompletionHandler handler);
```

- handler: 비동기로 io_service에서 호출해줄 함수

post로 handler를 등록한 후 io_service에서 run이나 run_one, 또는 poll이나 poll_one을 호출해야 한다.

여기서 잠깐_ post와 dispatch의 차이

io_service에는 post와 비슷한 동작을 하는 dispatch 함수가 있다. 이 둘은 다음과 같은 차이가 있다. 멀티 스레드 환경에서 스레드 A와 스레드 B가 있다고 가정해보자. 스레드 A에서 dispatch를 사용하여 핸들러를 등록하면, 등록한 핸들러는 스레드 A에서만 호출된다. 그러나 post를 사용하여 핸들러를 등록하면, 스레드 A에서 등록한 핸들러가 스레드 B에서도 호출될 수 있다. 즉, dispatch는 핸들러를 등록한 스레드에서만 이벤트가 처리되고, post는 등록한 스레드와 상관없이 이벤트가 처리된다.

이것으로 Boost.Asio의 비동기 기능을 사용해서 네트워크 프로그래밍, 타이머, 범용적인 비동기 프로그래밍하는 방법에 대해 알아보았다. 아직 소개하지 못한 소소한 기능에 대해서는 다음 장에서 하나씩 소개할 테니, 조금 더 힘내서 끝까지 완주하기 바란다.

9 | Boost.Asio의 기타 기능들

이번 장에서는 Boost.Asio 사용에 관한 팁을 소개한다. 관련 내용을 숙지하면 프로그래밍하는 데 많은 도움이 될 것이다.

표 9-1은 Boost.Asio와 Windows의 IOCP API⁰¹를 비교한 것이다. Windows에서 비동기 네트워크 프로그래밍을 할 때 사용하는 IOCP API와 비슷한 기능을 하는 Boost.Asio API를 비교해 놓았다. IOCP로 비동기 네트워크 프로그래밍을 해본 경험이 있다면 Boost.Asio를 사용할 때 각 API가 어떤 기능을 하는지 이해하기 쉬울 것이다.

표 9-1 Boost.Asio와 Windows의 IOCP 비교

Boost.Asio	IOCP
asio::io_service io_service;	CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
asio::tcp::socket socket(io_service);	CreateIoCompletionPort((HANDLE)client, g_hlocp, (DWORD)con, 0);
io_service.run();	_beginthreadex(NULL, 0, CompletionThread, (LPVOID)hCompletionPort, 0, NULL); ... unsigned int __stdcall CompletionThread(LPVOID pComPort) { while(1) { GetQueuedCompletionStatus(g_hlocp, &readbytes, &dwCompKey, (LPOVERLAPPED *)&pOverlap, INFINITE); } }

01 Windows 플랫폼 전용의 비동기 I/O 관련 Win32 API이다.

9.1 Boost.Asio와 스레드

4장과 5장에서 살펴본 비동기 네트워크 프로그램은 대부분 Boost.Asio를 하나의 스레드에서 사용했다. 이 말은 곧 하나의 스레드에서 `io_service.run()`을 호출했다는 의미다. Windows에서 네트워크 프로그래밍을 할 때, 보통 IOCP를 멀티 스레드로 사용한다. 이유는 바로 성능 때문이다. Boost.Asio에서도 멀티 스레드로 사용할 수 있는데 방법은 간단하다. `io_service.run()`을 멀티 스레드로 호출하기만 하면 된다.

`io_service.run()`을 스레드 하나(비동기 1-스레드 모델)와 멀티 스레드(비동기 n-스레드 모델)로 사용하는 방법은 다음 그림과 같다.

그림 9-1 비동기 1-스레드 모델

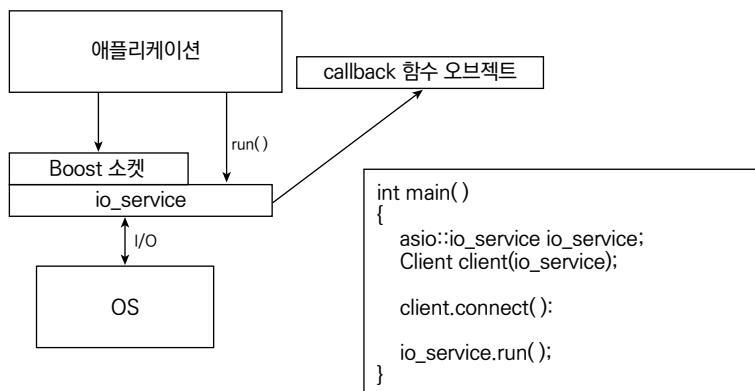
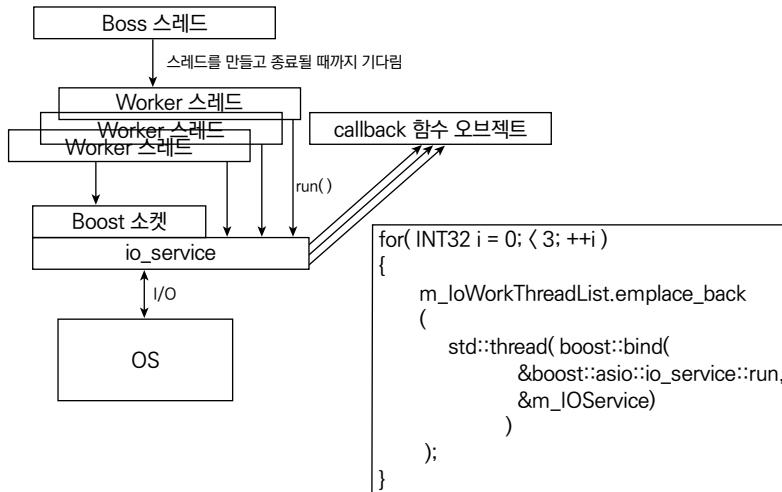


그림 9-2 비동기 n-스레드 모델



멀티 스레드로 사용하면 성능 면에서는 좋겠지만, 스레드 동기화에 따라 그만큼 주의해야 할 점이 늘어난다. 그러니 멀티 스레드 사용이 필수거나 만능이 아니라는 점을 꼭 기억하기 바란다.

9.1.1 boost::asio::io_service 서비스를 포함하는 설계

이 책의 예제는 Boost.Aasio의 io_service 객체를 모두 main() 함수에서 생성했다. 그러나 현장에서 프로그램을 만들 때는 기능별로 클래스를 만들기 때문에, io_service는 클래스에 포함될 것이다. io_service를 생성하는 방법에 따라 클래스를 두 가지 패턴으로 나눌 수 있다. 하나는 클래스에서 직접 io_service를 생성하는 패턴이고, 다른 하나는 외부에서 io_service를 인자로 받아서 사용하는 패턴이다.

얼핏 보면 별것 아니라고 생각할 수 있다. 대부분 클래스에서 직접 io_service를 생성하고 별문제가 없을 거라고 생각하지만, 경우에 따라 확장성에 문제가 생길 수 있다. Boost.Aasio는 최소한 하나 이상의 스레드로 io_service.run() 함수를 호출

해야 하는데, io_service 객체를 하나 만들 때마다 연관된 스레드를 하나 이상 만들어야 한다(생성한 스레드에서 io_service.run()을 호출해야 한다). 만약 직접 생성하는 클래스를 여러 개 만들어서 io_service를 사용한다면, 그만큼 스레드를 만들어야 하므로 스레드 생성에 따른 부하가 발생할 수 있다. 필자는 온라인 게임을 개발할 때 이 부분을 고려하지 않고 만들었다가, 이후에 더미 클라이언트를 만들 때 난감했던 경험이 있다(더미 클라이언트에서 클라이언트 객체를 100개 이상 생성해야 하는데, 스레드도 그만큼 만들어야 했다).

```
class Client
{
    std::shared_ptr< boost::asio::io_service > m_ptr_io_service;
    boost::asio::io_service & m_io_service;

public:
    // io_service를 인자로 받는 경우
    Client(boost::asio::io_service & io_service)
        :m_io_service(io_service)
    {
    }

    // io_service를 직접 생성
    Client()
        : m_ptr_io_service(new boost::asio::io_service()),
        m_io_service(*m_ptr_io_service)
    {
    }

    boost::asio::io_service & get_io_service() {return io_service_;}
};



---


```

9.1.2 boost::asio::io_service 클래스

다음은 Boost.Asio에서 자주 사용하는 멤버 함수다. 이를 숙지하면 프로그래밍하는 데 많은 도움이 될 것이다.

| `boost::asio::io_service` 클래스

소켓, 타이머 등 비동기 입출력 이벤트를 디스패치하는 클래스다.

```
| size_t run();  
| size_t run( boost::system::error_code& e );
```

run 함수는 모든 이벤트가 처리될 때까지 블록된다. 실행한 이벤트 개수를 반환한다.

```
| size_t run_one();  
| size_t run_one( boost::system::error_code& e );
```

run_one 함수는 하나의 이벤트가 처리될 때까지 블록된다. 실행한 이벤트 개수를 반환한다.

```
| size_t poll();  
| size_t poll( boost::system::error_code& e );
```

poll 함수는 이벤트가 처리될 때까지 블록하지 않으며, 처리할 이벤트를 모두 처리한 후에 종료된다. 이때 실행한 이벤트 개수를 반환한다. poll은 run과 달리 바로 사용할 수 있는 이벤트만을 처리하며, 그래서 IO 이벤트나 타이머 이벤트는 처리하지 못한다(이벤트를 즉시 처리할 수 없고 IO 완료 후나 시간이 지나야 가능하기 때문이다).

```
| size_t poll_one();  
| size_t poll_one( boost::system::error_code& e );
```

poll_one 함수는 이벤트가 처리될 때까지 블록하지 않으며, 한 개(또는 0개) 이벤트를 처리하고 종료된다. 이때 실행한 이벤트 개수를 반환한다.

```
| void stop();
```

run이나 poll 함수의 처리 루프를 정지할 때 stop 함수를 사용한다.

```
| void reset();
```

stop을 사용 후에 다시 run이나 poll을 사용하면 에러가 발생한다. 이때는 reset 함수를 사용해야 한다.

```
| boost::asio::io_service의 poll
```

poll은 run처럼 이벤트를 디스패치하지만, 이벤트를 처리할 때 대기하느냐, 아니면 즉시 처리할 이벤트를 처리하느냐 하는 차이점이 있다. poll이 run과 어떻게 다른지를 보여주는 다음 예제를 참고하기 바란다.

예제 9-1 poll

```
#include <SDKDDKVer.h>
#include <iostream>
#include <boost/asio.hpp>
#include <boost/asio/steady_timer.hpp>

void Function()
{
    std::cout << "Call Function !!!" << std::endl;
    //::Sleep(1000);
}

void OnTimer1( const boost::system::error_code& error )
{
    std::cout << "Call OnTimer1 !!! " << time(NULL) << std::endl;
}
```

```
int main()
{
    std::cout << "시작: " << time(NULL) << std::endl;

    boost::asio::io_service io_service;
    io_service.post( Function );
    io_service.post( Function );
    io_service.post( Function );

    boost::asio::steady_timer timer(io_service);
    timer.expires_from_now( boost::chrono::milliseconds(2000) );
    timer.async_wait( OnTimer1 );

    io_service.poll();

    std::cout << "종료: " << time(NULL) << std::endl;

    getchar();
    return 0;
}
```

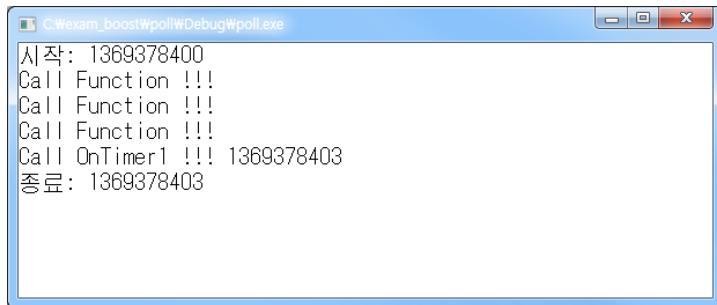
그림 9-3 예제 9-1의 실행 결과 화면



예제 9-1의 결과에서 보듯, poll을 사용한 경우 poll 호출 후 타이머로 등록한 OnTimer1는 즉시 사용할 수 없으므로 이벤트를 디스패치하지 않고 종료했다. 그러나 예제에 있는 Function() 함수의 Sleep에 걸어놓은 주석을 제거하면 그림 9-4와 같은 결과가 나온다.

```
void Function()
{
    std::cout << "Call Function !!!" << std::endl;
    ::Sleep(1000);
}
```

그림 9-4 Sleep을 사용한 poll 예제의 실행 결과 화면



처음 예제에서는 OnTimer1를 호출하려면 2초를 기다려야 해서 호출하지 못했지만, 두 번째 예제는 Function을 세 번 호출하는(각각 1초 소요) 동안 2초가 소요되어 OnTimer1을 바로 호출할 수 있게 되었다.

참고로 예제 9-1의 첫 번째 결과에서 이벤트를 디스패치하지 못해서 호출하지 못한 OnTimer1는 io_service.run() 함수를 호출하면 디스패치할 수 있다.

| boost::asio::io_service::work

비동기 요청을 하기 전에 `io_service::run()`을 호출하면 `io_service::run()`는 즉시 완료되어서 이후의 비동기 요청 작업을 처리할 수 없게 된다. 이것을 방지하기 위해 `io_service::work`를 사용한다.

`io_service::work`를 사용할 경우⁰² `work`가 파괴될⁰³ 때까지 `io_service::run()`은 종료하지 않으므로 언제든지 비동기 요청을 할 수 있다.

```
boost::shared_ptr<boost::asio::io_service::work> work_;  
work_.reset(new boost::asio::io_service::work(io_service_));
```

`io_service::work`의 사용 예는 예제 8-1을 참고하기 바란다.

9.1.3 strand 클래스

예제 9-1에서는 `io_service.run()`을 다음과 같이 멀티 스레드로 호출하였다.

```
for (std::size_t i = 0; i < size; ++i)  
{  
    m_Group.create_thread( boost::bind(&boost::asio::io_service::run,  
        &m_io_service) );  
}
```

그래서 아래와 같이 `Sleep`을 사용하여 1초 동안 대기 상태에 들어가는 Function 함수를 비동기로 두 번 실행해도, Function 함수는 한 번 실행된 후 1초 후에 다시 실행되지 않고 동시에 실행된다.

02 `run()` 호출 전에 비동기 요청이 없으면 `run()`은 대기하지 않고 바로 다음으로 진행하기 때문에 이후에 요청한 비동기 요청을 처리하지 못하지만, `work`를 사용하면 비동기 요청이 없어도 `run`에서 대기한다.

03 `work`의 인스턴스가 삭제된다는 의미다.

```
void Function( int nNumber )
{
    char szMessage[128] = {0,};
    sprintf_s( szMessage, 128-1, "%s(%d) | time(%d)", __FUNCTION__, nNumber,
               time(NULL) );
    {
        boost::mutex::scoped_lock lock(g_mutex);
        std::cout << "워커 스레드 ID: " << ::GetCurrentThreadId() << ". "
        << szMessage << std::endl;
    }

    ::Sleep(1000);
}

JobManager.post( boost::bind(Function, 11) );
JobManager.post( boost::bind(Function, 12) );
```

그러므로 Function 함수가 공용 리소스에 접근한다면 동기화 객체로 보호해야 한다. 위의 Function에서도 mutex를 사용하여 공용 리소스인 iostream을 보호하고 있다. 멀티 스레드 환경에서 Function 함수 자체를 동기화하고 싶을 때는 strand 클래스를 사용한다.

```
class strand
{
public:
    strand( io_service& io );

    template< typename Handler >
    unspecified wrap( Handler han );
};
```

warp 멤버 함수를 사용하여 strand 클래스의 동일한 인스턴스에 Function 함수를 등록하면, 멀티 스레드에서 병렬로 실행되지 않도록 동기화된 함수로 바꾸어준다. 예를 들어 strand의 인스턴스인 st에 Function 함수를 두 번 등록해도 boost::bind(Function, 11)이 끝난 후에 boost::bind(Function, 12)가 호출된다.

```
boost::asio::strand st( io_service );
JobManager.post( st.wrap( boost::bind(Function, 11) ) );
JobManager.post( st.wrap( boost::bind(Function, 12) ) );
```

반면, 다음과 같이 서로 다른 strand 인스턴스에서 Function 함수를 등록하면 boost::bind(Function, 11)와 boost::bind(Function, 12)는 동시에 호출된다.

```
boost::asio::strand st1( io_service );
JobManager.post( st1.wrap( boost::bind(Function, 11) ) );
boost::asio::strand st2( io_service );
JobManager.post( st2.wrap( boost::bind(Function, 12) ) );
```

사용 방법은 다음의 예제 9-2를 참고하기 바란다(예제 9-1과 거의 같아서 중요 부분만 가져왔다).

예제 9-2 strand 클래스를 사용한 멀티 스레드

```
void Function( int nNumber )
{
    char szMessage[128] = {0,};
    sprintf_s( szMessage, 128-1, "%s(%d) | time(%d)", __FUNCTION__, nNumber,
               time(NULL) );
    std::cout << "워커 스레드 ID: " << ::GetCurrentThreadId() << ". " <<
```

```

        szMessage << std::endl;
    }
    ::Sleep(1000);
}

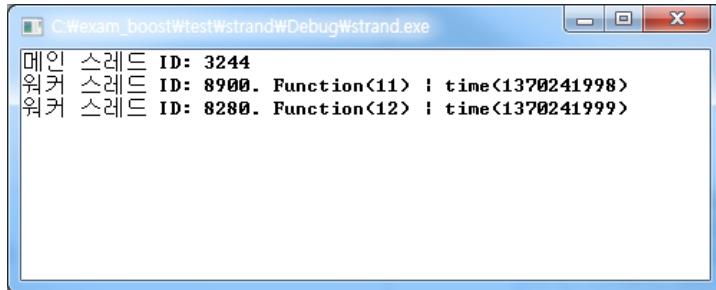
int main()
{
    std::cout << "메인 스레드 ID: " << ::GetCurrentThreadId() << std::endl;
    boost::asio::io_service io_service;
    BackGroundJobManager JobManager( io_service, 3 );

    boost::asio::strand st( io_service );
    JobManager.post( st.wrap( boost::bind(Function, 11) ) );
    JobManager.post( st.wrap( boost::bind(Function, 12) ) );

    ::Sleep(3000);
    return 0;
}

```

그림 9-5 예제 9-2의 실행 결과 화면



예제 8-1과는 달리, 멀티 스레드 환경에서도 Function 함수가 순차적으로 호출되고 있다. 다음 소스 코드는 예제 9-2의 핵심 부분이다.

```
boost::asio::strand st( io_service );
JobManager.post( st.wrap( boost::bind(Function, 11) ) );
JobManager.post( st.wrap( boost::bind(Function, 12) ) );
```

9.2 Windows에서 비동기로 파일 읽기

멀티 플랫폼을 지원하는 경우, 보통 각 OS의 특정 기능은 제외하고 서로 공통적인 부분만 지원한다. 그러나 Boost.Asio는 멀티 플랫폼을 지원하는 동시에 OS 고유의 기능 역시 지원해준다.

Boost.Asio는 Windows 플랫폼에서 비동기 파일 읽기를 지원한다. 예제 9-3은 비동기로 텍스트 파일을 읽어서 출력한다.

예제 9-3 AsyncReadFile

```
// 출처 : http://d.hatena.ne.jp/faith\_and\_brave/20110322/1300777903
```

```
#include <SDKDDKVer.h>

#include <iostream>
#include <vector>
#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/ref.hpp>

void read_end( const boost::system::error_code& error, const
std::vector<char>& result )
{
    if (error)
    {
        std::cout << error.message() << std::endl;
    }
}
```

```

        else
    {
        const std::string s(result.begin(), result.end());
        std::cout << "success : " << s << std::endl;
    }
}

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

    HANDLE handle = ::CreateFileA( "AsyncReadFile.txt",
                                GENERIC_READ,
                                FILE_SHARE_READ,
                                NULL,
                                OPEN_EXISTING,
                                FILE_FLAG_OVERLAPPED,
                                NULL );
}

if (handle == INVALID_HANDLE_VALUE)
{
    std::cout << "cannot open" << std::endl;
    return 0;
}

std::vector<char> buffer(::GetFileSize(handle, NULL));
boost::asio::windows::stream_handle file(io_service, handle);

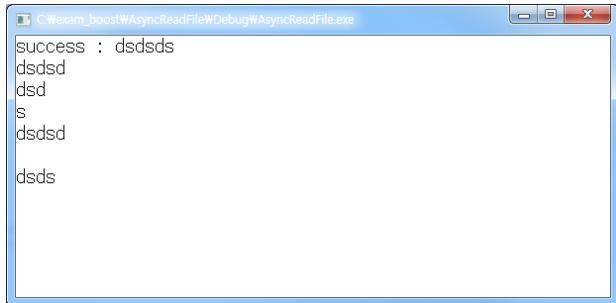
boost::asio::async_read( file,
                        boost::asio::buffer(buffer.data(), buffer.size()),
                        boost::bind(read_end, _1, boost::ref(buffer))
);

```

```
    io_service.run();

    getchar();
    ::CloseHandle(handle);
    return 0;
}
```

그림 9-6 예제 9-3의 실행 결과 화면



9.3 resolver을 사용하여 도메인 네임을 IP 주소로 변환하기

도메인 주소를 네트워크 IP 주소를 변환할 때는 boost::asio::ip::tcp::resolver 클래스를 사용한다. 동기식 방식과 비동기 방식, 두 가지가 있는데 여기서는 비동기 방식을 소개한다.

예제 9-4는 구글닷컴의 도메인 주소(google.com)를 네트워크 IP 주소로 변환하는 것이다.

예제 9-4 resolver

```
#include <SDKDDKVer.h>

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

void OnResolve(const boost::system::error_code& error, boost::asio::ip::tcp
::resolver::iterator endpoint_iterator)
{
    if (error) {
        std::cout << error.message() << std::endl;
    }
    else {
        boost::asio::ip::tcp::resolver::iterator end;

        for (; endpoint_iterator != end; ++endpoint_iterator)
        {
            std::cout << endpoint_iterator->endpoint().address().to_string() <<
            std::endl;
        }
    }
}

int main()
{
    boost::asio::io_service io_service;
    boost::asio::ip::tcp::resolver resolver(io_service);
    boost::asio::ip::tcp::resolver::query query("google.com", "http");

    resolver.async_resolve(query, boost::bind(OnResolve,
                                              boost::asio::placeholders::error,
```

```
        boost::asio::placeholders::iterator )  
    );  
    io_service.run();  
}
```

그림 9-7 예제 9-4의 실행 결과 화면

```
74.125.235.135  
74.125.235.136  
74.125.235.137  
74.125.235.142  
74.125.235.128  
74.125.235.129  
74.125.235.130  
74.125.235.131  
74.125.235.132  
74.125.235.133  
74.125.235.134  
2404:6800:4004:804::1000
```

9.3.1 특정 문자까지 비동기로 데이터 받기

Boost.Asio의 비동기 받기 중 지정한 문자를 받을 때 완료하는 함수가 있다. 이것을 사용하면 문자열을 보낼 때 유용하다. 예제 9-5는 데이터를 비동기로 받다가 줄 넘김 문자가 있으면 완료한다(별도로 서버를 만들어두지 않아서 실행 결과를 확인할 수 없다. 이렇게 사용한다는 정도로 짚고 넘어가길 바란다).

예제 9-5 `async_read_until`

```
#include <SDKDDKVer.h>  
  
#include <iostream>  
#include <string>  
#include <boost/ref.hpp>  
#include <boost/asio.hpp>
```

```

#include <boost/bind.hpp>

struct receiver
{
    typedef void result_type;

    void operator()(const boost::system::error_code& error,
                    boost::asio::streambuf& buffer)
    {
        if (error) {
            std::cout << error.message() << std::endl;
        }
        else {
            const std::string str = boost::asio::buffer_cast<const char*>
                (buffer.data());
            std::cout << str << std::endl;
        }
    }
};

int main()
{
    boost::asio::io_service io_service;
    boost::asio::ip::tcp::socket soc(io_service);

    boost::system::error_code connect_error;
    soc.connect(boost::asio::ip::tcp::endpoint(boost::asio::ip::
        address::from_string("127.0.0.1"), 31400), connect_error);

    if( connect_error )
    {

```

```
    std::cout << "서버에 접속 실패로 종료합니다" << std::endl;
    return 0;
}

boost::asio::streambuf buffer;
boost::asio::async_read_until(soc, buffer, '\n',
                             boost::bind( receiver(),
                             boost::asio::placeholders::error,
                             boost::ref(buffer) )
                           );
io_service.run();
}
```

9.3.2 비동기 핸들러 등록 시 메모리 풀 사용하기

Boost.Asio의 문서에 있는 예제 중 핸들러를 등록할 때 메모리 풀을 사용하는 예제가 있으니, 참고하기 바란다.

- http://www.boost.org/doc/libs/1_53_0/doc/html/boost_asio/example/allocation/server.cpp

10 | 참고 자료

- Boost 라이브러리 공식 웹 사이트

<http://www.boost.org>

- Boost.Asio 튜토리얼(Boost 1.5.3 버전 기준)

http://www.boost.org/doc/libs/1_53_0/doc/html/boost_asio/tutorial.html

- Boost.Asio 공식 문서(Boost 1.5.3 버전 기준)

http://www.boost.org/doc/libs/1_53_0/doc/html/boost_asio.html

- Boost와 독립적인 Asio 웹 사이트

<http://think-async.com/Asio>

- Boost 라이브러리 e-book 자료

<http://en.highscore.de/cpp/boost/>



01

유지보수하기 어렵게 코딩하는 방법

평생 개발자로 먹고 살 수 있다

로에디 그린 지음
우정은 옮김

이렇게 하면
개발자로 평생
먹고 살 수 있다

02

대용량 서버 구축을 위한 Memcached와 Redis

김대령 지음

대용량 서버 구축을 위한
분산 캐시의 이해!
간단한 Short URL 실습
으로 이해하는
분산 캐시 기술

03

스마트폰과 태블릿 환경을 위한 안드로이드 앱 프로그래밍

고강태 지음

스마트폰 앱을 태블릿 앱
으로 쉽고, 빠르게 전환
하고 싶으신가요?
다양한 기기와 호환하는
안드로이드 앱 개발의
모든 것!

04

프로젝트 성공을 위한 갑과 을의 상생협력

21명의 현직 전문가가 전하는

생생한 성공 노하우

이재용 지음

갑과 협력하기 어려우신가
요? 을과 협력하기
어려우신가요?
21명의 현직 전문가가
알려주는 갑과 을의
상생협력 전략

05

자바 개발자를 위한 험수형 프로그래밍

딘 월플러 지음 / 임백준 옮김
자바 개발자를 위한
험수형 프로그래밍 기법
험수형 프로그래밍 기법을
익힌 프로그래머와
그렇지 않은 프로그래머가
작성하는 코드의 품질은
완전히 다르다



06

일관성 있는 웹 서비스 인터페이스 설계를 위한 REST API 디자인 규칙

마크 미세 지음
김관례, 권원상 옮김
REST API를 사용하기
어려우신가요?
사전처럼 찾아 쓰는
REST API 규칙과 팁

07

웹 프로그래머를 위한 서블릿 컨테이너의 이해

최희탁 지음

웹 프로그래밍에 깊이를
더 하자!
서블릿 컨테이너를
제대로 알면
웹 프로그래밍이 쉬워진다

08

서비스스크립트와 SVG로 쉽게 만드는 웹 기반 데이터 비주얼라이제이션 D3

마이크 드워 지음

김보경 옮김
쉽게 배우는
웹 기반 데이터
비주얼라이제이션 D3

09

멀티스레드를 위한 자비스스크립트 프로그래밍 웹 워커

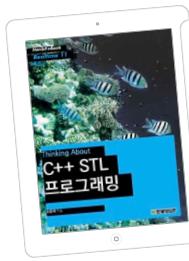
아이두 그린 지음

김보경 옮김
웹 애플리케이션에서
멀티스레드를 구현하는
가장 쉬운 방법.
웹 애플리케이션에
날개를 달자!

10

소프트웨어 생명 주기를 위한 원칙 Code Simplicity

맥스 카벗-알렉산더 지음
신정안 옮김
어떻게 해야
소프트웨어 설계를
잘할 수 있을까?
좋은 소프트웨어 설계는
무엇일까?



11

Thinking About
C++ STL
프로그래밍

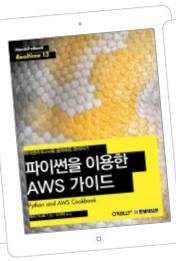
최종배 지음
C++ 프로그래밍에서는
STL은 필수다
STL을 아는 만큼
C++ 프로그래밍 스킬을
키울 수 있다



12

비데이터 처리를 위한
웹 개발 노하우
MongoDB와 PHP

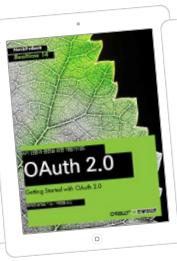
스티브 프랜시아 지음
최강현 옮김
MongoDB와
PHP의 만남.
빅데이터 처리
실전 노하우



13

파이썬과 Boto로
클라우드 관리하기
파이썬을 이용한
AWS 가이드

미치 가니트 지음
강권학 옮김
파이썬으로
AWS(아마존 웹 서비스)를
할 수 있다고요?
파이썬과 Boto로
AWS를 사용해보자



14

API 인증과 권한을 위한
개발 가이드
OAuth 2.0

라이언 보이드 지음
이정립 옮김
웹 API 인증을 위한
만능 키,
OAuth 2.0을 만나다



15

윤리적인 빅데이터 사용을 위한
정책 가이드
빅데이터 윤리

코드 데이터스 지음
강석주 옮김
빅데이터 시대!
윤리적인 책임은 없는가?
빅데이터 시대에서
지켜야 할 윤리 책임의
기준을 제시한다



16

윈도우 런타임을 이용한
실전 앱 개발
Windows 8 앱
개발 가이드

벤 뒤이 지음 / 이원영 옮김
윈도우 런타임
(Windows Runtime, WinRT)으로 시작하는
새로운 개발 세계



17

Xen으로 배우는
가상화 기술의 이해
CPU 가상화

박은병, 김태훈, 이상철,
문대혁 지음
번역/상화
전기상화와
전기상화 기술을 다루는
x86 이타틱처 기반의
CPU 가상화!



18

JSP 바이블
STEP 01
JSP 시작과
개발환경 구축

조효은 지음
JSP를 마스터하기 위한
길잡이
JSP를 위한 최고의 바이블
시리즈 첫 번째 책



19

빅데이터 시대의
개인정보보호와 사생활
프라이버시와 빅데이터

테렌스 크레이그,
메리 루들로프 지음
이준식, 김정환 옮김
빅데이터 시대!
개인정보와 프라이버시는
과연 보호받을 수 있을까?



20

프로그래머를 위한 가이드
드루팔

제니퍼 하지던 지음
김지원 옮김
드루팔 개발을 위한
원칙과 팁
이렇게 하면 실수를
피할 수 있다