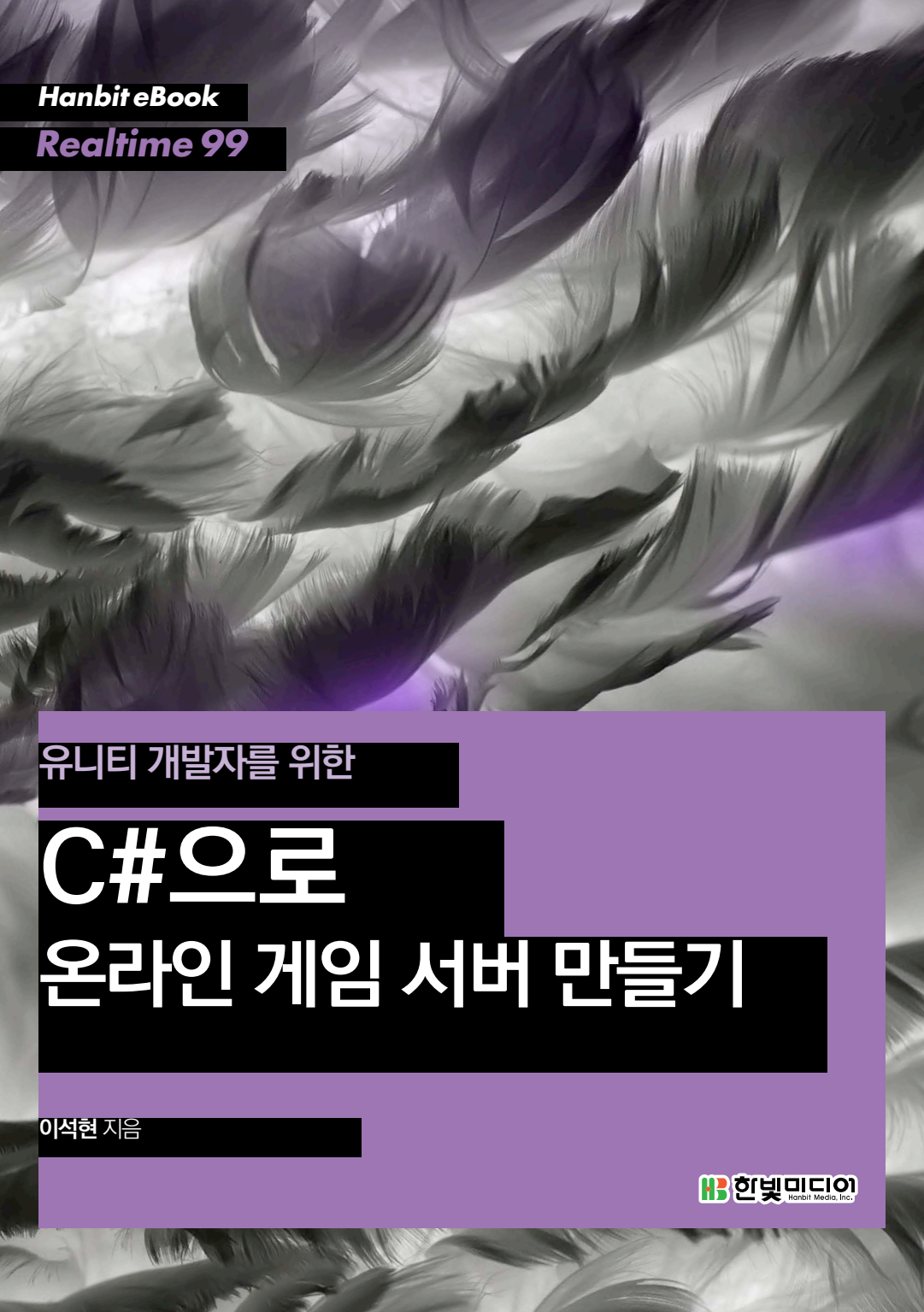

Hanbit eBook

Realtime 99

유니티 개발자를 위한

C#으로 온라인 게임 서버 만들기

이석현 지음

한빛미디어 Hanbit Media, Inc.

유니티 개발자를 위한

C#으로 온라인 게임 서버 만들기

이석현 지음

한빛미디어 Hanbit Media, Inc.

유니티 개발자를 위한 **C#으로 온라인 게임 서버 만들기**

초판발행 2015년 5월 7일

지은이 이석현 / **펴낸이** 김태헌
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 978-89-6848-748-4 15000 / **정가** 11,000원

총괄 배용석 / **책임편집** 김창수 / **기획·편집** 김상민
디자인 표지/내지 여동일, 조판 최송실
마케팅 박상용 / **영업** 김형진, 김진불, 조유미

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

저자 소개

지은이_ **이석현**

현재 (주)리젠소프트 프로그래머로 재직 중이며, 유니티 엔진과 C#을 이용한 모바일 실시간 온라인 게임 "최고의 마블스타"를 개발했다.

C++로 온라인 게임 콘텐츠 개발을 주로 해오다가 모바일 실시간 온라인 게임을 개발하게 되면서 C#으로 만드는 서버 개발에 관심이 생겼습니다.

이 책에서는 C#으로 온라인 게임 서버를 개발하는 방법을 소개합니다. 대상 독자는 유니티 엔진을 이용하여 클라이언트를 개발해온 프로그래머나 서버 개발에 관심은 있지만 어떻게 시작해야 할지 모르는 일반 독자들입니다.

닷넷 프레임워크에서 TCP 소켓 기능을 활용하는 방법을 소개한 뒤 간단한 에코 서버를 만들고 실시간 세균전 게임 서버를 제작해 볼 것입니다.

이 책의 내용으로 당장 서비스 가능한 게임 서버를 만들어낼 수는 없습니다. 실제로 서비스가 되는 게임 서버에는 단순히 소켓 기능만 존재하는 것이 아니기 때문입니다. 눈에 보이지 않은 수많은 노하우가 들어가고 무수한 테스트 과정을 거쳐야 안정된 게임 서버가 만들어지는 것입니다. 다만 게임 서버와 클라이언트가 어떤 원리로 통신하는지에 대해서는 이 책이 가이드 역할을 해줄 수 있을 것으로 생각합니다.

책이 나올 수 있게 도와주신 한빛미디어 관계자분들께 감사드립니다. 또한, 자신이 만든 소스코드를 공개해주신 온라인 서버 제작자 모임 카페[01] 회원분들께도 정말 감사드립니다. 이름도 얼굴도 잘 모르지만 공개해주신 코드를 통해서 저 또한 많이 배우고 이 책을 쓸 수 있게 되었습니다. 그리고 게임 개발자 커뮤니티인 게임코디[02]와 이곳에서 제 글을 보고 출판을 제안해주신 최흥배 님께도 감사드립니다.

마지막으로 ㈜리젠소프트에서 함께 재직 중인 선배이자 동료, 디자이너 이덕재 님, 기획자 김왕진 님, 대표 김효식 님께 감사드립니다.

01 http://cafe.naver.com/ongameserver/5486

02 http://www.gamecodi.com/

클라이언트는 유니티 엔진[01](버전 4.5.5f1)을 이용하여 개발했기 때문에 유니티 엔진을 설치해야 합니다. 무료 버전으로도 에디터 환경에서는 테스트할 수 있지만, 안드로이드 환경으로 빌드하여 소켓 기능을 사용하려면 프로 버전을 구매해야 합니다.

이 책을 집필 중인 현재 유니티 5 버전이 모든 기능을 포함하여 무료로 내려받을 수 있는 상태입니다. 하지만 이 책은 유니티 4 버전을 기준으로 작성되었으니 유니티 5 버전을 사용하는 독자들은 일부 오류가 발생할 수 있습니다.

이 책에서 사용하는 소스코드는 다음 경로에서 내려받을 수 있습니다.

- http://www.hanbit.co.kr/exam/2748

01 http://unity3d.com/kr/get-unity/download/archive

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 찾기도 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 First – 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 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 좀 더 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 위해 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 어려운 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 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.

차례

서버 제작의 기초

1

chapter 1

온라인 게임 서버를 만들기 위한 기초 지식

온라인 게임은 크게 서버와 클라이언트라는 두 가지 축으로 구성됩니다. 먼저 클라이언트에서는 주로 그래픽 처리를 다루며, 이와 관련된 코드를 작성합니다. 또한, 서버에서는 게임 로직이나 데이터베이스 등을 다루며, 이와 관련된 코드를 작성합니다. 이 둘은 물리적으로는 떨어져 있지만, 마치 옆에서 이야기를 주고받으며 온라인 게임 세계를 만들어 나가듯이 구현됩니다. 이렇게 서버와 클라이언트가 온라인 게임 세계를 위해 이야기를 주고받으려면 둘 사이를 연결해주는 기술이 필요한데, TCP Transmission Control Protocol (전송 제어 프로토콜)나 UDP User Datagram Protocol (사용자 데이터그램 프로토콜)라고 부르는 프로토콜을 주로 사용됩니다. 이 프로토콜을 통해 서버와 클라이언트는 하나로 연결될 수 있으며, 이를 통해 멀리 떨어진 유저 간에도 동시에 같은 게임을 즐길 수 있는 것입니다. 따라서 온라인 게임을 개발하기로 결정했다면 이런 통신 기술을 구현하는 방법을 반드시 숙지해야 합니다.

1.1 왜 C#으로 게임 서버를 구현하는가

최근 가장 인기 있는 게임 엔진인 유니티를 접해보신 독자라면 유니티에서 사용하는 C#에 매우 익숙할 겁니다. 유니티 엔진은 클라이언트 개발에 주로 사용됩니다. 하지만 유니티에서 사용하는 C#은 서버와 클라이언트에 두루 사용될 수 있고, 특히 닷넷 프레임워크의 기능을 활용하는 데 적합합니다. 또한, 아직은 개선해야

할 부분이 많이 있지만, 실시간 온라인 게임 서버 개발에도 사용됩니다. 이 책에서 C#을 선택한 가장 큰 이유가 C#이 유니티 엔진 덕분에 이미 많은 개발자들에게 익숙한 언어이기 때문입니다. 서버 개발은 클라이언트 개발과는 방식이나 관점이 달라 다소 생소할 수도 있지만, C#이라는 매개체를 통해 유니티만 사용하던 개발자들도 충분히 서버 개발에 도전할 수 있습니다.

1.2 네트워크 통신의 기초 지식

온라인 게임을 개발할 때 필수적이지만 어려운 부분이 바로 네트워크 통신 부분입니다. 일반적으로 대부분의 프로젝트에서 네트워크 통신 부분은 다양한 장르의 게임을 개발할 때마다 사용할 수 있도록 모듈화해 둡니다. 이는 개발하려는 게임마다 콘텐츠를 구현하는 방식은 제각각 다를 수 있지만, 네트워크 통신 모듈은 게임마다 비슷한 방법으로 구현하기 때문입니다. 다시 말해, 한번 모듈화해 두면 재활용이 가능하기 때문입니다.

이제부터 온라인 게임의 첫발을 내딛는 기분으로 가장 기본이 되는 네트워크 모듈에 대해 한 단계씩 살펴보겠습니다. 앞으로 나오는 내용을 학습하기 위해서는 소켓 프로그래밍의 개념에 대한 기초 지식이 필요합니다. 만약 소켓 프로그래밍을 직접 해본 경험이 없거나 어깨너머로 보기만 했다면 부록에 수록된 소켓 프로그래밍 기초 지식을 참조하여 기본적인 개념을 먼저 학습하고 다음 장을 시작하는 것을 권장합니다.

chapter 2

서버 네트워크 모듈 만들기

온라인 게임 서버에서 네트워크 모듈의 비중은 절대적이라고 할 수 있습니다. 게임의 콘텐츠가 이상 없이 구현될 수 있도록 보이지 않는 곳에서 묵묵히 자신의 일을 수행하는 것이 바로 네트워크 모듈의 역할입니다. 이 장에서는 소켓을 열고 클라이언트와 메시지를 주고받을 수 있는 서버 쪽 모듈을 만들어 볼 것입니다. 서버 네트워크 모듈의 출발점이 되는 CNetworkService 클래스부터 구현해 보겠습니다.

2.1 CNetworkService 클래스의 구성

1장에서 언급했듯이 온라인 게임에서는 TCP와 UDP를 주로 사용합니다. 이 책에서는 TCP 소켓만을 지원하는 모듈을 만들어 보겠습니다. CNetworkService 클래스에는 네트워크 통신을 위해 기반이 되는 코드가 들어갑니다. 여기에는 클라이언트의 접속을 기다리는 Listener 객체, 메시지 송/수신에 필요한 비동기 이벤트 객체, 메시지를 보관할 버퍼를 관리하는 버퍼 매니저 등이 있습니다. 먼저 멤버 변수부터 알아보겠습니다.

```
public class CNetworkService
{
    // 클라이언트의 접속을 받아들이기 위한 객체
    CListener client_listener;                                    ①

    // 메시지 수신, 전송 시 필요한 객체
    SocketAsyncEventArgsPool receive_event_args_pool;             ②
    SocketAsyncEventArgsPool send_event_args_pool;
```

```
// 메시지 수신, 전송 시 닷넷 비동기 소켓에서 사용할 버퍼를 관리하는 객체
BufferManager buffer_manager;                                    ③

// 클라이언트의 접속이 이루어졌을 때 호출되는 델리게이트
public delegate void SessionHandler(CUserToken token);           ④
public SessionHandler session_created_callback { get; set; }}

...
```

CNetworkService 클래스는 서버와 클라이언트 모두에서 사용할 수 있는 모듈이지만, 일단 서버 쪽에서 구현되는 원리부터 살펴보겠습니다.

① 클라이언트의 접속을 받아들이는 Listener 객체가 선언되어 있습니다. 이 클래스에 대해서는 2.2 CListener 클래스 구현하기에서 자세히 설명하겠습니다.

② SocketAsyncEventArgs라는 다소 생소한 클래스가 있습니다. 이 클래스는 닷넷 비동기 소켓에서 사용하는 개념으로 비동기 소켓 메서드를 호출할 때마다 반드시 필요한 객체입니다. 닷넷 3.5 이전 버전에서는 Begin ~ End 계열의 API를 사용했지만 3.5 이상 버전부터는 이 API를 사용하여 소켓 프로그래밍을 구현할 수 있습니다. Begin ~ End 계열의 API를 사용할 경우에는 매번 IAsyncResult라는 객체를 생성해야 했지만, 이 API는 객체를 풀링하여 사용할 수 있기 때문에 메모리를 재사용할 수 있다는 것이 장점입니다.

③ BufferManager는 이름에서도 알 수 있듯이 데이터 송/수신할 때 사용할 버퍼를 관리하는 매니저 객체입니다. TCP에서 데이터를 주고받을 때 소켓마다 버퍼가 할당됩니다. 이것은 OS에서 구현되는 부분이라 신경 쓸 필요는 없습니다. 이 소켓 버퍼로부터 메시지를 복사해 오고(수신) 밀어 넣는(전송) 작업을 할 때 사용할 버퍼를 설정하기만 하면 됩니다. 이 버퍼 역시 네트워크 통신이 지속되는 동안에 계속해서 사용하는 메모리입니다. 따라서 이것도 풀링하

여 메모리를 재사용하겠습니다.

④ 클라이언트가 접속했을 때 어딘가로 통보해주기 위한 수단으로 delegate를 사용했습니다. 이 클래스는 다음과 같은 작업을 수행합니다.

- 클라이언트의 접속을 받아들이기 위한 Listener 객체 설정
- 클라이언트의 접속이 이루어질 때 통보할 이벤트 처리
- 서버에 접속하는 모든 클라이언트에 대한 송/수신 데이터 버퍼 관리

서버는 접속할 클라이언트가 있어야만 의미가 있습니다. 따라서 제일 먼저 클라이언트와의 접속을 처리하는 부분을 구현하겠습니다. 클라이언트가 접속한 이후에는 서버와 클라이언트끼리 메시지를 주고받을 수 있는 환경을 마련해줘야 합니다.

이제 CListener 클래스를 구현하면서 서버가 클라이언트와의 연결을 어떻게 받아들이는지 알아봅시다.

2.2 CListener 클래스 구현하기

TCP 서버의 구현 흐름은 bind → listen → accept 순으로 진행됩니다.

```
class CListener
{
    // 비동기 Accept를 위한 EventArgs
    SocketAsyncEventArgs accept_args;

    // 클라이언트의 접속을 처리할 소켓
    Socket listen_socket;

    // Accept 처리의 순서를 제어하기 위한 이벤트 변수
    AutoResetEvent flow_control_event;

    // 새로운 클라이언트가 접속했을 때 호출되는 델리게이트
    public delegate void NewclientHandler(Socket client_socket, object
token);
```

```
    public NewclientHandler callback_on_newclient;

public CListener()
{
    this.callback_on_newclient = null;
}

    public void start(string host, int port, int backlog)            ①
    {
        // 소켓을 생성한다
        this.listen_socket = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Tcp);

        IPAddress address;
        if (host == "0.0.0.0")
        {
            address = IPAddress.Any;
        }
        else
        {
            address = IPAddress.Parse(host);
        }
        IPEndPoint endpoint = new IPEndPoint(address, port);         ②

        try
        {
            // 소켓에 host 정보를 바인딩시킨 뒤 Listen 메서드를 호출하여 대기한다.
            this.listen_socket.Bind(endpoint);                       ③
            this.listen_socket.Listen(backlog);

            this.accept_args = new SocketAsyncEventArgs();
            this.accept_args.Completed += new EventHandler<SocketAsyncEventArg
s>(on_accept_completed);

            // 클라이언트가 들어오기를 기다린다.
            // 비동기 메서드이므로 블로킹되지 않고 바로 리턴되며
            // 콜백 메서드를 통해서 접속 통보를 받는다.
            this.listen_socket.AcceptAsync(this.accept_args);        ④
        }
        catch (Exception e)
        {
```

```
            Console.WriteLine(e.Message);
        }
    }

    ...
```

Listen 처리 코드의 일부분입니다. 생소한 코드가 많겠지만 눈여겨봐야 할 중요한 부분은 몇 군데밖에 없습니다.

① start 메서드에 서버의 IP 주소와 포트 정보를 넣어서 호출하면 클라이언트가 접속할 수 있는 대기 상태가 됩니다. 이 클래스는 앞서 보여드린 CNetworkService 클래스에 멤버 변수로 포함되어 있습니다. 경우에 따라서는 Listener를 여러 개 두는 구현 방식도 고려할 수 있게 작성되었습니다. 이유는 하나의 서버가 반드시 하나의 포트로만 접속을 받아들이는 것은 아니기 때문입니다. 클라이언트의 접속을 받아들이는 Listener, 서버 간 통신을 위해서 다른 서버의 접속을 받아들이는 Listener 등 여러 개의 Listener가 존재할 수 있습니다.

② IPEndPoint 객체는 끝점을 의미하며 도착 지점이라고 이해하시면 됩니다. 클라이언트가 도착할 지점은 서버가 됩니다. 서버의 IP, Port 정보로 IPEndPoint가 구성됩니다. 이 서버가 Host가 되며 클라이언트는 Peer라고 부릅니다.

③ bind → listen 순으로 진행됩니다. 서버 정보를 소켓에 바인드시키고 listen을 호출하여 클라이언트가 접속할 수 있게 만들어 줍니다.

④ AcceptAsync 메서드를 호출하면 서버가 대기 상태에 있다가 클라이언트가 접속하는 순간 콜백 메서드로 통지가 오게 됩니다.

이제 코드의 내부로 들어가 좀 더 세부적인 부분을 살펴보겠습니다.

```
...

this.accept_args = new SocketAsyncEventArgs();
this.accept_args.Completed += new EventHandler<SocketAsyncEventArgs>(on_
accept_completed);

...
```

코드 중간을 보면, 드디어 앞에서 언급한 SocketAsyncEventArgs라는 객체를 사용합니다. 사용하는 방법은 Completed 프로퍼티에 이벤트 핸들러 객체를 연결해 주고 AcceptAsync를 호출할 때 파라미터로 넘겨주기만 하면 됩니다. Completed라는 이름에서 알 수 있듯이 accept 처리가 완료되었을 때 호출되는 델리게이트입니다. 닷넷 비동기 소켓은 이처럼 메서드 호출 → 완료 통지 순서로 이루어집니다.

```
...
this.listen_socket.AcceptAsync(this.accept_args);
...
```

좀 더 밑으로 내려가 accept 처리 부분을 살펴보겠습니다. 이 책에서는 비동기 메서드를 사용하므로 AcceptAsync를 호출합니다. 이 메서드는 호출한 직후 바로 리턴되며 accept 결과에 대해서는 콜백 메서드로 통보가 옵니다. 따라서 프로그램이 블로킹되지 않고 통보를 기다리면서 다른 작업들을 수행할 수 있는 상태가 됩니다. AcceptAsync까지 호출하면 클라이언트의 접속을 받아들일 수 있는 상태가 됩니다.

방금 우리는 간단하지만 사용할 수 있는 서버 하나를 만들었습니다.

2.3 스레드를 통해 Accept 처리하기

이번에는 앞 절에서의 AcceptAsync 메서드 호출 부분을 스레드로 처리하도록 바꿔 보겠습니다. 꼭 스레드를 통하지 않고도 accept를 처리할 수 있습니다. 하지만 특정 OS 버전에서 콘솔 입력이 대기 중일 때 accept 처리가 되지 않는 버그가 발생할 수도 있습니다[01].

메인 스레드가 입력을 위해 대기 상태에 있다고 하더라도 accept를 별도의 스레드에서 처리하게 구성한다면 앞의 문제를 회피할 수 있습니다. 따라서 스레드로 accept를 처리해 보겠습니다.

```
...
this.listen_socket.AcceptAsync(this.accept_args);
...
```

앞 부분을 다음처럼 바꿔보겠습니다.

```
Thread listen_thread = new Thread(do_listen);
listen_thread.Start();
```

스레드를 하나 생성하고 do_listen이라는 메서드를 스레드에서 처리하라는 의미입니다.

```
void do_listen()
{
    // accept 처리 제어를 위해 이벤트 객체를 생성한다.
    this.flow_control_event = new AutoResetEvent(false);          ①

    while (true)                                                  ②
    {
```

01 http://goo.gl/MONp9F

```
        // SocketAsyncEventArgs를 재사용하기 위해서 null로 만들어 준다.
        this.accept_args.AcceptSocket = null;

        bool pending = true;
        try
        {
            // 비동기 accept를 호출하여 클라이언트의 접속을 받아들인다.
            // 비동기 메서드이지만 동기적으로 수행이 완료될 경우도 있으니
            // 리턴 값을 확인하여 분기 처리를 해줘야 한다.
            pending = listen_socket.AcceptAsync(this.accept_args);     ③
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
            continue;
        }

        // 즉시 완료(리턴 값이 false일 때)가 되면
        // 이벤트가 발생하지 않으므로 콜백 메서드를 직접 호출해줘야 한다.
        // pending 상태라면 비동기 요청이 들어간 상태라는 뜻이며 콜백 메서드를 기다린다.
        // 참고: https://goo.gl/4b99VW
        if (!pending)
        {
            on_accept_completed(null, this.accept_args);
        }

        // 클라이언트 접속 처리가 완료되면 이벤트 객체의 신호를 전달받아 다시 루프를 수행한다.
        this.flow_control_event.WaitOne();                              ④
    }
}

this.flow_control_event = new AutoResetEvent(false);
```

① 스레드의 시작 부분에는 이벤트 객체를 생성하는 코드가 들어있습니다. 하나의 접속 처리가 완료된 이후 그다음 접속 처리를 수행하기 위해서 스레드의 흐름을 제어할 필요가 있는데, 이때 사용되는 이벤트 객체입니다.

② 루프를 돌며 클라이언트의 접속을 받아들입니다.

③ AcceptAsync의 리턴 값에 따라 즉시 완료 처리를 할 것인지 통보가 오기를 기다릴 것인지 구분해 줘야 합니다. Accept 처리가 동기적으로 수행이 완료된 경우에는 콜백 메서드가 호출되지 않고 false를 리턴합니다. 따라서 이 경우에는 완료 처리를 담당하는 메서드를 직접 호출해줘야 합니다. 그 외의 경우(true를 리턴)에는 닷넷 프레임워크에서 콜백 메서드를 호출해주기 때문에 직접 호출할 필요가 없습니다. 이 메서드뿐만 아니라 다른 비동기 메서드를 호출할 때는 즉시 완료될 경우와 그렇지 않을 경우를 구분해서 처리해줘야 합니다.

④ AcceptAsync를 통해서 하나의 클라이언트가 접속되기를 기다린 후 이벤트 객체를 이용하여 스레드를 잠시 대기 상태로 둡니다. 이벤트 객체는 두 가지가 있습니다. AutoResetEvent는 한 번 Set이 된 이후 자동으로 Reset 상태로 만들어주며, ManualResetEvent는 직접 Reset 메서드를 호출하지 않는다면 계속 Set 상태로 남아있습니다. 두 가지 이벤트 객체는 이러한 차이점이 있으니 상황에 맞게 사용하면 됩니다. 여기서는 AutoResetEvent를 사용했습니다.

```
void on_accept_completed(object sender, SocketAsyncEventArgs e)
{
    if (e.SocketError == SocketError.Success)
    {
        // 새로 생긴 소켓을 보관해 놓은 뒤
        Socket client_socket = e.AcceptSocket;

        // 다음 연결을 받아들인다.
        this.flow_control_event.Set();

        // 이 클래스에서는 accept까지의 역할만 수행하고 클라이언트의 접속 이후의 처리는
        // 외부로 넘기기 위해서 콜백 메서드를 호출해 주도록 한다.
        // 그 이유는 소켓 처리부와 콘텐츠 구현부를 분리하기 위해서다.
        // 콘텐츠 구현 부분은 자주 바뀔 가능성이 있지만, 소켓 Accept 부분은
        // 상대적으로 변경이 적은 부분이기 때문에 양쪽을 분리시켜 주는 것이 좋다.
        // 또한, 클래스 설계 방침에 따라 Listen에 관련된 코드만 존재하도록
        // 하기 위한 이유도 있다.
```

```
            if (this.callback_on_newclient != null)
            {
                this.callback_on_newclient(client_socket, e.UserToken);
            }

        return;
        }
        else
        {
        //Accept 실패 처리.
        Console.WriteLine("Failed to accept client.");
        }

        // 다음 연결을 받아들인다.
        this.flow_control_event.Set();
    }
```

AcceptAsync 호출 결과를 통보받을 on_accept_completed 메서드의 코드입니다. 파라미터로 넘어온 값을 비교하여 성공, 실패에 대한 처리를 구현해 줍니다. 성공했을 경우에는 자동으로 소켓이 하나 생성되는데, 이 소켓을 잘 보관해 놓았다가 클라이언트와 통신할 때 사용하면 됩니다. 그리고 콜백 메서드를 호출하여 성공했음을 알려준 뒤 다음 연결을 받아들이기 위해서 이벤트 객체를 Set 상태로 만들어 줍니다. 코드 흐름을 따라가 보면 잠시 대기해 있던 스레드가 이벤트 객체의 신호를 받아 다시 AcceptAsync 메서드를 호출하는 방식으로 진행된다는 것을 알 수 있습니다.

2.4. SocketAsyncEventArgs 객체 생성하기

Accept 처리가 완료되었을 때 on_new_client 델리게이트를 호출해주는 부분까지가 CListener 클래스의 역할이었습니다.

이제 CNetworkService 클래스에 대해서 좀 더 자세히 살펴보겠습니다. 앞서 설

명한 대로 CNetworkService 클래스에는 네트워크 기반이 되는 코드가 들어가게 됩니다.

2.4.1 Listen 처리하기

```
public void listen(string host, int port, int backlog)
{
    CListener listener = new CListener();
    listener.callback_on_newclient += on_new_client;
    listener.start(host, port, backlog);
}
```

listener 생성 부분입니다. 서버의 host, port, backlog 값을 전달 받아 listener를 생성한 뒤 start 메서드를 호출하여 클라이언트의 접속을 기다립니다.

2.4.2 SocketAsyncEventArgs 풀링 구현하기

소켓별로 두 개의 SocketAsyncEventArgs가 필요합니다. 하나는 전송용, 다른 하나는 수신용입니다. 그리고 SocketAsyncEventArgs마다 버퍼를 필요로 하는데, 결국 하나의 소켓에 전송용 버퍼 한 개, 수신용 버퍼 한 개 총 두 개의 버퍼가 필요합니다.

먼저, SocketAsyncEventArgs를 어떻게 풀링하여 사용하는지 알아보겠습니다.

```
public class CNetworkService
{

…

    // 메시지 수신용 풀.
    SocketAsyncEventArgsPool receive_event_args_pool;
```

```
    // 메시지 전송용 풀.
    SocketAsyncEventArgsPool send_event_args_pool;

    ...
```

전송용, 수신용 풀 객체를 각각 선언합니다. 풀링에 사용되는 클래스는 SocketAsyncEventArgsPool입니다. 이 코드는 MSDNMicroSoft Developer Network(마이크로소프트 개발자 네트워크)[02]에 올라와 있는 샘플 코드[03]를 그대로 사용하였습니다.

```
// Represents a collection of reusable SocketAsyncEventArgs objects.
class SocketAsyncEventArgsPool
{
    Stack<SocketAsyncEventArgs> m_pool;

    // Initializes the object pool to the specified size
    //
    // The "capacity" parameter is the maximum number of
    // SocketAsyncEventArgs objects the pool can hold
    public SocketAsyncEventArgsPool(int capacity)
    {
        m_pool = new Stack<SocketAsyncEventArgs>(capacity);
    }

    // Add a SocketAsyncEventArg instance to the pool
    //
    //The "item" parameter is the SocketAsyncEventArgs instance
    // to add to the pool
    public void Push(SocketAsyncEventArgs item)
    {
        if (item == null) { throw new ArgumentNullException("Items added to a
SocketAsyncEventArgsPool cannot be null"); }
```

02 https://msdn.microsoft.com/ko-kr/dn308572.aspx

03 http://goo.gl/RXgSOQ

```
        lock (m_pool)
        {
            m_pool.Push(item);
        }
    }

    // Removes a SocketAsyncEventArgs instance from the pool
    // and returns the object removed from the pool
    public SocketAsyncEventArgs Pop()
    {
        lock (m_pool)
        {
            return m_pool.Pop();
        }
    }

    // The number of SocketAsyncEventArgs instances in the pool
    public int Count
    {
        get { return m_pool.Count; }
    }
}
```

코드 자체는 굉장히 간단합니다. 객체를 담을 수 있는 Stack을 생성하여 Pop/Push 메서드를 통해서 꺼내오고 반환하는 작업을 수행합니다.

```
this.receive_event_args_pool = new SocketAsyncEventArgsPool(this.max_
connections);
this.send_event_args_pool = new SocketAsyncEventArgsPool(this.max_
connections);
```

허용 가능한 최대 동시 접속 수치만큼 객체를 생성합니다.

```
for (int i = 0; i < this.max_connections; i++)
{
    CUserToken token = new CUserToken();
```

```
    // receive pool
    {
        //Pre-allocate a set of reusable SocketAsyncEventArgs
        arg = new SocketAsyncEventArgs();
        arg.Completed += new EventHandler<SocketAsyncEventArgs>(receive_
completed);
        arg.UserToken = token;

        // add SocketAsyncEventArg to the pool
        this.receive_event_args_pool.Push(arg);
    }


    // send pool
    {
        //Pre-allocate a set of reusable SocketAsyncEventArgs
        arg = new SocketAsyncEventArgs();
        arg.Completed += new EventHandler<SocketAsyncEventArgs>(send_
completed);
        arg.UserToken = token;

        // add SocketAsyncEventArg to the pool
        this.send_event_args_pool.Push(arg);
    }
}
```

SocketAsyncEventArgs 객체를 미리 생성하여 풀에 넣어두는 코드입니다. Completed 속성으로 수신, 전송이 완료되었을 때 호출하게 될 메서드를 입력해 줍니다. 수신, 전송 모두 같은 클라이언트에서 일어나는 이벤트이므로 CUserToken 클래스의 인스턴스는 하나만 생성하여 양쪽 모두에 똑같이 설정해 주었습니다.

2.5 송, 수신 버퍼 풀링 기법

다음으로 버퍼 관리에 대해 알아보겠습니다. 앞에서 SocketAsyncEventArgs마다 버퍼가 하나씩 필요하다고 설명했습니다. 이 버퍼라는 것은 바이트 배열로 이루어진 메모리 덩어리입니다.

```
BufferManager buffer_manager;
this.buffer_manager = new BufferManager(this.max_connections * this.buffer_
size * this.pre_alloc_count, this.buffer_size);
```

버퍼 매니저를 선언하고 생성하는 코드입니다. 버퍼의 전체 크기는 다음 공식으로 계산됩니다.

버퍼의 전체 크기 = 최대 동시 접속 수치 × 버퍼 하나의 크기 × (전송용 1개 + 수신용 1개)

전송용 한 개, 수신용 한 개 총 두 개가 필요하기 때문에 pre_alloc_count = 2로 설정해 놨습니다. 그럼, 이제 버퍼 매니저의 코드를 살펴보겠습니다. 이 코드 역시 MSDN에 있는 샘플 코드[04]를 가져온 것입니다.

```
/// <summary>
/// This class creates a single large buffer which can be divided up and
assigned to SocketAsyncEventArgs objects for use
/// with each socket I/O operation.  This enables bufffers to be easily
reused and gaurds against fragmenting heap memory.
///
/// The operations exposed on the BufferManager class are not thread safe.
/// </summary>
internal class BufferManager
{

    int m_numBytes;                 // the total number of bytes controlled
by the buffer pool
```

04 http://goo.gl/ipHRDR

```
    byte[] m_buffer;                // the underlying byte array maintained
by the Buffer Manager
    Stack<int> m_freeIndexPool;     //
    int m_currentIndex;
    int m_bufferSize;

    public BufferManager(int totalBytes, int bufferSize)
    {
        m_numBytes = totalBytes;
        m_currentIndex = 0;
        m_bufferSize = bufferSize;
        m_freeIndexPool = new Stack<int>();
    }

    /// <summary>
    /// Allocates buffer space used by the buffer pool
    /// </summary>
    public void InitBuffer()                                        ①
    {
        // create one big large buffer and divide that out to each
SocketAsyncEventArg object
        m_buffer = new byte[m_numBytes];
    }

    /// <summary>
    /// Assigns a buffer from the buffer pool to the specified
SocketAsyncEventArgs object
    /// </summary>
    /// <returns>true if the buffer was successfully set, else false</returns>
    public bool SetBuffer(SocketAsyncEventArgs args)                ②
    {
        if (m_freeIndexPool.Count > 0)
        {
            args.SetBuffer(m_buffer, m_freeIndexPool.Pop(), m_bufferSize);
        }
        else
        {
            if ((m_numBytes - m_bufferSize) < m_currentIndex)
            {
                return false;
            }
```

```
            args.SetBuffer(m_buffer, m_currentIndex, m_bufferSize);
            m_currentIndex += m_bufferSize;
        }
        return true;
    }

    /// <summary>
    /// Removes the buffer from a SocketAsyncEventArg object.  This frees the
buffer back to the
    /// buffer pool
    /// </summary>
    public void FreeBuffer(SocketAsyncEventArgs args)            ③
    {
        m_freeIndexPool.Push(args.Offset);
        args.SetBuffer(null, 0, 0);
    }
}
```

① InitBuffer 메서드에서 하나의 거대한 바이트 배열을 생성합니다.

②SetBuffer(SocketAsyncEventArgs args) 메서드에서 SocketAsyncEventArgs 객체에 버퍼를 설정해 줍니다. 하나의 버퍼를 설정한 다음에는 index 값을 증가시켜 다음 버퍼 위치를 가리킬 수 있도록 처리합니다. 넓은 땅에 금을 그어서 이쪽은 내 것, 저쪽은 네 것하는 식으로 나눈다고 생각하면 이해하기 쉽습니다.

③ FreeBuffer 메서드는 사용하지 않는 버퍼를 반환시키기 위한 일을 수행합니다. 이 메서드는 지금 우리가 만들고 있는 네트워크 모듈에서는 사용하지 않는 메서드가 될 것 같습니다. 왜냐하면 프로그램을 시작할 때 최대 동시 접속 수치만큼 버퍼를 할당한 뒤 중간에 해제하지 않고 계속 물고 있을 것이기 때문입니다. SocketAsyncEventArgs만 풀링하여 재사용할 수 있도록 처리해 놓으면 이 객체에 할당된 버퍼도 같이 따라가게 되기 때문이죠.

이제 이 버퍼 매니저를 어떻게 활용하는지 알아보겠습니다. 이전에 Socket

AsyncEventArgs를 생성하여 풀링 처리 했던 부분에 버퍼 매니저의 코드가 추가됩니다.

```
for (int i = 0; i < this.max_connections; i++)
{
    CUserToken token = new CUserToken();

    // receive pool
    {
        //Pre-allocate a set of reusable SocketAsyncEventArgs
        arg = new SocketAsyncEventArgs();
        arg.Completed += new EventHandler<SocketAsyncEventArgs>(receive_
completed);
        arg.UserToken = token;

        // 추가된 부분.
        // assign a byte buffer from the buffer pool to the
SocketAsyncEventArg object
        this.buffer_manager.SetBuffer(arg);                              ①

        // add SocketAsyncEventArg to the pool
        this.receive_event_args_pool.Push(arg);
    }


    // send pool
    {
        //Pre-allocate a set of reusable SocketAsyncEventArgs
        arg = new SocketAsyncEventArgs();
        arg.Completed += new EventHandler<SocketAsyncEventArgs>(send_
completed);
        arg.UserToken = token;

        // 추가된 부분.
        // assign a byte buffer from the buffer pool to the
SocketAsyncEventArg object
        this.buffer_manager.SetBuffer(arg);

        // add SocketAsyncEventArg to the pool
        this.send_event_args_pool.Push(arg);
```

```
    }
}
```

① this.buffer_manager.SetBuffer(arg);라는 코드가 보이시나요? 이 부분이 바로 SocketAsyncEventArgs 객체에 버퍼를 설정하는 코드입니다. SetBuffer 내부를 보면 범위를 잡아서 arg.SetBuffer를 호출하게 되어 있죠. m_currentIndex += m_bufferSize; 이런 식으로 버퍼 크기만큼 인덱스 값을 늘려서 서로 겹치지 않게 해주고 있는 겁니다. 다시 정리해 보면, 하나의 거대한 버퍼를 만들고 버퍼 사이즈만큼 범위를 잡아 SocketAsyncEventArgs 객체에 하나씩 할당해 주는 것입니다. 이런 구현 방식으로 SocketAsyncEventArgs 객체와 버퍼 메모리를 풀링하여 사용합니다.

닷넷 프레임워크에는 가비지 컬렉터가 작동되므로 풀링하지 않아도 객체 참조 관계만 잘 끊어주면 알아서 메모리를 정리해 줍니다. 하지만 그 시점은 프로그래머가 알 수 없으며 임의로 메모리 정리를 수행하는 것은 권하지 않는 방식입니다. 가비지 컬렉터가 작동되는 시점이 우리가 만들 서버에 잘 들어맞는다는 보장이 없기에 여기서는 마음 편하게 풀링하는 쪽을 선택하겠습니다. 서버가 살아 있는 동안 거의 매번 쓰이게 될 메모리라는 것도 풀링을 선택하는 데 하나의 기준이 되었죠.

2.6 CUserToken 클래스

클라이언트의 접속을 받아들이기 위해 Listener를 설정하고 AcceptAsync 메서드를 통해서 접속을 수락하는 코드까지 작성해 봤습니다. 접속한 클라이언트와 메시지를 주고받는 데 필요한 SocketAsyncEventArgs 객체도 풀링 처리하여 준비를 마쳤습니다. 단순한 서버라면 이런 준비과정이 필요 없을 수 있지만, 수천의 동시 접속을 처리하는 게임 서버라면 조금 번거롭더라도 이런 작업들이 꼭 필요합니다.

또한, 만들어 놨다고 끝난 것이 아니라 그때부터 시작인 셈입니다. 처음 설계한 대로 제대로 코딩을 해놨어도 실제 테스트를 해보면 부족한 부분이 보일 때가 있기 때문이죠. 디버깅하면서 뭔가 개운하지 못한 느낌이 든다면 설계를 다시 할 각오도 해야 합니다(가끔 귀찮을 때는 흔히 땜빵 코드라고 부르는 임시방편을 사용하기도 합니다).

이제 클라이언트의 접속을 받아들인 후 해야 할 일들에 대해서 작성해 보겠습니다. CUserToken 클래스는 접속한 클라이언트 하나하나에 대응되는 객체입니다. 접속자가 천 명이라면 CUserToken 객체도 천 개가 생기게 됩니다. 또한, 이 객체를 통해서 어느 클라이언트가 메시지를 전송했는지 실제적으로 어느 클라이언트에 메시지를 전송해야 하는지를 알 수 있게 됩니다. 먼저 클라이언트의 접속이 이루어진 후 호출되는 콜백 메서드인 on_new_client를 살펴보겠습니다.

```
void on_new_client(Socket client_socket, object token)          ①
{
    // 풀에서 하나 꺼내와 사용한다.
    SocketAsyncEventArgs receive_args = this.receive_event_args_pool.Pop(); ②
    SocketAsyncEventArgs send_args = this.send_event_args_pool.Pop();

    // SocketAsyncEventArgs를 생성할 때 만들어 두었던 CUserToken을 꺼내와서
    // 콜백 메서드의 파라미터로 넘겨준다.
    if (this.session_created_callback != null)
    {
        CUserToken user_token = receive_args.UserToken as CUserToken;    ③
        this.session_created_callback(user_token);
    }

    // 클라이언트로부터 데이터를 수신할 준비를 한다.
    begin_receive(client_socket, receive_args, send_args);          ④
}
```

① Accept 처리가 완료된 후 생성된 새로운 소켓을 on_new_client의 파라미터로 넘겨받습니다. 앞으로 이 소켓으로 클라이언트와 메시지를 주고받게 될 것입니다.

② 드디어 이전에 풀링해놨던 SocketAsyncEventArgs 객체를 사용할 때가 왔습니다. 메시지 수신용, 전송용 객체를 풀에서 각각 하나씩 꺼냅니다.

③ CUserToken 클래스는 원격지 클라이언트 하나당 한 개씩 매칭되는 유저 객체라고 생각하면 됩니다.

④ 클라이언트가 접속한 이유는 서버와 무언가를 주고받기 위함이죠. 이제 클라이언트로부터 메시지 수신을 위한 작업을 시작하겠습니다.

CNetworkService 클래스의 begin_receive 메서드의 코드를 살펴보겠습니다.

```
void begin_receive(Socket socket, SocketAsyncEventArgs receive_args,
SocketAsyncEventArgs send_args)                                    ①
{
    // receive_args, send_args 아무곳에서나 꺼내와도 된다. 둘다 동일한 CUserToken을
물고 있기 때문이다.
    CUserToken token = receive_args.UserToken as CUserToken;      ②
    token.set_event_args(receive_args, send_args);

    // 생성된 클라이언트 소켓을 보관해 놓고 통신할 때 사용한다.
    token.socket = socket;

    // 데이터를 받을 수 있도록 수신 메서드를 호출해준다.
    // 비동기로 수신될 경우 워커 스레드에서 대기 중으로 있다가 Completed에 설정해놓은 메
서드가 호출된다.
    // 동기로 완료될 경우에는 직접 완료 메서드를 호출해줘야 한다.
    bool pending = socket.ReceiveAsync(receive_args);              ③
    if (!pending)
    {
        process_receive(receive_args);
    }
}
```

닷넷 비동기 소켓 메서드는 대부분 사용법이 비슷합니다. xxxAsync 메서드를 호출한 뒤 리턴 값을 확인하여 콜백 메서드를 기다릴지, 직접 완료 처리를 할지 결정해주는 부분은 이전에 accept 처리 부분과 흡사합니다.

① 먼저 CUserToken 객체에 수신용, 전송용 SocketAsyncEventArgs를 설정해 줍니다.

② 메시지 송, 수신 시 항상 이 SocketAsyncEventArgs가 따라다니게 되기 때문에 작업 편의를 위해서 CUserToken 객체에도 설정해 주었습니다.

③ 마지막으로 ReceiveAsync 메서드를 호출해주면 해당 소켓으로부터 메시지를 수신받을 수 있게 됩니다. 비동기 메서드니 리턴 값을 꼭 확인해야 한다는 점 잊지 말길 바랍니다.

지금까지의 내용들을 처음 접하는 독자라면 잘 이해되지 않는 부분도 많을 겁니다. 특히 CListener부터 SocketAsyncEventArgsPool, BufferManager, UserToken 등 많은 클래스가 서로 관계를 맺도록 구조가 잡혀있는데, 이런 시스템이 한번에 이해되긴 어렵습니다. 저도 처음 라이브러리를 설계할 때 모두 완성해놓고 코딩한 것은 아닙니다. 하나의 클래스에 단순하게 구현해본 뒤 점점 확장/분리할 필요성을 느끼게 되면서 다시 설계하는 작업을 여러 번 반복한 결과입니다. 따라서 처음에는 흐름을 파악한다고 생각하고 직접 코딩해 보면서 다시 책의 내용을 참고하는 방법을 권해드립니다.

다음은 ReceiveAsync 메서드 호출 이후에 콜백으로 호출되는 메서드입니다.

```
void receive_completed(object sender, SocketAsyncEventArgs e)
{
    if (e.LastOperation == SocketAsyncOperation.Receive)
    {
        process_receive(e);                                    ①
        return;
    }

    throw new ArgumentException("The last operation completed on the socket
was not a receive.");
}
```

① 오류가 났을 때 예외를 던져주는 부분만 빼면 그냥 process_receive 메서드를 호출해주는 것밖에 없습니다. 사실 우리가 작성했던 코드에서는 e.LastOperation이 Receive가 아닌 경우는 발생하지 않습니다. 왜냐하면 메시지 송, 수신 완료 처리를 각각 별도의 메서드에서 분리되어 호출되도록 구성하였기 때문이죠. 만약 메시지 송, 수신 완료 처리를 하나의 콜백 메서드에서 구현하려고 할 경우가 있다면 LastOperation 값을 참조하여 분기하는 코드를 작성해야 할 것입니다.

이제 process_receive 메서드를 살펴보겠습니다(메시지 하나 받는 데 참 많은 메서드를 거치게 되는군요).

```
// This method is invoked when an asynchronous receive operation completes.
// If the remote host closed the connection, then the socket is closed.
//
private void process_receive(SocketAsyncEventArgs e)                    ①
{
    // check if the remote host closed the connection
    CUserToken token = e.UserToken as CUserToken;
    if (e.BytesTransferred > 0 && e.SocketError == SocketError.Success)
    {
        // 이후의 작업은 CUserToken에 맡긴다.
        token.on_receive(e.Buffer, e.Offset, e.BytesTransferred);        ②

        // 다음 메시지 수신을 위해서 다시 ReceiveAsync 메서드를 호출한다.      ③
        bool pending = token.socket.ReceiveAsync(e);
        if (!pending)
        {
            process_receive(e);
        }
    }
    else
    {
        Console.WriteLine(string.Format("error {0},  transferred {1}",
e.SocketError, e.BytesTransferred));
        close_clientsocket(token);
    }
}
```

① 이제 SocketAsyncEventArgs라는 객체가 어느 정도 눈에 익을 겁니다. 이번에도 역시 이 객체를 사용하여 메시지 수신을 처리하게 됩니다. CUserToken 객체를 얻어와서 on_receive 메서드를 호출해 줍니다.

② 이 부분이 정말 중요합니다. 나중에 패킷 처리할 때 더 자세히 알아보겠지만, 지금 보이는 세 개의 파라미터가 패킷 수신 처리의 핵심입니다.

- e.Buffer에는 클라이언트로부터 수신된 데이터들이 들어있습니다.
- e.offset은 버퍼의 포지션을 뜻합니다.
- e.BytesTransferred는 이번에 수신된 바이트 수를 나타냅니다.

지금은 간단히 소개만 하고 자세한 내용은 TCP 패킷 처리 부분에서 코드와 함께 설명하겠습니다. 일단 코드의 흐름이 어떻게 흘러가는가에 집중해 봅시다.

③ 데이터를 한 번 수신한 뒤에는 ReceiveAsync를 다시 호출해줘야 합니다. 그래야 계속해서 데이터를 받을 수 있게 됩니다. 한 번의 ReceiveAsync로 모든 데이터를 다 받지는 못하기 때문입니다. 클라이언트가 잠시 쉬었다 보낼 수도 있고, 한 번에 보낸 것도 네트워크 환경에 따라서 여러 차례에 걸쳐 받을 수 있기 때문이죠. 스트림 기반 프로토콜인 TCP의 특징입니다. 비동기 메서드를 호출한 뒤에는 항상 리턴 값을 확인했습니다. ReceiveAsync 메서드도 똑같습니다. Pending 상태가 아니면 메시지 수신을 처리할 수 있게 직접 process_receive를 호출해줘야 합니다.

2.7 닷넷 네트워크 API

이번 절에서는 사용된 닷넷 네트워크 API에 대해서 간략히 소개하겠습니다. 먼저 비동기 네트워크 메서드에 대한 설명입니다.

public bool AcceptAsync(SocketAsyncEventArgs e) 클라이언트의 연결을 수락합니다.
public bool ReceiveAsync(SocketAsyncEventArgs e) 메시지를 수신합니다.
public bool SendAsync(SocketAsyncEventArgse) 메시지를 전송합니다.
public bool ConnectAsync(SocketAsyncEventArgse) 서버에 접속을 수행합니다.

리턴 값:

비동기 상태로 I/O 작업이 진행되는 경우 true가 리턴됩니다. 비동기 상태란 말은 아직 작업이 끝나지 않았다는 뜻입니다. 이후 I/O 작업이 완료되었을 때 SocketAsyncEventArgs.Completed 이벤트가 발생합니다. 만약 I/O처리가 동기적으로 완료된 경우 false를 반환하며 SocketAsyncEventArgs.Completed 이벤트는 발생하지 않습니다. 이 경우에는 즉시 완료된 것으로 간주하고 처리하면 됩니다.

다음으로 비동기 메서드의 파라미터로 사용되는 SocketAsyncEventArgs 클래스에 대한 설명입니다.

SocketAsyncEventArgs 클래스는 비동기 소켓 작업에 사용되는 클래스입니다. MSDN에는 "Represents an asynchronous socket operation"이라고 나와 있습니다. 비동기 네트워크 메서드를 이용하여 I/O 작업을 수행한 뒤 해당 작업이 완료될 때, 이 클래스 객체의 Completed 이벤트를 통해서 작업 완료 통보가 들어 옵니다. Completed 이벤트에서 호출되는 메서드는 다음과 같은 형태를 갖게 됩니다.

```
$ git checkout linux-next
```

이 메서드의 파라미터로 들어오는 SocketAsyncEventArgs 객체는 비동기 메서드를 호출할 때 파라미터로 넣어줬던 그 객체입니다. 따라서 메서드를 호출할 때마다 따로 보관해놓을 필요 없이 파라미터로 넘어오는 SocketAsyncEventArgs 객체를 그대로 사용하면 됩니다.

지금까지 서버 네트워크 모듈의 기초가 되는 부분을 구현해 봤습니다. C#에서 비동기 소켓 프로그래밍을 어떻게 구현하는지에 대한 부분이 주를 이루었는데 눈에 보이는 부분이 없어서 다소 지루할 수도 있었겠네요. 다음 장에서는 TCP에서 메시지를 처리하는 방법과 함께 패킷 구조를 설계해 서버와 클라이언트 사이에 어떻게 메시지를 교환하는지 알아보겠습니다.