Hanbit eBook

Realtime 80

CGSF 파헤쳐 보기

C++로 온라인 게임 서버 구축하기

박주항 지음

한빛미디어 Hanbit Media, Inc.

CGSF 파헤쳐 보기

C++로 온라인 게임 서버 구축하기

CGSF 파헤쳐 보기 C++로 온라인 게임 서버 구축하기

초판발행 2014년 8월 25일

지은이 박주항 / **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 978-89-6848-676-0 15000 / **정가** 11,000원

책임편집 김창수 / **기획·편집** 정지연
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 김옥현

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

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_박주항

클라이언트 개발자로 게임 회사에 입사했지만, 회사 사정에 의해 서버 업무를 맡게 되면서 여러 게임의 온라인 플랫폼을 구축하고 제작해온 개발자다. JC엔터테인먼트(現 조이시티)에서 서버 개발자로 근무했으며 라인 주식회사에서 게임 플랫폼 관련 업무를 담당하였다. 어릴 적 도스를 접한 뒤로 마이크로소프트를 줄곧 추종해 왔으며, 가장 존경하는 인물도 마이크로소프트사에서 커널을 개발하는 이름 모를 사람이다. WIN32 플랫폼과 C++에 익숙한 나머지 안드로이드 프로그래밍도 자바로 편하게 개발할 수 있는데 굳이 NDK로 개발하려는 융통성 없는 사람이기도 하다. 현재 프리랜서로 일하고 있다.

저자 서문

『CGSF를 활용한 게임 서버 제작』(한빛미디어, 2014)에 이어 후속편을 출간할 수 있어서 개인적으로 매우 뿌듯하다. 전작이 CGSF의 활용에 중점을 뒀다면 이번 책에서는 CGSF의 내부 구조에 대해 자세히 설명하고 있다. 이 책에서 기술하는 내용에 대해 예전부터 의견 교류를 많이 하고 싶었는데, 게임 회사에 다닐 때에는 그럴 기회가 많지 않았다. 이제 시간이 흘러 오히려 이렇게 책으로 필자의 경험을 독자에게 전달하게 되었다. 책을 보고 잘못된 부분이 있다면 지적해 주시고 부족한 부분에 대해 알려주신다면 내용을 보충하겠다.

게임 서버 제작은 밑바닥부터 차근차근 구조를 구축하는 과정이라고 할 수 있다. CGSF는 게임 서버라는 틀에 한정되지만, 이 CGSF가 어떤 식으로 구축되었는지에 대한 세부 사항을 파악하면 다른 프레임워크를 개발할 때 도움이 될 것이라 생각한다. 또한, 시대가 급변하고 습득해야 할 IT 기술이 쏟아지고 있는 상황에서 모든 기능을 일일이 개발하기보다 이미 검증된 오픈 소스를 활용하는 것이 현명한 행동이라고 판단한다. CGSF에서 활용한 수많은 오픈 소스와 클래스는 다른 프로젝트에서도 쉽게 활용할 수 있으므로 살펴본다면 큰 도움이 될 것이다.

CGSF는 현재도 계속 개발하는 중이며 CGSF를 활용한 프로젝트가 활성화된다면 미래에도 CGSF 개발은 지속할 것이다. 미래를 예측할 수는 없지만, 이 CGSF 모듈을 활용하는 미래의 그 누군가에게 감사의 마음을 전한다.

글쓴이 박주항

이 책을 읽기 전에

온라인 게임 서버 프레임워크의 필요성

다양한 장르의 온라인 게임이 출시되고 있고 같은 장르라도 그 게임만의 색깔이 나타나긴 하지만, 게임 접속 후 플레이하기까지의 일련의 과정은 모든 게임이 유사하다는 걸 알 수 있습니다. 그 이유는 온라인 게임에서 게임 서버와의 연동은 필수고 게임 서버가 구축한 시스템을 거쳐야 하기 때문입니다.

특히 우리가 캐주얼 게임이라고 부르는 FPS 게임이나 스포츠 게임 등은 개발하는 과정이 매우 유사합니다. 따라서 서버를 구현할 때 공통적인 부분을 미리 모듈화해 둔다면 새 프로젝트를 진행할 때 게임 개발 일정을 대폭 줄일 수 있습니다. 하지만 현실은 그렇지 않습니다. 새 프로젝트를 진행하면 서버도 새로 만들고 같은 회사 내 다른 부서에서도 기술 공유를 하지 않아서 매번 반복적인 작업을 수행하는 경우가 대부분입니다.

이 책에서는 게임 서버, 특히 캐주얼 게임 서버에서 나타나는 공통적인 부분을 모듈화하여 새 프로젝트를 진행할 때 빠르게 개발하는 것에 역점을 두었습니다. 또한, 온라인 게임 서버 프로그래밍에 필요한 기본적인 개념을 익히고 여러 라이브러리의 사용법을 배우며 간단한 게임을 제작하여 온라인 게임 서버 시스템을 구축할 수 있도록 합니다. 이 책에서 소개할 CGSF[Casual Game Server Framework]는 오픈 소스를 최대한 활용한 서버 라이브러리입니다. 최종 사용자[End-user]가 사용하기 쉽게 설계하는 것을 목표로 하며 아울러 온라인 게임 서버의 구조를 파악하기 쉽게 만들어서 입맛에 맞게 커스터마이징할 수 있도록 구현하는 것을 최종 목표로 합니다.

컴퓨터 과학 VS 소프트웨어 공학

게임 프로그래머는 컴퓨터학과 또는 전산학과 출신이 많습니다. 그런데 이 프로그래머들이 공통으로 경험하는 난감한 문제가 있습니다. 그것은 게임 프로그래밍이 대부분 컴퓨터 과학의 영역이 아니라 소프트웨어 공학의 영역이라는 것입니다. 예를 들어, 어떤 프로그램에서 Quick Sort가 필요할 때 컴퓨터 과학에서는 Quick Sort를 직접 구현하지만, 소프트웨어 공학에서는 이미 구현된 Quick Sort를 사용하려고 합니다. 또 다른 예로 사운드 관련 라이브러리가 필요하면 컴퓨터 과학에서는 사운드 라이브러리를 직접 개발해서 MP3의 헤더 정보를 파싱하는 등의 처리를 직접 구현하지만, 소프트웨어 공학에서는 그냥 기존의 사운드 라이브러리(Miles Sound System, Bass, 또는 FMOD)를 가져다 씁니다.

이렇게 두 영역은 극명하게 차이가 납니다. 그렇다면 소프트웨어 공학에 가까운 게임 프로그래밍의 이점에는 어떤 것이 존재할까요? 제 생각으로는 창의력의 발현, 전체적인 그림을 그릴 수 있는 능력의 획득이라고 봅니다.

자기 생각을 프로그래밍으로 구현할 수 있다는 것은 정말 매력적인 부분이라고 생각합니다. 그래서 과거에는 없던 정말로 옛날 사람들이 보면 이상하게 생각할 기현상, 즉 온종일 컴퓨터 모니터에서 눈을 떼지 않는 패턴에 수긍하고 몰입할 수 있는 것입니다. 또한, 전체적인 그림을 그리는 것, 게임을 만들기 위해 렌더링 엔진의 준비, 사운드 라이브러리의 선택, 재질 효과를 위한 쉐이더 코드 작성, 길 찾기 알고리즘, 물리 엔진의 적용, 해킹을 막기 위한 보안 라이브러리의 적용 등 딴 분야에서는 절대 맛볼 수 없는 여러 영역을 게임 프로그래머는 맛볼 수 있습니다. 물론 특정 영역의 깊이는 얕을 수 있겠지만, 아니 그보다 그 깊이를 파야 할 필요성이 굳이 느

껴지진 않겠지만, 프로그래밍 자체를 즐기는 사람이라면 호기심에서 그 깊이를 파헤쳐 볼 것이며 이를 통해 게임 프로그래밍을 하면서 느꼈던 아쉬운 부분을 해소할 수 있을 것입니다.

이 책에서는 온라인 게임 시스템 구축을 소프트웨어 공학 측면에서 설명하려고 합니다. 어떻게 보면 앞에서 언급한 바와 같이 그 깊이는 얕을지도 모르겠습니다. 그러나 전체를 보는 능력에는 이 책이 큰 도움이 된다고 생각합니다. 빌딩을 건설하기 위해 자재를 준비하는 것과 고층을 어떻게 완성해 나갈지 생각하는 것은 완전히 다른 영역입니다. 필자는 후자의 입장에서 책을 서술하고 있으며 이 책이 온라인 게임 서버의 전반적인 이해에 대해 큰 도움이 될 수 있을 거라 확신하는 바입니다.

또한, 책의 내용의 깊이가 얕을지도 모른다고 언급했지만, 서버 엔진의 전반적인 내용을 대부분 설명하고 있으므로 어떻게 보면 깊이가 얕은 것도 아닙니다. 특히 서버 프로그래밍은 시스템 프로그래밍에도 익숙해야 하므로 컴퓨터 과학 측면과 소프트웨어 공학 측면을 모두 다뤄야 하는 영역이라고 할 수 있습니다. 이런 매력적인 요소를 지닌 게임 서버 프로그래밍에 도전을 진정으로 환영하는 바입니다.

오픈 소스의 매력

앞에서 언급하였지만, 게임을 개발할 때 클라이언트이든 서버이든 간에 소프트웨어 공학 측면에서 접근할 필요가 있습니다. 로그를 남기기 위해 로그 라이브러리를 개발하기보다 기존의 로그 라이브러리를 활용하는 것이 현명한 행동입니다. 또한, 프로세스 덤프를 남기기 위해 직접 덤프 모듈을 개발하기보다 이전에 검증받은 덤프 모듈을 활용하는 것이 생산성을 더 높일 수 있겠죠. 게다가 C++는 아직 주요한

언어이긴 하지만, 자바 등의 다른 프로그래밍 언어와 비교하면 생산성이 떨어지는 것은 부인할 수 없는 사실입니다(물론 그만큼 자유도는 매우 높습니다).

이런 상황에서 오픈 소스를 활용하는 것은 매우 생산적인 행위이며 원하는 제품의 개발 시간을 획기적으로 당겨줄 수 있습니다. 물론 그 안정성을 검증받지 못한 오픈 소스가 부지기수입니다. 이런 경우에는 수많은 테스트와 디버깅을 통해 그 신뢰성을 확보하면서 프로젝트에 적용하면 크게 문제가 되지 않습니다. 또한, 부스트 Boost나 ACE 라이브러리 같은 이미 수많은 고급 프로그래머에 의해 검증된 오픈 소스도 굉장히 많이 존재합니다. 이런 검증받은 오픈 소스를 사용하면 프로그램의 잠재적인 버그에 대한 리스크를 크게 줄일 수 있습니다.

CGSF는 이미 검증된 수많은 오픈 소스 라이브러리를 사용해서 구축하였으며 다음 책에서 이런 좋은 오픈 소스 라이브러리를 소개하겠습니다.

책의 구성

이 책은 두 권으로 구성되어 있습니다. 1권에서는 에코 서버, 채팅 서버 제작을 통해 범용 서버를 제작하는 방법을 다루고, 서버를 제작하는 데 도움을 주는 CGSF 라이브러리의 활용 방법을 소개합니다. 또한, 세븐 게임 프로젝트와 FPS 게임 프로젝트를 통해서 게임 서버를 쉽게 구축하는 방법에 관해서도 다룹니다. 2권에서는 CGSF 라이브러리의 활용 방법을 넘어서 CGSF의 내부 구조와 구현 원리를 설명하고 모바일 클라이언트와의 연동 방법에 대해 간단히 설명합니다.

대상 독자

CGSF는 온라인 게임 서버 시스템을 최대한 빨리 구현해서 프로토타입을 구축하려는 의도로 제작되었습니다. 현업에서 게임 서버를 개발했을 때 구축했던 구조를 담고 있기 때문에 온라인 게임 제작에 관심이 많거나, 게임 서버 프로그래밍에 관심이 많은 학생에게 좋은 가이드가 될 것입니다. 그리고 서버 개발에 대한 세부 지식 없이 게임 서버를 구축하려는 분들께도 하나의 대안이 될 수 있습니다. 아울러 서버 엔진을 직접 개발해 보려는 분들께도 이 책은 좋은 안내서가 될 것입니다. 현업 개발자에게도 온라인 게임 서버의 리뷰 차원에서 이 책은 가치가 있을 것입니다. 또한, CGSF는 다양한 오픈 소스를 활용하고 있고, 조금만 커스터마이징한다면 일반적인 서버로도 활용할 수 있으므로 네트워크에 관심 있는 분이라면 모두 대상 독자에 속한다고 볼 수 있습니다.

용어 표기 기준

이 책의 용어들은 표준국어대사전[01]의 한글 맞춤법과 표준어 규정, 외래어 표기법을 따릅니다. 또한, 표준국어대사전 규정에 없는 용어들은 한국정보통신기술협회 정보통신용어사전[02]을 참고하였습니다.

예) Thread → 스레드
Release → 릴리스

---

01 http://stdweb2.korean.go.kr/main.jsp
02 http://word.tta.or.kr/terms/terms.jsp

사전 지식 및 준비사항

CGSF는 윈도우 기반 라이브러리입니다. 크로스 플랫폼 지원은 현재 고려하고 있지 않습니다. 서버는 클라이언트보다 크로스 플랫폼을 크게 고려할 필요가 없으며 클라우드 서버는 윈도우 기반으로 많이 지원하므로 활용하는 데 별다른 문제가 없을 것입니다. 다만, 클라이언트는 모바일 환경에도 대응해야 한다고 생각하고 있기 때문에 나중에 모바일용 라이브러리를 제작할 계획입니다.

CGSF를 활용해서 게임 서버 및 응용 서버를 제작하려면 다음과 같은 도구와 지식이 필요합니다.

Visual Studio 2013

CGSF는 Visual Studio 2013으로 제작되었습니다. 따라서 윈도우 환경에서만 개발할 수 있으며 윈도우 운영체제에서만 서버 운영이 가능합니다. Visual Studio 2013은 무료 버전인 Express 버전이 존재하므로 Express 버전을 설치하면 CGSF의 빌드가 가능합니다. Visual Studio 2013 Express 버전은 다음 사이트에서 내려받을 수 있습니다.

http://www.microsoft.com/ko-kr/download/details.aspx?id=40787

링크를 따라가면 DVD 이미지를 받아서 설치할 것인지 웹에서 내려받아 설치할 것인지를 묻는 화면이 나옵니다. 편의성을 생각해서 wdexpress_full.exe 파일을 내려받아서 설치합니다.

C++

CGSF는 전체를 C++로 구현하였습니다. 따라서 C++의 기본적인 문법은 숙지해

야 합니다. 해당 모듈을 활용만 한다면 C++의 기본적인 지식으로도 충분히 활용할 수 있지만, CGSF의 엔진 구조를 이해하기 위해서는 템플릿이나 추상 인터페이스 등의 개념을 제대로 이해하고 있어야 합니다. 또한, C++는 객체 지향 언어이므로 객체 지향에 대한 개념을 또한 알고 있어야 합니다.

디자인 패턴

CGSF에서는 단순하지만, 여러 가지 패턴을 활용하고 있습니다. 디자인 패턴 관련 서적을 살펴본 분이라면 무리 없이 책의 내용을 소화할 수 있을 것으로 생각합니다. 디자인 패턴에 대해 잘 모른다 하더라도 내용이 그렇게 어렵지 않으므로 모르는 패턴이 나올 때마다 웹에서 관련 자료를 참조한다면 이해하는 데 크게 무리가 없습니다.

소스 코드

다음 링크에 접속하면 소스 코드를 내려받을 수 있습니다.

https://github.com/pdpdds/CGSF

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 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 힘든 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

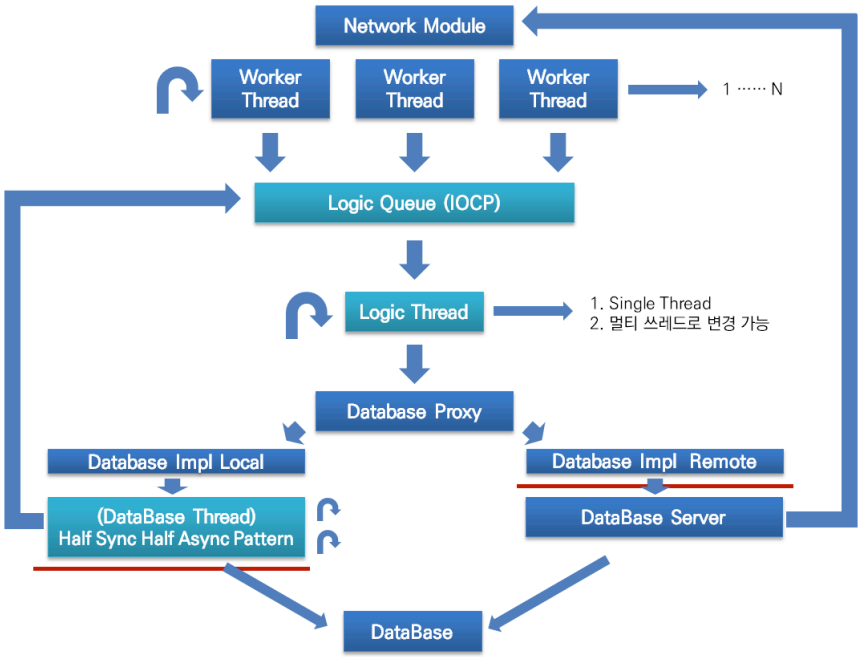
1 | CGSF Internal

『CGSF를 활용한 게임 서버 제작』(한빛미디어, 2014)[01]에서는 CGSF[Casual Game Sever Framework] 모듈의 활용에 중점을 두고 설명했다. 1권을 읽었다면 CGSF에 대해 어느 정도 감을 잡았을 것이다. 2권에서는 CGSF의 내부를 더 자세하게 파헤쳐서 CGSF에 대한 이해도를 높이고 아울러 자신만의 서버를 제작할 수 있도록 CGSF를 커스터마이징 방법에 대해 설명하겠다.

CGSF의 내부 구조는 상용으로 사용된 온라인 게임 서버의 구조를 토대로 구축되었다. KGC 2012에서 CGSF를 주제로 강연했을 때 현업 개발자분들이 자신들의 게임 서버와 CGSF의 구조가 유사하다고 언급했었다. 즉, CGSF의 내부 구조는 어느 정도 검증된 모델이다. 물론 CGSF는 계속 수정되어 구조적으로 큰 변화가 있었고 여러 서버의 장점을 흡수해 왔다. CGSF를 활용한 프로젝트가 활성화된다면 CGSF는 더욱더 유용하고 강력한 서버 모듈로 변모할 수 있을 것이다.

01 http://www.hanbit.co.kr/ebook/look.html?isbn=9788968486708

[그림 1-1] CGSF 내부 구조도

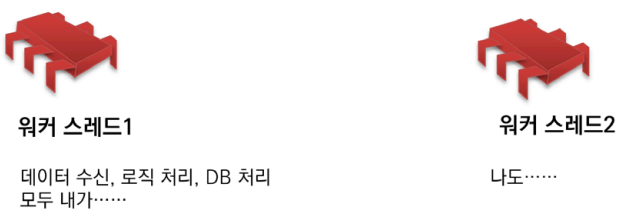


[그림 1-1]은 CGSF의 전체 내부 구조도다. 크게 네트워크 레이어, 엔진 레이어, 로직 레이어, 데이터베이스 레이어로 나눌 수 있고, P2P 레이어는 로직 레이어가 이용하는 구조다. 사실 CGSF는 여러 가지 스레드 모델 및 구조적인 특징이 추가된 상태라 이 구조도가 CGSF 전체의 구조를 모두 표현하지는 못하고 있다. 엄밀히 말하자면 캐주얼 온라인 게임 서버에 특화된 구조도다. 즉, CGSF의 초기 서버 모델이라 할 수 있으며 현재 CGSF는 범용 서버 제작을 위해 구조가 계속 변경되고 있다. 이렇게 변경되는 내용은 소스 코드 분석을 통해 파악할 수 있으리라 생각하고 현시점에서는 CGSF의 캐주얼 온라인 게임 서버 구조에 집중하도록 하겠다.

1.1 네트워크 레이어

네트워크 모듈Network Module 박스와 워커 스레드Worker Thread 부분이 네트워크 레이어에 해당한다. 해당 레이어는 TCP 입출력을 처리하고 유저로부터 받은 데이터를 엔진 레이어에 넘기는 역할을 담당한다. 게임 서버의 초기 모델은 이 네트워크 레이어 한 부분으로 충분했다. 워커 스레드가 네트워크 이벤트를 대기하고 있다가 이벤트가 발생하면 해당 이벤트를 처리하고 그 결과를 클라이언트에 전송하는 역할을 담당했으며 데이터베이스 호출 처리도 전담했었다.

[그림 1-2] 초기 게임 서버 모델

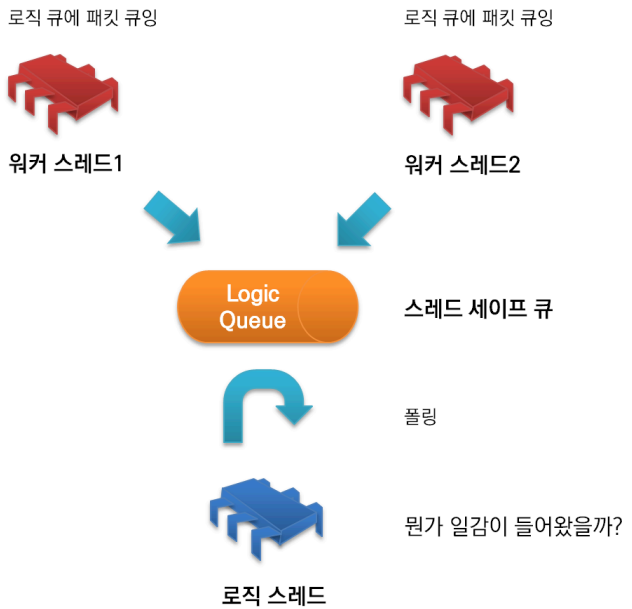


하지만 구조도를 보면 알 수 있듯이 워커 스레드는 하나가 아니라 다수로 운용되므로 공유자원에 접근하기 위해서 동시성 관련 코드가 필수로 들어가게 되었다. 이에 따라 코드가 지저분해지고 개발자의 실수에 따른 멀티스레드 버그가 발생할 확률이 높아진다.

[그림 1-3] 스레드 간 공유자원 획득 경합

그래서 게임 서버, 특히 캐주얼 게임 서버는 성능이 약간 떨어지더라도 안전성을 높이고 게임 로직을 쉽게 구현하는 방향으로 선회했다. CGSF에서는 이 워커 스레드가 로직 처리를 하지 않고 로직 레이어에 그 역할을 일임한다. 네트워크 레이어가 넘기는 패킷은 로직 레이어에 넘어가기 전에 엔진 레이어를 거치게 된다.

[그림 1-4] CGSF의 데이터 처리 구조



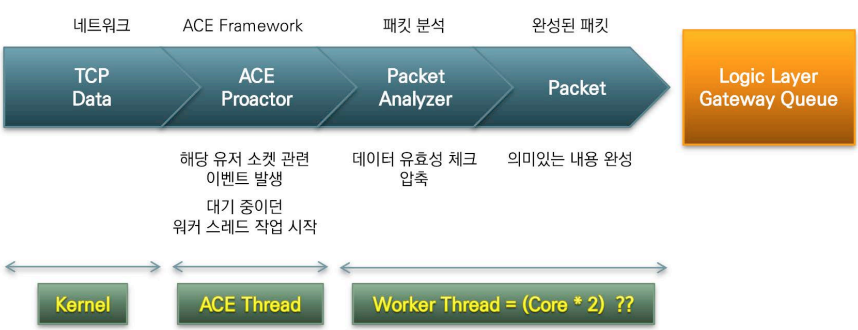
CGSF의 기본 네트워크 모듈은 ACE로 제작되었으며 다른 모듈로 교체하기 쉽게 DLL 형태로 제작되었다(Boost.Asio 모듈로도 쉽게 교체할 수 있다). 여러 가지 상황에 따라 ACE 네트워크를 사용하지 않고 자신이 구현한 서버 라이브러리를 적용할 필요가 생길 수도 있는데, CGSF에서는 직접 제작한 서버 라이브러리를 쉽게 적용할 수 있도록 인터페이스를 제공한다. 또한, 수많은 C++ 서버 라이브러리를 리서치하고 있으며 이 모듈들을 전부 CGSF에 반영할 예정이다. 하지만 엔진 레이어에

서 네트워크 모듈을 호출하는 인터페이스가 아직 안정화되지 않아서 계속 변경되고 있는 만큼, 현시점에서 새로운 네트워크 모듈을 추가하면 추후 인터페이스를 변경함에 따라 수정작업이 대폭 증가하게 될 것이다(네트워크 모듈 인터페이스를 변경하면서 ACE 모듈과 Boost.Asio 모듈의 내부 코드를 둘 다 수정했는데, 이 작업이 만만치 않았다). 향후 네트워크 모듈의 인터페이스가 더는 변경되지 않는다고 판단되면 리서치한 네트워크 모듈들을 CGSF에 추가할 것이다. 이 책이 출간된 시점에서 2년 후면 수많은 네트워크 모듈이 CGSF 프로젝트에 포함되어 있을 지도 모를 일이다. 물론 이를 위한 전제조건인 CGSF를 활용한 프로젝트가 활성화되었을 경우에만 가능한 일이다.

1.2 엔진 레이어

엔진 레이어는 [그림 1-1]에서 로직 큐Logic Queue 박스와 그 직전 부분에 해당한다. 네트워크 레이어에서 넘어온 데이터를 로직 큐에서 큐잉하기 위해 엔진 레이어에서는 설정된 패킷 프로토콜을 이용해서 로직 레이어에서 인식할 수 있는 패킷으로 변경한 후 로직 큐에 큐잉한다. 내부적으로 더 많은 작업을 수행하는데, 자세한 구조에 대해서는 3장 엔진 레이어에서 살펴보겠다.

[그림 1-5] 유저 데이터 처리 흐름



[그림 1-5]에서 네트워크과 ACE Framework 부분이 네트워크 레이어에 해당한

다. 네트워크 레이어를 거쳐 넘어온 데이터는 엔진 레이어의 패킷 분석기를 통해 패킷으로 생성되며 완성된 패킷은 로직 큐로 큐잉된다.

1.3 로직 레이어

로직 레이어는 [그림 1-1]에서 로직 스레드Logic Thread 박스에 해당한다. 로직 레이어는 이름 그대로 게임의 로직을 처리하는 계층으로, 엔진 레이어가 패킷을 로직 큐에 집어넣으면 로직 스레드는 이를 감지하고 패킷을 꺼내서 작업을 수행한다. 작업 수행을 완료하면 로직 큐에 큐잉된 패킷이 있는지 확인하고 없으면 이벤트가 발생할 때까지 대기한다.

로직 스레드는 싱글 스레드로 운용되므로 고속으로 코드가 실행되어야 한다. 따라서 로직 스레드의 처리 코드에 Sleep 함수나 블로킹 관련 코드가 들어가서는 절대 안 된다. 로직을 처리하는 스레드가 하나라서 성능상 불이익이 있지만, 네트워크나 데이터베이스 처리를 비동기로 처리해서 성능 향상을 도모하고 있고 단일 스레드로 로직을 처리하므로 콘텐츠 개발자 입장에서는 클라이언트 개발하듯이 코드를 구현할 수 있다. CGSF는 개발 편의와 성능의 트레이드오프(절충)를 어느 정도 고려해서 제작되었다. 또한, CGSF는 싱글 로직 스레드를 멀티 로직 스레드로 쉽게 변경할 수 있다. 단지 이 과정에서 동기화 관련 코드가 추가되며 코드 작성이 불편해진다. 하지만 이것도 게임 서버의 성격에 따라 구조화시켜 놓으면 콘텐츠 구현부는 편하게 작성할 수 있다.

한 가지 짚고 넘어갈 부분이 있는데, 지금 설명한 로직 레이어는 CGSF의 캐주얼 게임 프레임워크에 해당한다는 점이다. 하지만 CGSF의 구조를 개선하면서 CGSF의 최신 코드에는 캐주얼 게임 프레임워크뿐만 아니라 게임 서버 초기 모델 및 MO/MMO 프레임워크를 어느 정도 구현하였다. 일단 설명이 복잡해지니 로직 레이어는 캐주얼 게임 프레임워크를 지칭하는 걸로 하겠다.

1.4 데이터베이스 레이어

앞에서 언급했지만, 로직 레이어의 로직 스레드에는 블로킹 코드가 존재하면 안 된다. 일반 서버의 데이터베이스의 호출은 대부분 동기 호출로 구현되는데, CGSF에서는 데이터베이스 호출이 비동기로 구현될 필요가 있다. 이를 위해 데이터베이스 레이어는 데이터베이스 호출의 비동기 처리가 가능하도록 구현되었으며, 프로세스 내부에 별도의 스레드를 생성하거나 원격지의 데이터베이스 서버에 데이터베이스 호출을 위임하는 형태로 설계되었다(현재 원격지 데이터베이스 서버를 호출하는 기능은 문제를 단순화하기 위해 제외되었다).

데이터베이스 모듈은 다음을 지원한다.

- MySQL
- MSSQL
- FastDB

해당 데이터베이스 모듈 역시 단위 전략 패턴을 사용해서 쉽게 교체할 수 있는데, 캐주얼 게임 프레임워크에서는 MySQL을 사용했다. MSSQL이나 MySQL의 설치가 번거롭다면 게임 서버 초기 개발 때에는 sqlite나 메모리 데이터베이스인 FastDB 모듈을 사용해서 개발하는 것도 좋은 선택이 될 수 있다. 데이버베이스 쿼리 호출 후 처리된 결과를 로직 큐에 큐잉하면 로직 스레드가 그 결과를 받아서 처리할 수 있다.

1.5 P2P 레이어

P2P 레이어는 빠른 응답을 요구하는 FPS 게임이나 스포츠 게임을 지원하기 위한 계층으로, 로직 레이어에서 이 P2P 레이어를 이용한다. 게임 방에 유저가 입장하면 P2P 기능이 활성화되어 유저 간 데이터를 직접 주고받는 로직이 실행된다.FPS 프로젝트에서 캐릭터의 움직임과 바라보는 방향에 대한 처리는 이 P2P 레이어를 사용하였다.

1.6 정리

1장에서는 CGSF의 레이아웃을 간략히 살펴보았다. 아키텍처 구조도에서는 드러나지 않지만 각종 유틸리티 클래스의 집합인 베이스 레이어를 포함해서 CGSF는 총 6개의 레이어로 구성되어 있다. 이 레이어를 다시 정리하면 다음과 같다.

[표 1-1] CGSF를 구성하는 레이어

CGSF 레이어	역할
네트워크 레이어	네트워크 이벤트를 처리한다. 다양한 네트워크 모듈을 적용할 수 있다.
엔진 레이어	서버의 동작을 제어한다. CGSF의 핵심 뼈대다.
로직 레이어	캐주얼 게임, MO/MMO, 또는 기타 비즈니스 로직을 추가할 수 있는 계층이다. 프로젝트 성격에 따라 변동성이 크다(소스 코드 수정이 빈번하다).
데이터베이스 레이어	데이터베이스와 연동을 구현한 계층이다. 현재 MSSQL, MySQL과 연동하기 위한 기능이 구현되어 있으며 다른 데이터베이스 플랫폼과의 연동 기능도 추가할 예정이다.
P2P 레이어	응답성이 높은 FPS 게임 등에서 요구되는 계층이다. 현재는 OCF P2P 모듈만 있지만, 다른 P2P 모듈도 추가할 예정이다.
베이스 레이어	CGSF의 개발 편의성을 위한 유틸리티 클래스 모음 계층이다.

2장부터는 각 레이어에 대해 자세히 설명하겠다.

2 | 네트워크 레이어

네트워크 레이어는 TCP/IP 처리를 담당하는 계층으로, 소켓 및 TCP 처리를 담당한다. 이번 장에서는 CGSF의 기본 네트워크 엔진인 CGSFNet(ACE 네트워크)에 대해 알아보고 CGSFNet과 엔진 레이어의 연동을 설명한다. 아울러 네트워크 엔진의 커스터마이징 방법도 살펴본다.[01]

2.1 ACE Network

컴퓨터 분야는 여러 부문에서 표준화가 이루어졌다. C++도 최근에 C++11이 나오는 등 점차 표준화되고 추상화되고 있다. 네트워크 소프트웨어 분야에서도 이미 이런 추상화 흐름이 나타나기 시작하여 수많은 미들웨어가 나왔는데, 이중 대표적인 미들웨어가 ACE[ADAPTIVE Communication Environment][02]다.

ACE는 네트워크 애플리케이션의 동시 접근과 I/O를 효율적이고 유연하게 제어하기 위해 많은 핵심 패턴을 제공하는 오픈 소스 기반 크로스 플랫폼이자 객체지향 C++ 프레임워크다. 이 ACE 프레임워크만 제대로 활용할 줄 안다면 네트워크 프로그래밍은 완전히 정복한 것이나 마찬가지라고 감히 말하고 싶다. 또한, 특정 개념에 대한 추상화가 잘 되어 있어서 ACE 프레임워크를 살펴보는 것은 차후 자신만의 프로젝트를 진행할 때 인터페이스를 구축하는 데 큰 도움이 될 것이다.

개인적으로 이런 미들웨어를 쓰는 것이 Raw Socket 코딩을 직접 하는 것보다 더 의미있는 작업이라고 생각한다. 기존 고품질의 네트워크 라이브러리는 이미 검증되었으므로 의심의 여지 없이 사용할 수 있고, 이에 따라 개발자의 실수가 적어지기 때문이다.

---

01 네트워크 레이어는 NetworkLayer 폴더에서 확인할 수 있다.

02 http://www.cs.wustl.edu/~schmidt/ACE.html

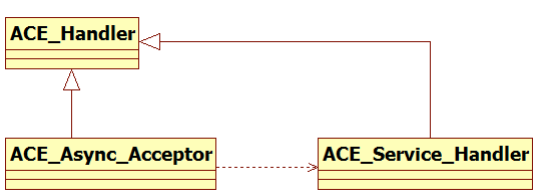
예전에 진행했던 프로젝트에서는 소켓 처리를 래핑한 클래스를 직접 상속하여 세션 처리를 했는데, 이런 구조 때문에 소켓 관련 처리를 직접 수정하다가 프로그램에 버그가 나는 경우가 많았다. 따라서 새 프로젝트를 시작할 때마다 저수준 소켓 API를 다루는 것보다 신뢰할 수 있는 네트워크 미들웨어를 사용하는 것이 더 좋다고 생각한다. 하지만 이것이 미들웨어가 더 다루기 쉽다는 것을 의미하는 것은 아니다. 미들웨어를 제대로 이해하고 정확히 활용하기 위해서는 프로그래밍 기초가 어느 정도 뒷받침되어야 한다.

ACE는 이벤트의 다중 수신이 가능하고(소켓 이벤트, 타이머 이벤트) 스레드, 프로세스 간 통신 방법이 정교하게 정의되어 있어, 비동기로 다른 스레드나 프로세스에 작업을 넘기는 것과 동시 접근에 대한 전략을 변경하기(잠금 유무)가 쉽고 동기화를 위한 편리한 인터페이스를 제공한다. CGSF는 Proactor 패턴[03]을 사용한 ACE의 비동기 서버 기능을 활용하는데 여기서 간략히 그 구조를 살펴보자.[04]

2.1.1 Acceptor & Service Handler

서버에는 유저의 접속 요청을 받아들이기 위한 Acceptor가 존재한다. Acceptor는 접속 요청을 한 유저를 위한 서비스 핸들러Service Handler를 생성하고 해당 유저에 대해서 더는 관여하지 않는다. 접속 요청을 한 유저는 이후 Acceptor에서 생성한 서비스 핸들러를 통해서 서버와 교신한다.

[그림 2-1] Acceptor & Service Handler 클래스 다이어그램

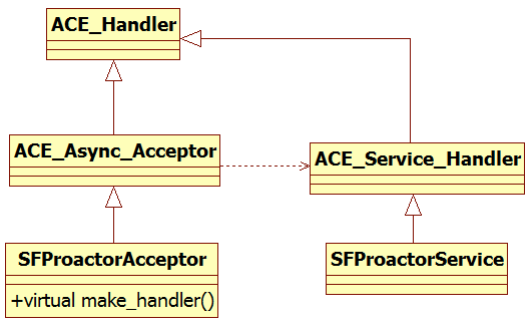


03 http://en.wikipedia.org/wiki/Proactor_pattern
04 이 부분은 CGSFNet 프로젝트를 참조하면서 보기 바란다

즉, ACE_Service_Handler가 제대로 생성되면 ACE_Async_Acceptor와 ACE_Service_Handler 클래스 간 상호 의존성은 없어진다.

이제 [그림 2-2]의 관계를 기초로 응용 개발자가 생성해야 할 클래스들을 살펴보겠다.

[그림 2-2] 확장된 Acceptor & Service Handler 클래스 다이어그램



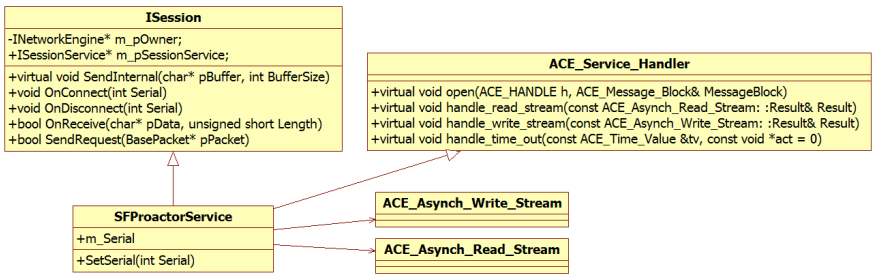
[그림 2-2]는 개발자가 확장해서 구현해야 할 부분까지 포함한 클래스 다이어그램이다. CGSFNet에서는 SFProactorAcceptor 클래스와 SFProactorService 클래스를 각각 추가하여 목적에 맞는 Acceptor와 서비스 핸들러가 되도록 하였다.

SFProactorAcceptor에서는 ACE_Async_Acceptor의 가상 메소드를 오버라이드하여 재정의할 수 있는데, 그중에서 필수로 구현해야 하는 메소드는 make_handler다. ACE Proactor에서 클라이언트 접속 이벤트를 받으면 SFProactorAcceptor의 make_handler 메소드를 호출하고, 이 메소드에서 SFProactorService 객체를 생성한다. make_handler 메소드는 ACE_Service_Handler 타입의 객체를 반환하고, ACE Proactor에서는 생성된 객체의 open 메소드를 호출하여 클라이언트 연결 처리를 위한 초기화 작업을 수행한다. 여기서 SFProactorService 클래스는 유저 세션이라고 이해하면 된다.

2.1.2 유저 세션 및 데이터 처리

다음 클래스 다이어그램을 살펴보자.

[그림 2-3] SFProactorService 클래스 다이어그램



앞에서 SFProactorService 클래스는 Acceptor가 생성한 서비스 핸들러라고 언급한 바 있다. 이후 해당 유저는 이 객체를 통해서 서버와 통신하는데, 이때 SFProactorService 클래스는 단순하게 소켓이라고 생각해도 무방하다. SFProactorService 클래스는 ACE_Service_Handler 클래스를 상속했으며 클라이언트와 데이터를 주고받는 데 사용된다. SFProactorService 클래스는 세 가지 메소드를 오버라이드해서 구현해야 한다. 다음은 세 가지 메소드의 원형이다.

[코드 2-1] SFProactorService의 오버라이드 메소드

```
virtual void handle_read_stream(const ACE_Asynch_Read_Stream::Result& Result)
override;
virtual void handle_write_stream(const ACE_Asynch_Write_Stream::Result&
Result) override;
virtual void handle_time_out (const ACE_Time_Value &tv, const void *act = 0)
override;
```

SFProactorService 클래스는 ACE_Async_Read_Stream, ACE_Async_

Write_Stream이라는 두 개의 클래스를 참조하는데, 각각의 클래스는 비동기 읽기와 쓰기 스트림 처리를 담당한다.

[코드 2-2] ACE 비동기 입출력 처리를 위한 클래스

```
ACE_Asynch_Read_Stream, ACE_Asynch_Write_Stream
```

읽기와 쓰기 처리에 대한 결과를 기다려야 한다면 해당 스레드의 흐름이 멈추게 된다. 이는 성능상 좋지 않으므로 비동기로 처리하는 것은 지극히 당연하다.

ACE_Async_Read_Stream 클래스는 비동기로 읽기 작업을 요청하는 역할을 하며, 그 결과는 SFProactorService의 handle_read_stream( ) 메소드에서 처리된다. 마찬가지로 ACE_Async_Write_Stream 클래스는 유저에게 데이터를 전송하기 위해 쓰기 요청을 담당하는 클래스로, 그 결과는 SFProactorService의 handle_write_stream( ) 메소드에서 처리된다.

SFProactorService 클래스는 ACE_Service_Handler 외에 ISession 클래스를 상속하는데, 이 ISession은 CGSF의 엔진 레이어와 통신하기 위해 제공되는 인터페이스다. SFProactorService 클래스에서는 유저 소켓과 마찬가지로 상황에 따라 다음과 같은 세 가지 이벤트가 발생한다.

- 유저 접속
- 유저 접속 종료
- 유저 데이터 전송

각 상황에 따라 엔진 레이어가 인지할 수 있도록 ISession 인터페이스의 OnConnect, OnDisconect, OnReceive 메소드를 호출해야 한다.

[코드 2-3] SFProactorService 클래스의 open 메소드

```
void ProactorService::open( ACE_HANDLE h, ACE_Message_Block& MessageBlock )
{
      this->handle(h);

      //비동기 읽기/쓰기 객체 초기화, 실패하면 유저 세션을 종료한다.
      if (this->m_AsyncReader.open(*this) != 0 || this->m_AsyncWriter.open(*this)
!= 0)
      {
         delete this;
         return;
      }

      m_pTimerLock = new InterlockedValue();
      m_pTimerLock->Release();

      m_bServiceCloseFlag = false;

      m_Serial = ProactorServiceMapSingleton::instance()->Register(this);

      assert(m_Serial != INVALID_ID);

      RegisterTimer();

      ISession::OnConnect(this->m_Serial); //엔진 레이어로 유저 접속을 알림

      PostRecv(); //비동기 읽기 작업 호출
}
```

유저가 처음 접속하면 ProactorService의 open 메소드가 호출되고, 이 메소드는 다음과 같은 흐름으로 진행된다.

- ACE_Async_Read_Stream 객체와 ACE_Async_Write_Stream 객체를 초기화한다.
- 유저 리스트에 해당 유저를 등록한다.
- 엔진 레이어에서 유저가 접속했음을 인지할 수 있도록 ISession 인터페이스의 OnConnect 메소드를 호출한다.
- 비동기 읽기 작업을 요청한다.

[코드 2-4] 비동기 읽기 작업을 호출하는 PostRecv 메소드

```
void ProactorService::PostRecv()
{
      ACE_Message_Block* pBlock;

      ACE_NEW_NORETURN(pBlock, ACE_Message_Block (2048));
      //비동기 읽기 작업 요청
      if(this->m_AsyncReader.read(*pBlock, pBlock->space()) != 0)
      {
         pBlock->release();
         ReserveClose();
         return;
      }
}
```

PostRecv 메소드를 통해 읽기 작업을 비동기로 요청했으므로 이후 유저가 데이터를 전송하면 handle_read_stream 메소드가 호출된다. 비동기 읽기 작업을 위해 읽을 데이터를 저장할 메모리 공간을 할당하는 데 ACE_NEW_NORETURN 매크로를 사용했다. 이 매크로는 메모리 할당에 실패하면 더는 코드를 수행하지 않고 리턴해 버리므로 주의한다. 메모리 크기를 2048바이트로 할당했지만, 서버 클라이언트 간 오가는 패킷의 크기가 매우 크다고 판단되면 이 값을 늘려준다.

[코드 2-5] handle_read_stream 메소드

```
void ProactorService::handle_read_stream(const
ACE_Asynch_Read_Stream::Result& Result)
{
      ACE_Message_Block& Block = Result.message_block();
      if(!Result.success() || Result.bytes_transferred() == 0)
      {
         Block.release();
         ReserveClose();
      }
      else //정상적으로 데이터를 읽었다면 해당 데이터를 엔진 레이어로 보낸다.
      {
         if(false == ISession::OnReceive(Block.rd_ptr(), Block.length()))
         {
            Block.release();
            ReserveClose();
            return;
         }

         PostRecv();
      }
}
```

handle_read_stream 메소드는 다음과 같은 흐름으로 진행된다.

- 전송된 데이터가 0바이트라면 연결이 끊겼거나 내부적으로 문제가 발생한 상태이므로 접속을 종료한다.
- ISession 인터페이스의 OnReceive 메소드를 호출해서 엔진 레이어에 유저 데이터를 전송한다.
- PostRecv 메소드를 호출해서 다시 비동기 읽기 요청을 한다.

또한, 네트워크 모듈은 엔진 레이어가 접속한 클라이언트에게 패킷을 전송할 수 있도록 SendInternal 메소드를 구현해야 한다.

[코드 2-6] SendInternal 메소드

```
void SFProactorService::SendInternal(char* pBuffer, int BufferSize, int
ownerSerial)
{
	ACE_Message_Block* pBlock = NULL;

	//패킷 전송을 위한 버퍼 할당, 버퍼 할당에 실패하면 해당 메소드는 수행을
멈추고 리턴한다.
	ACE_NEW_NORETURN(pBlock, ACE_Message_Block (BufferSize));

	//버퍼 복사
	pBlock->copy((const char*)pBuffer, BufferSize);

	if(NULL == pBlock->cont())
	{
	   m_AsyncWriter.write(*pBlock, pBlock->length());
	}
	else
	{
	   m_AsyncWriter.writev(*pBlock, pBlock->total_length());
	}
}
```

엔진 레이어에서 패킷 전송을 요청하면 SFProactorService 객체의 SendInternal 메소드가 호출되고, SendInternal 내부에서는 패킷 내용을 저장하기 위해 버퍼를 할당하여 ACE_Async_Write_Stream 객체, 즉 m_AsyncWriter에 비동기 쓰기를 요청한다. 이후 패킷 전송 결과는 handle_write_stream 메소드에서 확인할 수 있다.

[코드 2-7] handle_write_stream 메소드

```
void ProactorService::handle_write_stream( const
ACE_Asynch_Write_Stream::Result& Result )
{
     Result.message_block().release();
}
```

쓰기 이벤트 결과에 대해서는 특별한 처리를 하지 않는다.

2.1.3 Worker Thread

워커 스레드Worker Thread는 네트워크 이벤트를 대기하고 처리하는데, ACE에서는 다음과 같은 코드로 스레드를 생성할 수 있다.

[코드 2-8] 워커 스레드의 생성

```
m_workThreadGroupID =
ACE_Thread_Manager::instance()->spawn_n(OptimalThreadCount,
(ACE_THR_FUNC)ProactorWorkerThread, NULL, THR_NEW_LWP,
ACE_DEFAULT_THREAD_PRIORITY, 1);
```

각각의 인자에 대한 의미는 다음과 같다.

- OptimalThreadCount: 생성할 스레드 수
- ProcatorWorkThread: 스레드 엔트리
- Return Value: 스레드 그룹 번호

이 워커 스레드가 어떻게 작업을 진행하는지 구체적으로 살펴보자.

[코드 2-9] 워커 스레드 메인 함수

```
void ProactorWorkerThread(void* Args)
{
      ACE_Proactor::instance()->proactor_run_event_loop();
}
```

앞의 코드가 워커 스레드와 관련된 코드의 전부로, 워커 스레드는 네트워크 이벤트가 발생할 때까지 무한정 대기한다. IOCP 버전으로 바꾼다면 다음 코드와 대응된다.

[코드 2-10] GetQueuedCompletionStatus 메소드

```
::GetQueuedCompletionStatus(m_hIOCP, &NumberOfBytesTransferred,
&pCompletionKey, &pOverlapped, INFINITE);
```

이렇게 워커 스레드가 네트워크 이벤트를 대기하다가 유저가 데이터를 전송하면 SFProactorService 클래스의 handle_read_stream 메소드가 호출된다. 이때의 콜 스택은 다음과 같다.

[코드 2-11] 유저의 데이터 전송 시 handle_read_stream 메소드 내부의 콜 스택

```
CGSFNet.dll!SFProactorService::handle_read_stream()
ACEd.dll!ACE_WIN32_Asynch_Read_Stream_Result::complete()
ACEd.dll!ACE_WIN32_Proactor::application_specific_code()
ACEd.dll!ACE_WIN32_Proactor::handle_events(unsigned long
milli_seconds=4294967295)
ACEd.dll!ACE_WIN32_Proactor::handle_events()
ACEd.dll!ACE_Proactor::handle_events()
ACEd.dll!ACE_Proactor::proactor_run_event_loop(int (ACE_Proactor *)*
eh=0x00000000)
```

```
NetworkEngine.dll!ProactorWorkThread(void * Args=0x00000000)
ACEd.dll!ACE_Thread_Adapter::invoke_i()
ACEd.dll!ACE_Thread_Adapter::invoke()
ACEd.dll!ace_thread_adapter(void * args=0x0118b558)
```

빨간색으로 강조된 부분은 워커 스레드가 네트워크 이벤트를 대기하고 있을 때 호출한 proactor_run_event_loop 메소드다. 유저로부터 이벤트를 받으면 유저 소켓을 나타내는 SFProactorService 객체의 handle_read_stream 메소드가 호출되고 이를 통해 서버는 해당 유저와 데이터를 주고받을 수 있다. 이런 일련의 작업을 통해 완성된 패킷이 CGSF의 로직 레이어로 넘어간다(완전한 패킷을 생성하는 작업은 엔진 레이어에서 처리된다).

2.1.4 정리

지금까지 네트워크 레이어에서 서버에 접속한 유저를 처리하는 방식에 대해서 간략히 살펴보았다. ACE 라이브러리를 사용하는 CGSFNet 모듈을 예로 들어 설명하였지만, 이 모듈의 네트워크 흐름을 파악한다면 다른 네트워크 모듈에 대한 이해와 적용도 쉬울 것이다. 사실 수많은 C++ 네트워크 모듈을 CGSF에 등록해서 입맛에 따라 다양한 네트워크 DLL 파일을 활용하고 싶었는데 앞에서 언급한 바와 같이 네트워크 모듈의 인터페이스가 완전히 확립되지 않아 작업을 미루고 있다. 인터페이스가 더는 수정되지 않는다고 판단되면 기존 온라인 게임 서버 서적에서 제공되었던 게임 서버 소스나 개인이 제작한 서버 라이브러리를 CGSF에서 활용할 수 있도록 프로젝트에 추가할 것이다.

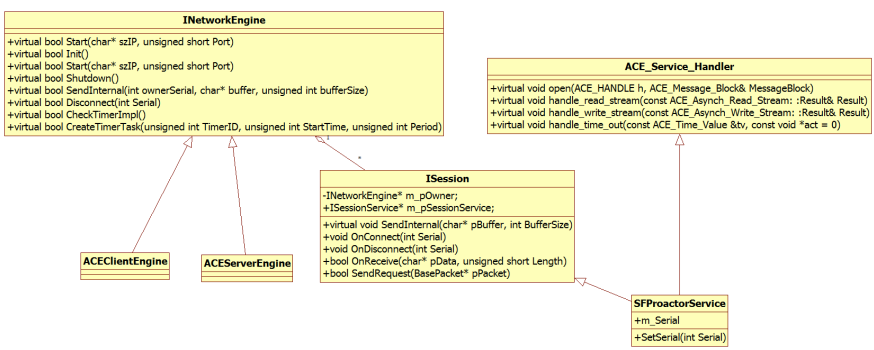
이전에 제작된 훌륭한 C++ 게임 서버 라이브러리가 다수 존재하는데 이대로 사장되는 것은 안타까운 일이라 생각한다. 그래서 네트워크 모듈을 재활용하자라는 취지에서 플러그인 형식의 DLL 형태로 인터페이스를 제공하는 아이디어를 구상했다. 하지만 아직 인터페이스가 계속 변경되고 있는 걸 생각하면 해당 부분은 아직

개선의 여지가 남아있다.

2.2 네트워크 엔진 커스터마이징 이론

CGSF에서 네트워크 모듈은 쉽게 커스터마이징할 수 있다. 즉, 자신이 만든 네트워크 모듈이 있다면 쉽게 CGSF에 적용할 수 있다는 뜻이다. CGSF 엔진에서는 INetworkEngine 인터페이스를 구현한 네트워크 모듈을 DLL 형태로 메모리에 적재해서 정해진 메소드를 호출한다. 그럼 CGSF의 기본 네트워크 모듈인 CGSFNet 프로젝트를 예로 들어 커스터마이징 이론을 설명하겠다.

[그림 2-4] INetworkEngine 클래스 및 관련 클래스 다이어그램



[그림 2-4]는 CGSF의 기본 네트워크 모듈인 CGSFNet의 다이어그램이다. 이 다이어그램에서 확인할 수 있는 것처럼 자신만의 네트워크 모듈을 제작하는 데 필요한 작업은 다음과 같다.

- ACEClientEngine, ACEServerEngine과 같은 INetworkEngine 인터페이스를 구현한 클래스
- SFProactorService와 같은 유저 세션을 나타내는 클래스가 ISession 인터페이스를 상속하게 하고 유저의 접속과 종료, 데이터 전송에 따른 이벤트 발생 시 ISession의 적절한 메소드 호출

2.2.1 INetworkEngine 인터페이스

이 코드는 INetworkEngine 인터페이스를 간략히 정리한 것이다.

[코드 2-12] INetworkEngine 인터페이스

```
class NETWORKENGINEDECL INetworkEngine
{
virtual bool Start(char* szIP, unsigned short Port) = 0; //엔진 시작
virtual bool Shutdown() = 0; //엔진 종료
     virtual bool SendInternal(int ownerSerial, char* buffer, unsigned int
bufferSize) = 0;// 데이터 전송
     virtual bool Disconnect(int Serial) = 0; //강제로 세션을 종료
     virtual bool CheckTimerImpl() {return false;} //타이머 기능 구현 여부,
기본은 구현 안됨
     virtual bool CreateTimerTask(unsigned int TimerID, unsigned int
StartTime, unsigned int Period) {return false;} //타이머 생성
};
```

CGSF엔진에서는 네트워크 모듈을 메모리에 적재한 후 CreateNetworkEngine 메소드를 호출해서 INetworkEngine 정적 객체에 대한 참조를 유지한다. 그리고 상황에 따라(대부분 서버 구동 시에 호출되지만) INetworkEngine 객체의 메소드들을 호출한다.

[코드 2-13] INetworkEngine 객체의 생성

```
INetworkEngine * CreateNetworkEngine(bool Server, IEngine* pEngine)
{
     if(Server)
        return new ACEServerEngine(pEngine);
     else
        return new ACEClientEngine(pEngine);
}
```

CGSF가 네트워크 모듈을 생성하기 위해 호출하는 코드다. 상황에 따라 서버 또는 클라이언트 네트워크 모듈을 생성하고 그 결과를 반환한다. 서버와 클라이언트 모두 살펴봐야 하지만, 흐름이 비슷하므로 서버 쪽 네트워크 모듈만 살펴보겠다.

[코드 2-14] CGSFNet 네트워크 엔진의 시작을 위한 Start 메소드

```
bool ACEServerEngine::Start(char* szIP, unsigned short Port)
{
      SYSTEM_INFO si;
      GetSystemInfo(&si);

      int OptimalThreadCount =si.dwNumberOfProcessors * 2;

      m_workThreadGroupID = ACE_Thread_Manager::instance()-
>spawn_n(OptimalThreadCount, (ACE_THR_FUNC)ProactorWorkerThread, NULL,
THR_NEW_LWP, ACE_DEFAULT_THREAD_PRIORITY, 1);

      if (m_workThreadGroupID == -1)
      {
         return false;
      }

      ACE_INET_Addr listen_addr;

      listen_addr.set(Port);

      if(0 != m_Acceptor.open(listen_addr, 0, 1, ACE_DEFAULT_BACKLOG, 1, 0, 1,
1, 1024))
         return false;

      return true;
}
```

정해진 포트로 서버를 시작하는 코드로, 흐름은 다음과 같다.

- 시스템 정보를 얻어와서 프로세서의 개수를 알아낸다.
- 프로세서 코어의 개수 × 2만큼의 워커 스레드를 생성한다(워커 스레드의 수는 프로세서 코어의 개수 × 2가 최적이라고 알려졌으나 워커 스레드 외에 다양한 목적의 많은 스레드가 존재하므로 테스트를 통해 최적의 개수를 구하는 것이 필요하다). 워커 스레드가 생성되면 이 워커 스레드들은 네트워크 이벤트 처리를 위해 대기한다.
- Acceptor를 오픈해서 유저의 접속을 받아들일 준비를 한다.

m_Acceptor.open() 메소드가 성공하면 유저의 접속을 받아들일 준비가 끝난다. 이제 유저가 접속하면 Acceptor는 해당 유저를 위한 소켓을 생성하고 유저 관련 네트워크 이벤트는 워커 스레드가 처리한다.

[코드 2-15] 유저에게 패킷 전송을 위한 SendInternal 메소드

```
bool ACEServerEngine::SendInternal(int ownerSerial, char* buffer, unsigned int
bufferSize)
{
      ProactorServiceMapSingleton::instance()->SendInternal(ownerSerial,
buffer, bufferSize);

     return true;
}
```

특정 타깃 유저에게 패킷을 전송할 수 있는 SendInternal 메소드다.

[코드 2-16] 타이머 기능 구현

```
bool ACEServerEngine::CheckTimerImpl()
{
     return true;
```

```
}

bool ACEServerEngine::CreateTimerTask(unsigned int TimerID, unsigned int
StartTime, unsigned int Period)
{
      ACE_Time_Value Interval(Period/1000, (Period%1000)*1000);
      ACE_Time_Value Start(StartTime/1000, (StartTime%1000)*1000);

      if (ACE_Proactor::instance ()->schedule_timer (m_TimeOutHandler,
         (void*)TimerID,
         Start,
         Interval) == -1)
         return false;

      return true;
}
```

타이머 기능이 필요 없을 수도 있지만, 반드시 구현해야 할 기능 중 하나다. 예를 들어, 1분 간격으로 현재 접속 유저 수를 체크한다고 가정해 보자. 이 경우 앞의 코드처럼 타이머를 구현해 두면 CGSF에서 이벤트를 받아서 해당 작업을 처리할 수 있다. 네트워크 모듈에서 타이머 기능을 수행할 능력이 있으면 CheckTimerImpl 메소드가 TRUE를 반환하도록 하고, CreateTimerTask 메소드에서 타이머 태스크를 생성할 수 있도록 구현한다. ACE에는 타이머를 스케줄링할 수 있는 기능이 있다.

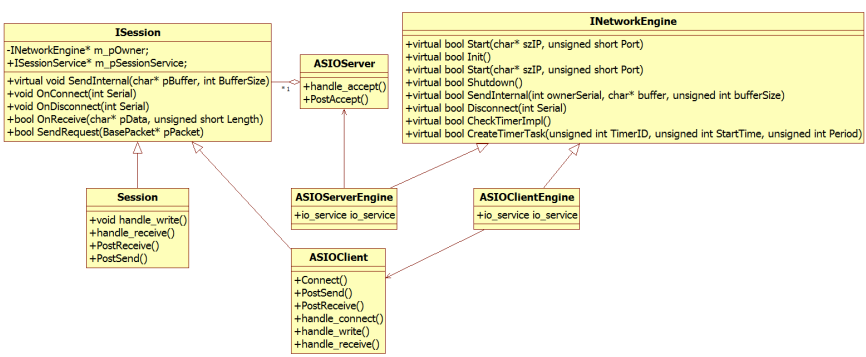
이제 남은 처리사항은 유저를 나타내는 객체인 세션의 구현이다. CGSF 네트워크 모듈은 유저를 표현하는 클래스가 SFProactorService였다. 이 클래스가 ISession을 상속받게 하고 데이터의 수신이나 유저의 접속 및 종료 시에 ISession 인터페이스의 메소드를 호출할 수 있도록 해야 한다. 해당 부분에 대해서는 CGSFNet 모듈을 설명하면서 언급한 바 있다. 기존 CGSFNet 모듈을 대체할 수 있는 커스터마이징 이론을 배웠으니 이제 Boost.Asio 모듈을 CGSF에 적용해 보자.

2.3 Boost.Asio 커스터마이징

지금부터는 현재까지 사용했던 CGSFNet.dll을 대체하는 네트워크 엔진을 제작해 보겠다. Boost.Asio 라이브러리를 적용하는 데 『Boost.Asio를 사용한 네트워크 프로그래밍』(한빛미디어, 2013)[05] 저자인 최흥배 님의 채팅 서버 소스의 코드를 사용하고 CGSF 네트워크 모듈 양식에 맞게 포팅한다.[06]

이 다이어그램은 ASIONet 모듈의 클래스 관계를 나타낸다. ASIOServerEngine 클래스와 ASIOClientEngine 클래스는 엔진 레이어에서 호출할 수 있게 INetworkEngine의 인터페이스를 구현하였다. ASIOServerEngine 클래스는 ASIOServer 클래스를 참조하고, ASIOServer 클래스는 Acceptor에 대한 처리와 서버에 접속한 유저 세션에 대한 관리를 책임지고 있다. Session 클래스는 유저 세션을 나타내며 엔진 레이어와 통신하기 위해 ISession 인터페이스를 상속한다.

[그림 2-5] Boost.Asio 모듈 커스터마이징 다이어그램



05 http://www.hanbit.co.kr/ebook/look.html?isbn=9788968486111

06 이 부분은 ASIONet 프로젝트를 참조하기 바란다.

[표 2-1] Session/ASIOClient 클래스의 메소드

클래스	메소드	설명
Session	PostSend	비동기 쓰기 요청
	PostReceive	비동기 읽기 요청
	handle_receive	비동기 읽기 요청 결과 처리
	Handle_write	비동기 쓰기(전송) 요청 결과 처리
ASIOClient	Connect	서버에 연결 시도
	handle_connect	서버 연결 요청 결과 처리

ASIOClient도 ASIOServer 클래스처럼 세션을 관리하는 형태로 구현해도 상관없으나 클라이언트 입장에서는 세션이 하나이므로 ASIOClient 클래스가 ISession 인터페이스를 상속하도록 하였다(클라이언트에서 다중 연결이 필요하다면 해당 구조는 변경되어야 한다).

이제 코드를 살펴보자(서버 쪽만 살펴본다).

[코드 2-17] AsioSever 클래스

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

        pSession->Init();
        pSession->OnConnect(pSession->SessionID());
        pSession->PostReceive();
        PostAccept();
     }
     else
     {
```

```
        std::cout << " error No: " << error.value() << " error Message: " <<
error.message() << std::endl;
    }
}
```

AsioServer 클래스의 handle_accept 메소드다. Acceptor가 세션 생성에 성공하면 handle_accept 메소드가 호출되는데, 이 메소드에서는 세션을 초기화한 뒤 유저가 접속했다는 메시지를 엔진 레이어에 전달하기 위해 Session 클래스의 OnConnect 메소드를 호출한다. 이후 유저가 전송한 데이터를 읽기 위해 Session 객체의 PostReceive 메소드를 호출해서 읽기 요청을 하고, Acceptor는 PostAccept 메소드를 요청하여 새로운 유저의 접속 요청에 대한 처리를 대기하게 한다.

[코드 2-18] Session 클래스

```
void Session::SendInternal(char* pBuffer, int BufferSize, int ownerSerial)
{
    PostSend(false, BufferSize, pBuffer);
}
```

Asio 서버는 비동기 쓰기 요청을 할 때 즉시 데이터 전송과 지연된 데이터 전송 기능을 제공한다. PostSend 메소드의 첫 번째 인자인 false는 지연된 데이터 전송을 하겠다는 의미다. 첫번째 인자에 true를 넣으면 데이터를 그 즉시 전송한다. 데이터 전송 요청 결과, 해당 데이터가 완전히 전송되지 않았을 가능성이 있으므로 handle_write 메소드에 데이터를 재전송할 수 있는 코드를 넣어준다.

[코드 2-19] Session 클래스의 handle_write 메소드

```
void Session::handle_write(const boost::system::error_code& error, size_t
bytes_transferred)
```

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

      m_SendDataSizeQueue.pop_front();

      if( m_SendDataQueue.empty() == false )
      {
         m_bCompletedWrite = false;

         char* pData = m_SendDataQueue.front();

         int nSize = m_SendDataSizeQueue.front();

         PostSend(true, nSize, pData);
      }
      else
      {
         m_bCompletedWrite = true;
      }
}
```

[코드 2-19]는 전송 데이터 큐(m_SendDataQueue)가 비어 있지 않으면 다시 비동기 쓰기 작업을 요청하는 코드다.

[코드 2-20] Session 클래스의 handle_receive 메소드

```
void Session::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;
            }
            m_pServer->CloseSession( m_nSessionID );
            OnDisconnect(m_nSessionID);
         }
         else
         {
            if(false == OnReceive(m_ReceiveBuffer.data(), bytes_transferred))
            {
               //강제로 끊게 하는 메소드는?
            }
            PostReceive();
         }
      }
```

[코드 2-20]은 유저로부터 데이터를 전송받았을 때 호출되는 메소드다. 읽기 과정에서 문제가 발생하면 연결을 종료하고, 그렇지 않으면 ISesssion 인터페이스의 OnReceive 메소드를 호출하여 데이터를 처리한 후 다시 비동기 읽기 작업을 요청한다.

이제 AsioNet 프로젝트를 빌드한 후 EngineConfig.xml 파일과 Connection.ini 파일에서 네트워크 라이브러리를 AsioNet.dll로 변경한 후 샘플 프로젝트를 실행해 보자. 전혀 다른 네트워크 모듈을 적용하였지만, 문제없이 잘 동작함을 알 수 있

을 것이다. Boost.Asio를 사용한 네트워크 엔진은 CGSF에 여러 네트워크 엔진을 적용할 수 있음을 보여주는 실험적인 엔진으로 차후 정교하게 수정할 예정이다.

2.4 정리

2장에서는 CGSFNet 네트워크 엔진에서 사용되었던 ACE 라이브러리를 분석하고, Boost.Asio 모듈을 가볍게 리뷰한 다음 CGSF에서 사용할 수 있게 커스터마이징해 보았다. 분명 ACE와 Boost.Asio는 전혀 다른 프레임워크지만, 네트워크 모듈로 동작하는 방식은 별로 다르지 않음을 알 수 있다. 다른 서버 라이브러리들도 설명한 범주에서 많이 벗어나지 않으므로 괜찮은 서버 라이브러리가 있다면 CGSF의 네트워크 모듈로 적용하는 것에 도전해 보길 바란다. 물론 커스터마이징된 네트워크 모듈을 서드 파티로 제공해 준다면 언제든지 환영하는 바이다.

최신 CGSF 소스 코드는 책에서 설명한 내용과 조금 다를 수 있어서 내용상 약간의 혼선이 있을 수 있다. 책의 내용과 일치하는 버전은 github의 release 페이지[07]에 올려져 있으므로 내려받길 바란다. CGSF의 코드는 계속해서 리팩토링되므로 책의 내용과는 많이 달라지겠지만, 그 근본적인 원리는 동일하기 때문에 책의 내용만 제대로 숙지한다면 CGSF의 최신 코드에도 쉽게 적응할 수 있을 것이다.

07 https://github.com/pdpdds/CGSF/releases