KGDS 2007

『 Proactor 패턴 기반의 IOCP 서버 어플리케이션 디자인 』

outbreak@empal.com

안준석

목 차

그림 목차

표 목차

1 Abstract

대용량 서버 어플리케이션에서 네트워크 I/O와 관련한 소켓(Socket) 처리 방식의 차이는 동시 접속자를 처리하는 성능과 직결된다. Windows는 멀티스레드 환경을 감안하여 설계가 되었으며, 폴링이 아니라 IOCP(Input Output Completion Port) 라는 막강한 기능을 통한 이벤트처리(Event-driven) 방식을 택하고 다중 사용자를 처리해야 하는 시스템에 뛰어난 성능을 보여준다.

디자인 관점에서 Proactor 패턴은 Event-driven 방식의 어플리케이션이 비동기 방식으로 완료된 요청을 효과적으로 처리할 수 있게 한다. Proactor 패턴을 사용하면 효율적으로 다중처리를 할 수 있는 멀티쓰레드 어플리케이션 디자인이 가능해 진다.

이런 Proactor 패턴 구현을 위해 필요한 OS 기능을 IOCP 메커니즘이 모두 지원한다.

이번 발표에서는 IOCP와 Proactor 패턴에 대해 살펴보고, Proactor 패턴 기반의 IOCP 대용량 서버 디자인을 제안한다.

2 Introduction

2.1 IOCP 소개

IOCP는 Input Output Completion Port의 줄임말로 Microsoft Windows OS 에서 제공하는 비동기 I/O 이벤트 처리 기술이다. Socket을 이용한 네트워크 I/O 뿐만 아니라 File 등 다양한 I/O 에 대한 비동기 이벤트 처리가 가능하다.

Windows OS 에는 다양한 네트워크 I/O 처리 방식이 있는데, 다수의 Socket 에 대한 다중 처리에 IOCP가 최적인 것으로 나타난다.

Microsoft Network Programming for MS Windows S/E 실험결과

I/O Model	Attempted/Connected	Memory Used (KB)	Non-Paged Pool	CPU Usage	Threads	Throughput (Send/ Receive Bytes Per Second)
Blocking	7000/ 1008	25,632	36,121	10–60%	2016	2,198,148/ 2,198,148
	12,000/ 1008	25,408	36,352	5– 40%	2016	404,227/ 402,227
Non- blocking	7000/ 4011	4208	135,123	95–100%*	1	0/0
	12,000/ 5779	5224	156,260	95–100%*	1	0/0
WSA- Async Select	7000/ 1956	3640	38,246	75–85%	3	1,610,204/ 1,637,819
	12,000/ 4077	4884	42,992	90–100%	3	652,902/ 652,902
WSA- Event Select	7000/ 6999	10,502	36,402	65–85%	113	4,921,350/ 5,186,297
	12,000/ 11,080	19,214	39,040	50–60%	192	3,217,493/ 3,217,493
	46,000/ 45,933	37,392	121,624	80–90%	791	3,851,059/ 3,851,059
Over- lapped (events)	7000/ 5558	21,844	34,944	65–85%	66	5,024,723/ 4,095,644
	12,000/12,000	60,576	48,060	35–45%	195	1,803,878/ 1,803,878
	49,000/48,997	241,208	155,480	85–95%	792	3,865,152/ 3,834,511
Over- lapped (comple- tion port)	7000/ 7000	36,160	31,128	40–50%	2	6,282,473/ 3,893,507
	12,000/12,000	59,256	38,862	40–50%	2	5,027,914/ 5,027,095
	50,000/49,997	242,272	148,192	55–65%	2	4,326,946/ 4,326,496

[표 1] Windows I/O 처리 메커니즘 성능비교

2.2 IOCP 기능 개요

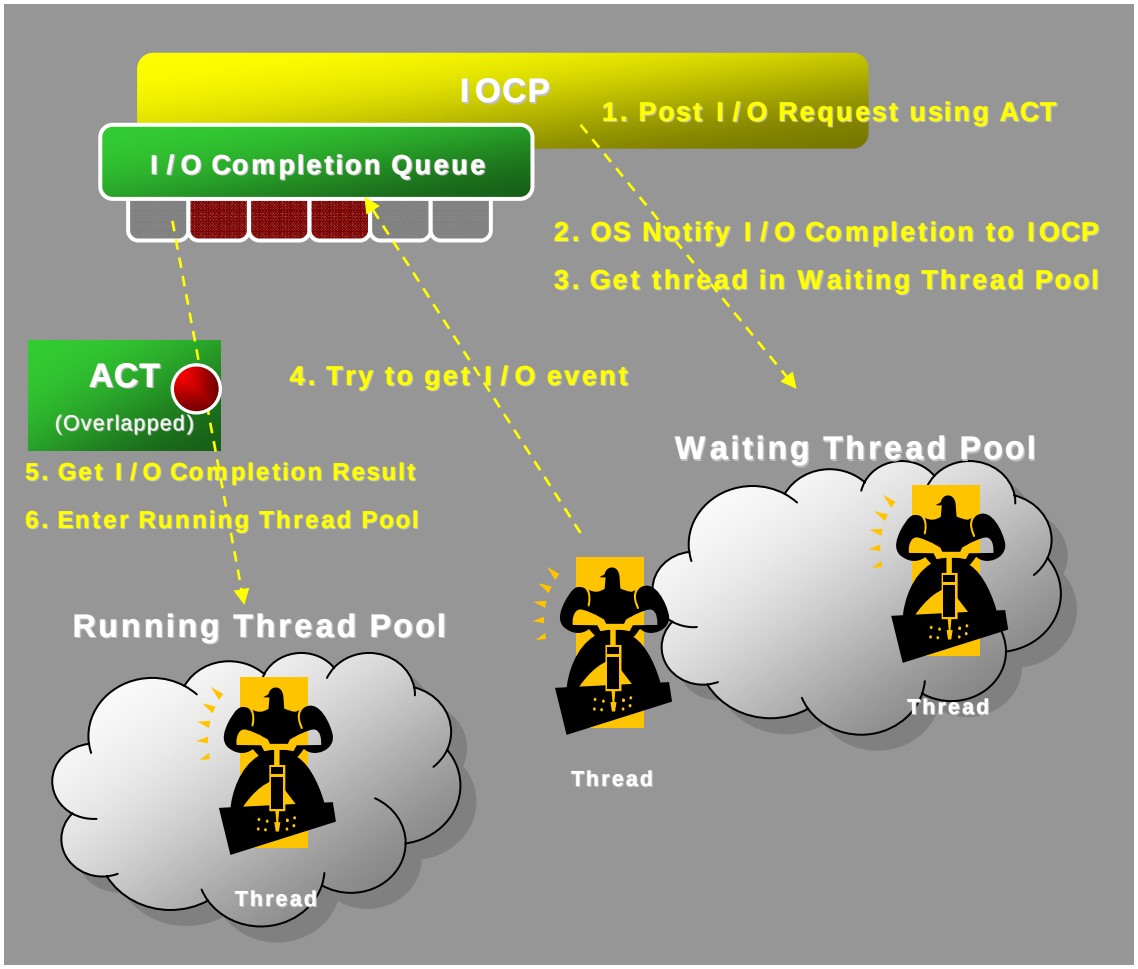
IOCP는 Completion Port라는 OS 커널 오브젝트를 ::CreateIoCompletionPort() Windows 함수로 생성 한다. 이 커널 객체를 이용해 I/O를 감시하고, I/O 완료를 통지하는 메커니즘을 IOCP 라고 통칭한다.

Completion Port 커널 오브젝트를 이용해서 지정된 수만큼의 쓰레드들이 Overlapped IO 를 처리하게 된다. Overlapped IO는 비동기 I/O 가 가능하게 해주는 메커니즘으로 버퍼 복사 오버헤드 방지, 중첩 입출력, ACT(Asynchronous Completion Token)로의 확장 등 다양한 특성을 갖고 있다. 지정된 쓰레드들은 IOCP 에 의해 OS에서 관리되는 Thread Pool 을 생성한다.

IOCP는 다양한 Windows I/O를 위하여 만들어진 것이지만 본 발표에서는 Socket I/O 에 적용된 내용을 다룬다.

2.3 IOCP 작동 원리

2.3.1 비동기 I/O 처리



[그림 1] IOCP 작동 원리 개념도

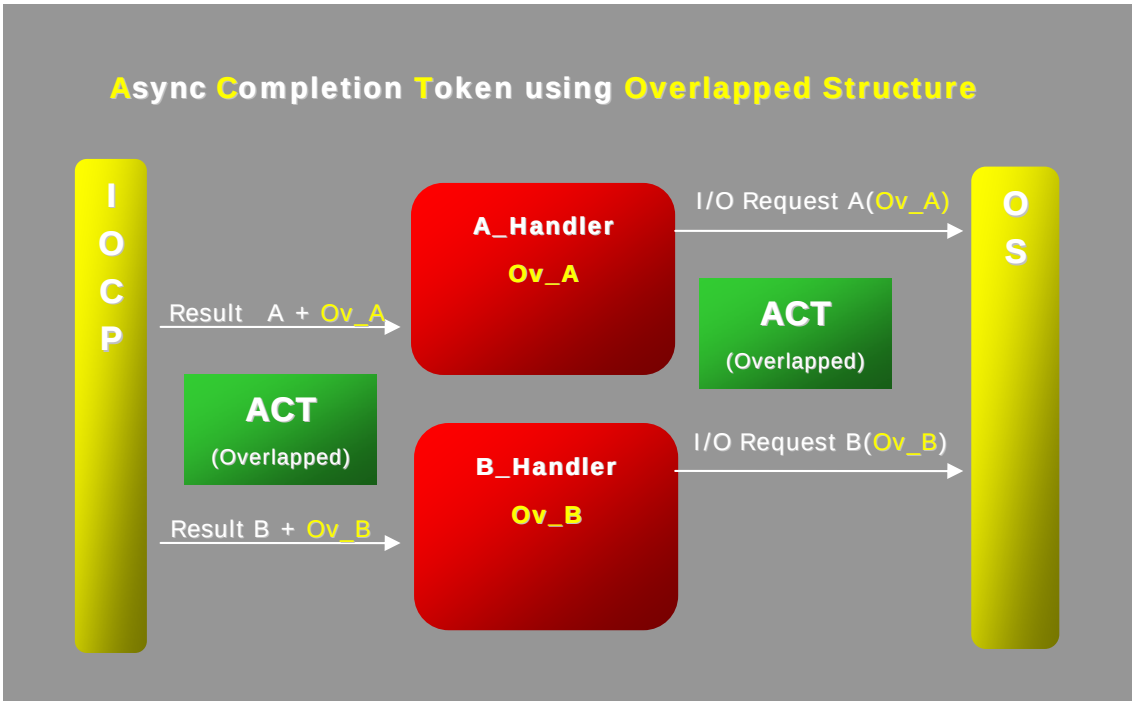
비동기 I/O 란 처리 결과 응답을 기다리지 않고, 다음 처리를 한다는 것이다. 즉, Send() 함수를 호출했을 때, 데이터의 전송이 완료됐는지 확인하지 않고 바로 함수가 리턴 된다. 전송 결과는 OS가 완료를 알려주는데, 다수개의 완료 통지를 위한 최적의 메커니즘이 IOCP 이다. IOCP를 이용한 비동기 I/O는 다음과 같은 순서로 진행된다.

1. I/O 요청을 호출한다.
   (WSASend(), WSARecv(), AcceptEx(), DisconnectEx()..)
2. OS에서 I/O 완료를 Completion Port 객체를 이용해 알려준다.
3. 대기 쓰레드 풀에서 쓰레드 A 를 얻는다
4. 쓰레드 A는 I/O 완료 큐에서 I/O 완료 정보 얻기를 시도한다

(GetQueuedCompletionStatus() 함수사용)

5. Overlapped 구조체 or 확장된 구조체의 완료 정보를 얻는다.
6. 쓰레드 A는 Running Thread Pool 에 들어간다.

2.3.2 Overlapped 구조체를 사용한 Asynchronous Completion Token 구현



[그림 2] Async Completion Token using Overlapped Structure

Asynchronous Completion Token (이하 ACT)는 이벤트기반 어플리케이션에서 비동기 요청을 효율적으로 처리하는 방법이다. 데이터 흐름에서 demultiplexing 하기 보다는 결과를 어떻게 처리하는지에 대한 정보를 담은 Token을 이용하는 것이다.

비동기 요청에 대한 결과 응답(completion events) 시 어떤 Handler에게 응답해야 하는지 알기 힘들다. ACT를 이용하면 이 문제에 대한 처리를 쉽게 할 수 있다. 비동기 요청 시 ACT를 함께 보내고 ACT에 어떤 Handler가 결과를 처리하는지에 대한 정보를 포함 시킨다.

IOCP 는 Overlapped 구조체를 ACT 로 사용한다. 예를 들면, WSARecv() 함수를 사용할 때, 마지막 파라미터로 Overlapped 구조체를 넣어준다. IOCP가 결과를 통지 할 때, 이 구조체가 함께 전달 된다.

C++ 상속 메커니즘을 이용해 Overlapped 구조체를 상속받아 확장시킨 클래스를 ACT로 사용하면, 결과 처리 시 다양한 정보를 사용할 수 있다.

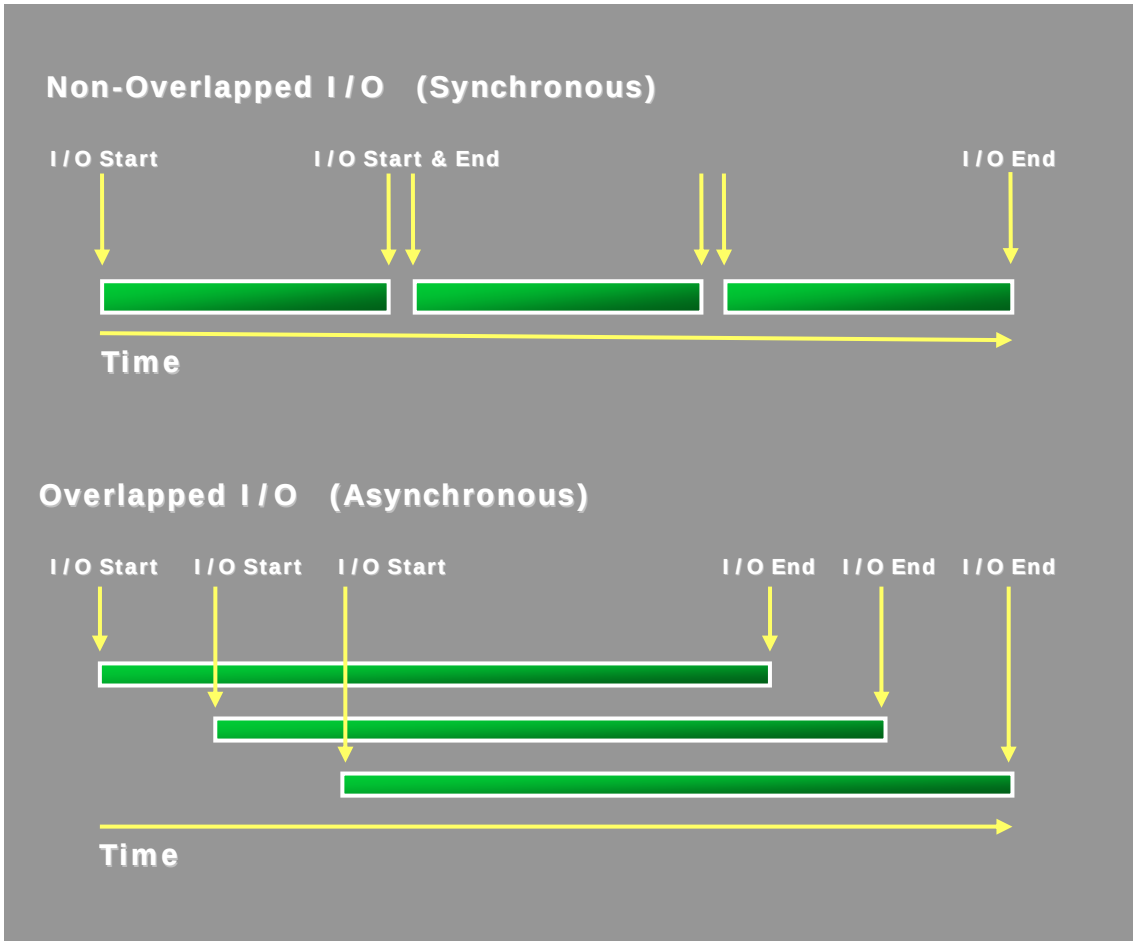
2.3.3 Overlapped I/O의 특징

Overlapped I/O는 비동기 I/O 가 가능하게 해주는 메커니즘이다.

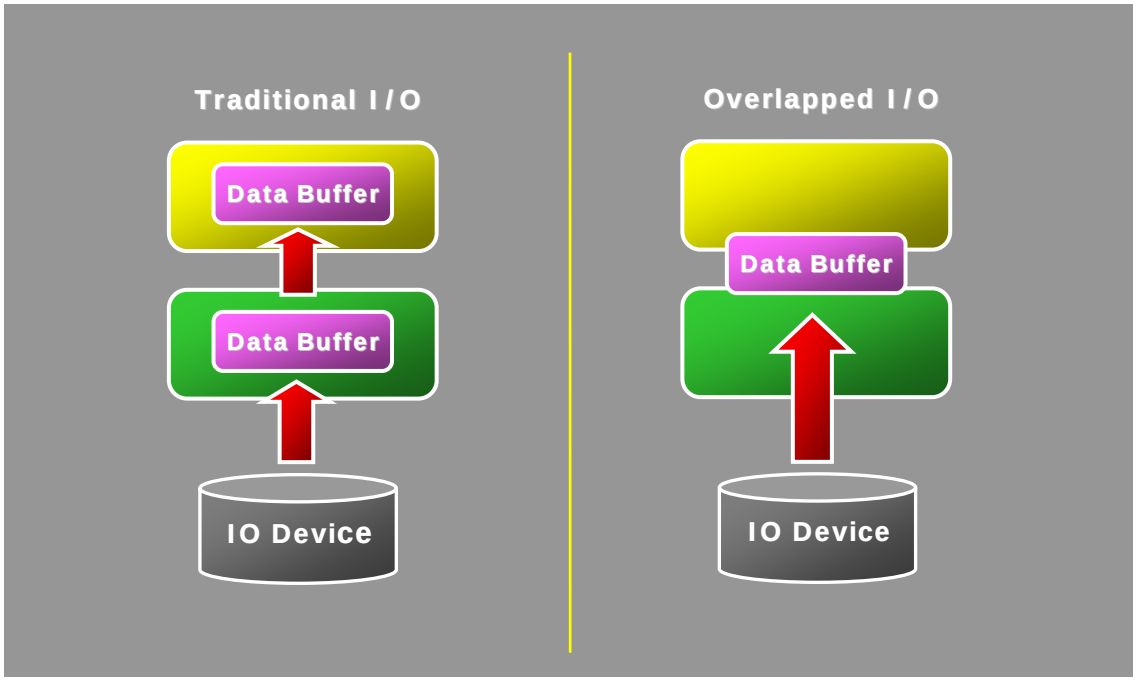
Overlapped I/O는 Overlapped 구조체를 이용서 이루어지는데, 이 구조체 멤버에 대한 사용은 OS만의 권한으로 한정되어 있다.

Overlapped I/O는 다음과 같은 특성을 갖고 있다.

1) 중첩 입출력을 통한 비동기 I/O 기능 구현
2) I/O 버퍼를 사용자 지정할 수 있게 해서 버퍼 복사 오버헤드 방지
3) ACT(Asynchronous Completion Token)로의 확장 등 다양한 특성을 갖고 있다.

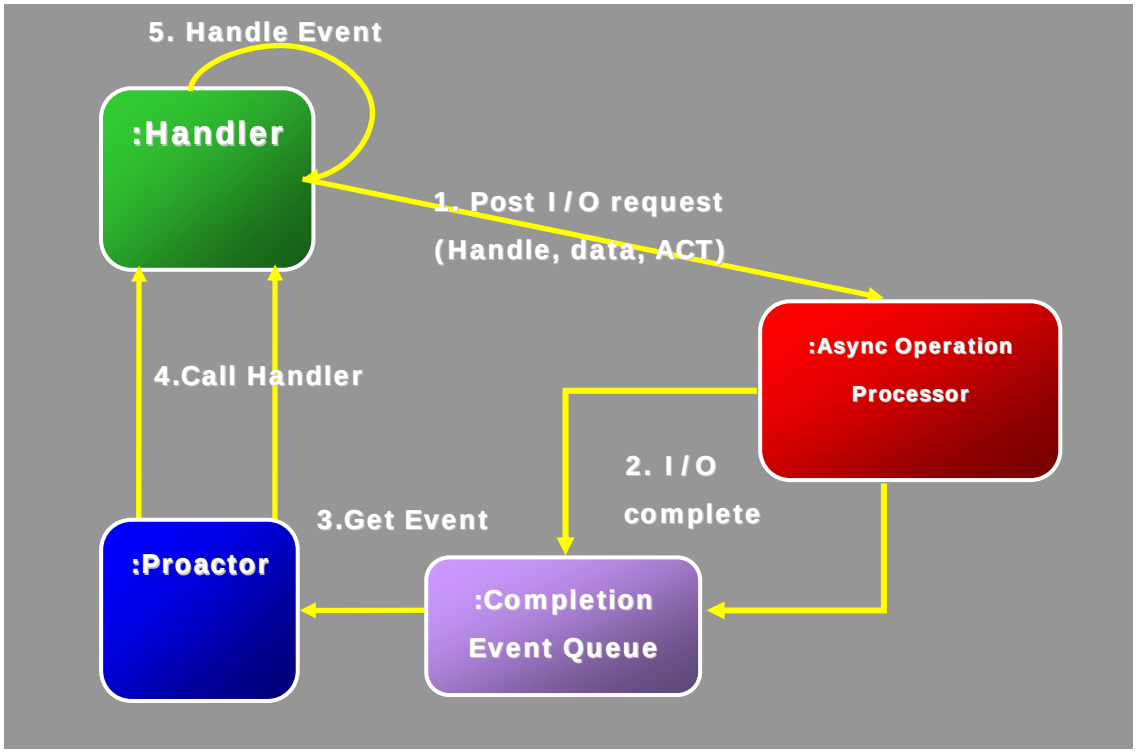


[그림 3] 중첩 I/O 메커니즘



[그림 4] 버퍼 오버헤드 방지 메커니즘

2.4 Proactor 패턴 구조



[그림 5] Proactor 패턴

Proactor 패턴은 [1]Douglas Schmidt(외3명)에 의해 제안 된 I/O 이벤트 처리 모델이다. Proactor 패턴은 단어의 의미처럼 '먼저 처리하는' 패턴 이다. 비동기 I/O 요청을 '먼저' 하고 Async Operation Processor 에서 통지되는 I/O 완료에 대한 처리를 Proactor 가 하게 하는 것이다.

그림(5) 를 보면 Proactor 패턴의 원리를 쉽게 알 수 있다.

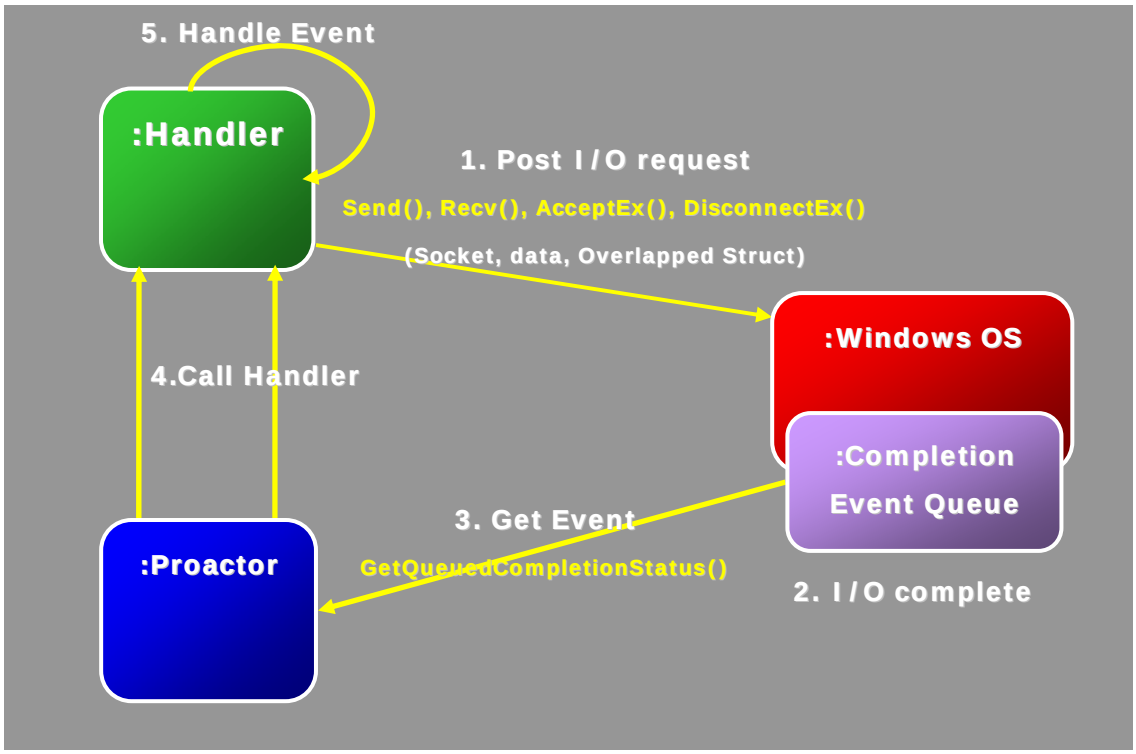
1. 비동기 I/O 요청 한다. 결과를 기다리지 않는다.
2. Async Operation Processor 가 비동기 I/O 요청을 받고, I/O 요청이 완료되면 Completion Event Queue에 I/O 완료 결과를 집어 넣는다.
3. Proactor 는 Completion Event Queue 에 완료 결과가 들어오면 I/O 완료 통지를 받고, 완료 이벤트를 처리한다.
4. Proactor 는 ACT를 이용해 I/O 완료 결과를 처리할 Handler를 호출한다.
5. Handler 는 I/O 완료 이벤트에 대한 고유의 처리를 한다.

Proactor 패턴은 Event-driven 어플리케이션에서 I/O 완료 결과를 효과적으로 처리하는 메커니즘을 제시한다. Completion Event Queue 를 이용함으로써 동시성에 대한 안정성과 성능 이점을 얻을 수 있게 된다. 또한 Proactor에서 다중 쓰레드를 구현함으로써 I/O 완료 통지 처리를 효율적으로 구현할 수 있다.

1) Completion Event Queue 를 사용 함으로써 동시성에 대한 안정성 및 성능 이점을 얻을 수 있다.
2) ACT 를 이용함으로써 결과처리를 위한 demultiplexing 오버헤드를 줄일 수 있고, Token 방식으로 결과를 처리함으로써 I/O를 요청한 클라이언트에 대한 데이터 흐름을 연결 시키지 않아도 된다.
3) ACT, Proactor, Handler 로 각각 처리 객체를 분리 함으로써 디자인 측면에서 확장이 용이해지고 개념 이해가 쉬워진다.

이런 Proactor 패턴을 구현할 때, OS 수준의 기능 지원이 선행 되야 한다. 즉 비동기 I/O 를 위한 기능을 제공해야 하는 것이다.

IOCP는 Proactor 패턴을 이용해 서버 어플리케이션을 디자인 할 때 필요한 OS 수준의 기능을 모두 지원한다.



[그림 6] IOCP를 이용한 Proactor 패턴 구현

그림(6)은 Proactor 패턴의 각 요소들을 Windows 기반의 IOCP로 구현 한 도식이다. Async Operation Processor 과 Completion Event Queue 역할은 Windows OS 가 제공해 주고, WINDOWS API 인 GetQueuedCompletionStatus() 함수를 이용해 Proactor 에서 비동기 I/O 완료 통지를 받게 한다.

본 발표에서는 Windows OS 에서 IOCP를 이용해 Proactor 패턴을 응용한 대용량 서버 디자인을 제시 하고자 한다.

3 Proactor 패턴을 응용한 대용량 IOCP 서버 디자인

3.1 디자인 목표

대용량 네트워크 I/O 처리를 효율적으로 한다
각 기능의 확장을 쉽게 한다
개념을 쉽게 이해하고 사용할 수 있게 한다
Windows API 를 적극 활용한다.
자원을 효율적으로 사용한다.

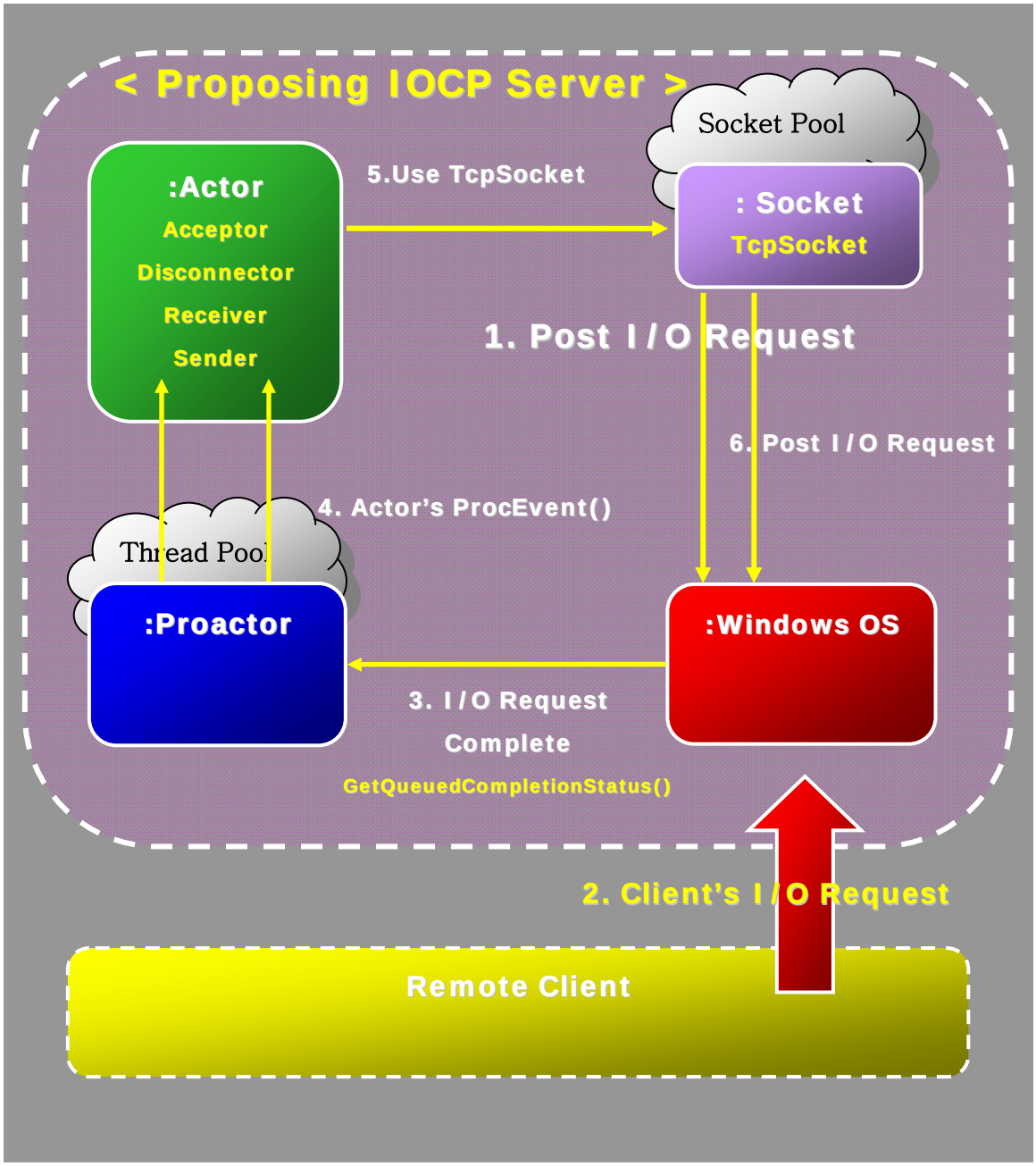
Proactor 패턴을 이용한 디자인 장점을 얻으면서 IOCP의 성능을 최대한 끌어 올릴 수 있도록 한다. 뿐만 아니라 WINDOWS 네트워크 API 를 활용한다. WINDOWS API AceeptEx(), DisconnectEx()를 결합 시켜 소켓풀을 구현한다. 소켓풀을 이용해 자원을 효율적으로 사용한다. 또한 IOCP에서 제공하는 쓰레드풀을 활용할 수 있도록 구조를 유연하게 만든다.

3.2 주요 Windows API 설명

API 이름	API 역할
CreateIoCompletionPort()	Completion Port 커널 객체를 생성한다.
GetQueuedCompletionStatus()	I/O Completion Queued 에서 완료 이벤트를 얻는다.
AcceptEx()	Socket 을 미리 Accept 해놓는다.
DisconnectEx()	Socket 의 연결을 끊고 Socket 재사용을 요청한다 (구현프로그램에서는 TransmitFile() 함수가 대신 사용되었다. OS 버전에 따른 지원문제가 있음)
WSARecv()	Socket 으로 전송된 데이터를 받는다.
WSASend()	Socket 을 이용해 데이터를 전송한다.

[표 2] 주요 Windows Network API

3.3 Design Overall Diagram



[그림 7] Proactor 패턴 기반의 IOCP 대용량 서버 디자인

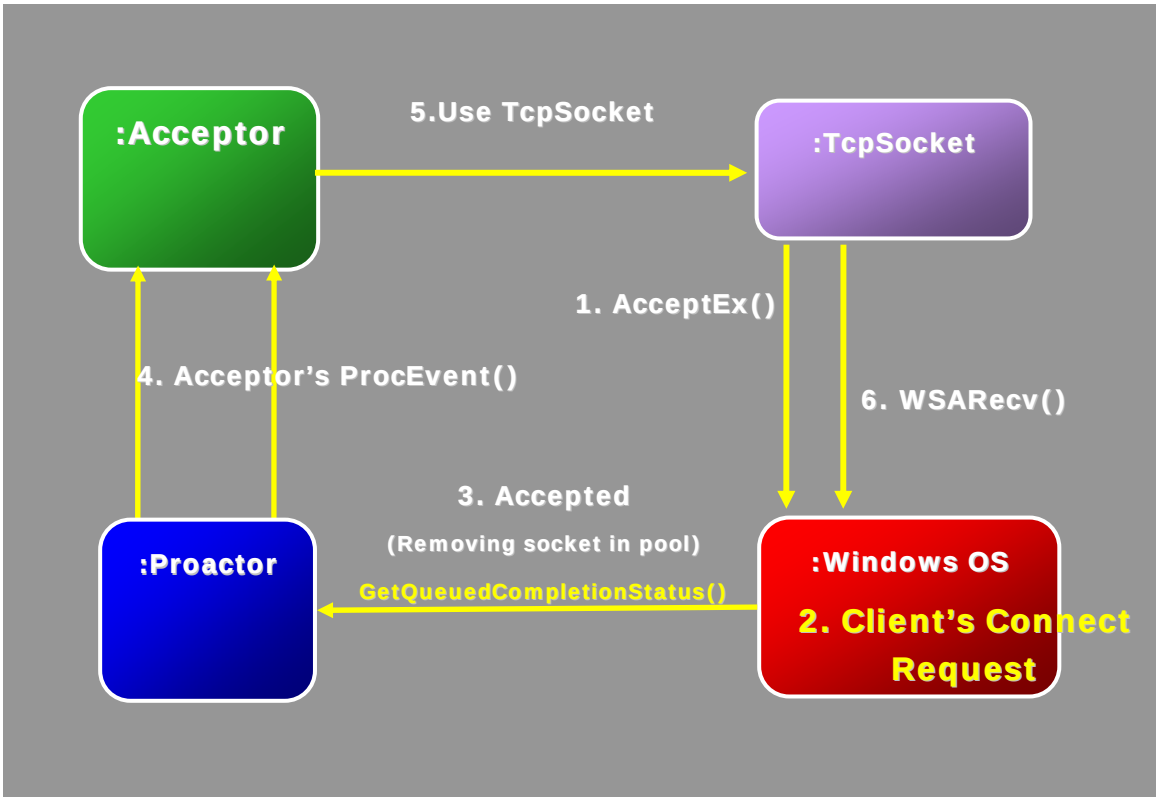
3.4 구현 : 주요 클래스 설명

Name	Super Class	Resposibilities
Act	OVERLAPPED	IOCP에서 사용되는 OVERLAPPED 확장 클래스. ACT
TcpSocket		SOCKET 을 멤버로 갖고 있고, Recv(), Send() 등 네트워크 기능을 갖고 있다
TcpAct	Act	TCP 용으로 사용하기 위한 ACT
Proactor		GetQueuedCompletionStatus()를 이용해서 I/O 완료를 수신한다.
Actor		가상 ProcEvent() 함수를 갖고 있고, Proactor가 수신한 I/O 완료 처리를 실질적으로 한다.
Acceptor	Actor	Accept 에 대한 완료 처리자
Disconnector	Actor	Disconnect 에 대한 완료 처리자
Sender	Actor	Send 에 대한 완료 처리자
Receiver	Actor	Recv 에 대한 완료 처리자

[표 3] 주요 구현 클래스 설명

4 구현 : 각 Actor의 행동 설명

4.1 Accept



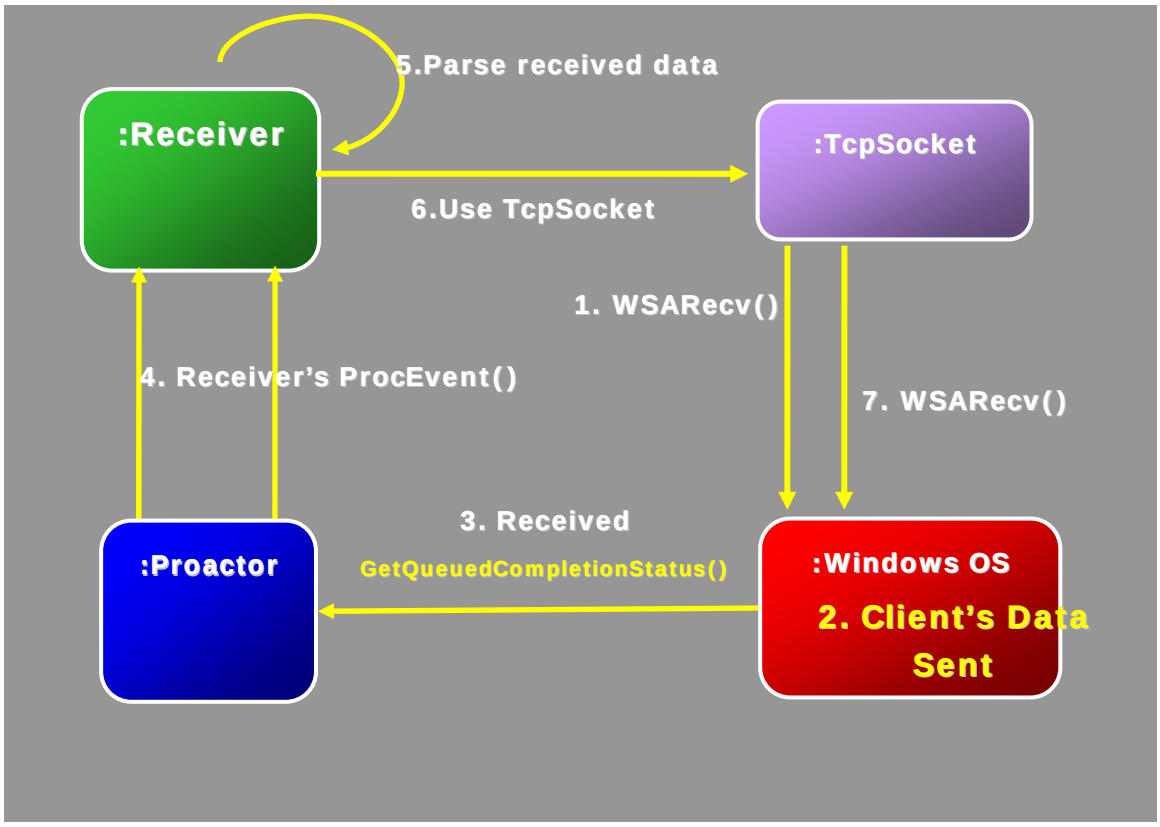
[그림 8] Accept 기능 실행도

Acceptor 는 원격 사용자의 접속을 처리 한다. 일반적인 Accept 기능은 사용자가 접속을 요청했을 때 Accept 작업을 하지만 AcceptEx() 함수와 IOCP 기능을 이용하면 <u>미리 Accept 작업 처리를 할 수 있다.</u>

1. AcceptEx() 함수 실행. 해당 소켓에 대해 미리 Accept 처리를 해놓는다.
2. 원격 사용자가 Connect 요청을 한다.
3. Proactor로 미리 Accept 해 놓은 소켓 중 한 개에 대한 Accept 완료가 통지된다.
4. Proactor는 Acceptor의 ProcEvent() 함수를 호출한다.
5. Acceptor는 접속처리를 하기 위해 ACT 에서 해당 TcpSocket을 사용한다.
6. Accept 완료된 소켓은 데이터를 받을 수 있는 상태가 되기 위해 WSARecv를 비동기 호출한다.

AcceptEx() 를 이용해 소켓들에 미리 Accept 요청을 해놓음으로써 소켓풀을 생성한다. 소켓풀은 소켓에 대해 AccpetEx()를 호출하면 풀에 소켓이 추가되고, AcceptEx() 완료 통지가 오면 소켓풀에서 소켓이 제거 된다.

4.2 Recv

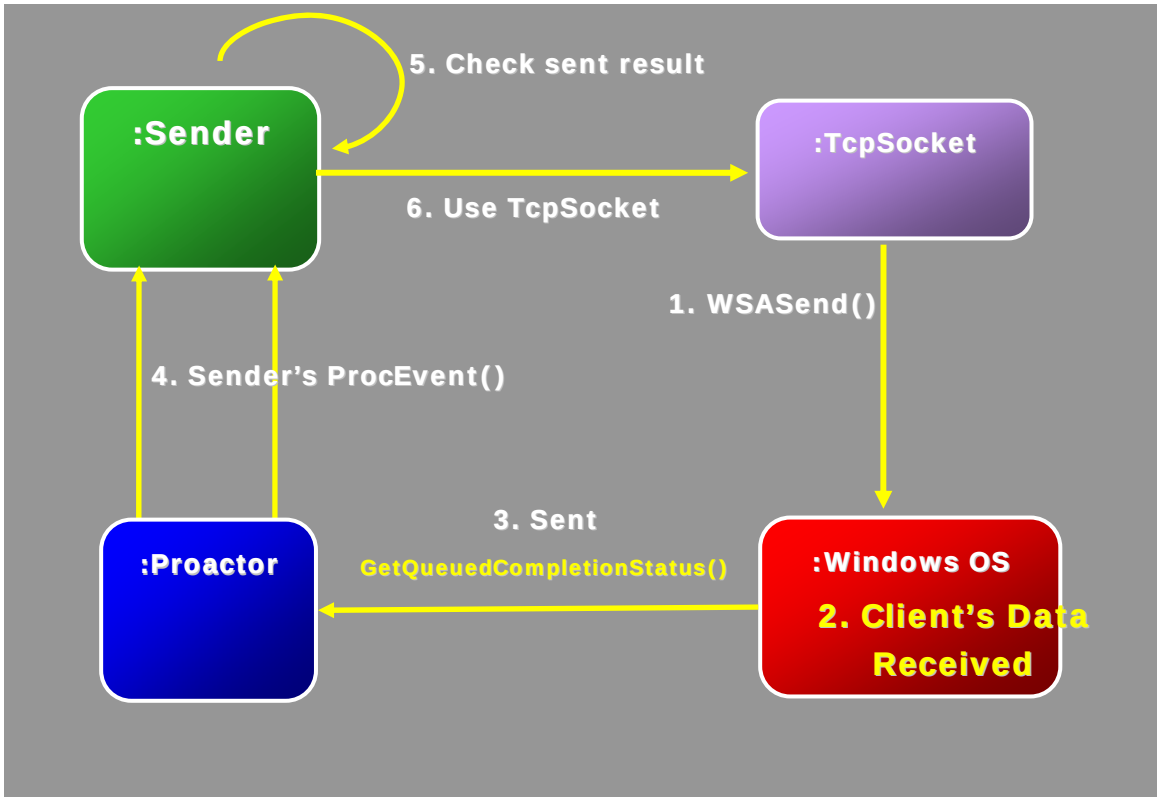


[그림 9] Recv 기능 실행도

Receiver 는 원격 사용자로부터 수신 받은 데이터를 처리한다. 데이터를 파싱해서 DB에 접근하는 등의 작업을 처리 할 수 있다. WSARecv() 함수를 미리 호출해서 비동기 수신 완료 통지를 기다린다. 비동기 수신 완료는 원격 사용자로부터 데이터가 오면 통지 된다.

1. WSARecv() 호출. 비동기 수신 요청을 한다.
2. 원격 사용자가 데이터를 보낸다.
3. Proactor로 수신 완료 통보가 통지된다.
4. Proactor는 Receiver의 ProcEvent() 함수를 호출한다.
5. Receiver는 수신된 데이터(버퍼)를 처리한다.
6. Receiver는 수신 완료를 처리하기 위해 TcpSocket을 사용한다.
7. TcpSocket은 다음 데이터를 받을 수 있는 상태가 되기 위해 WSARecv를 비동기 호출한다.

4.3 Send

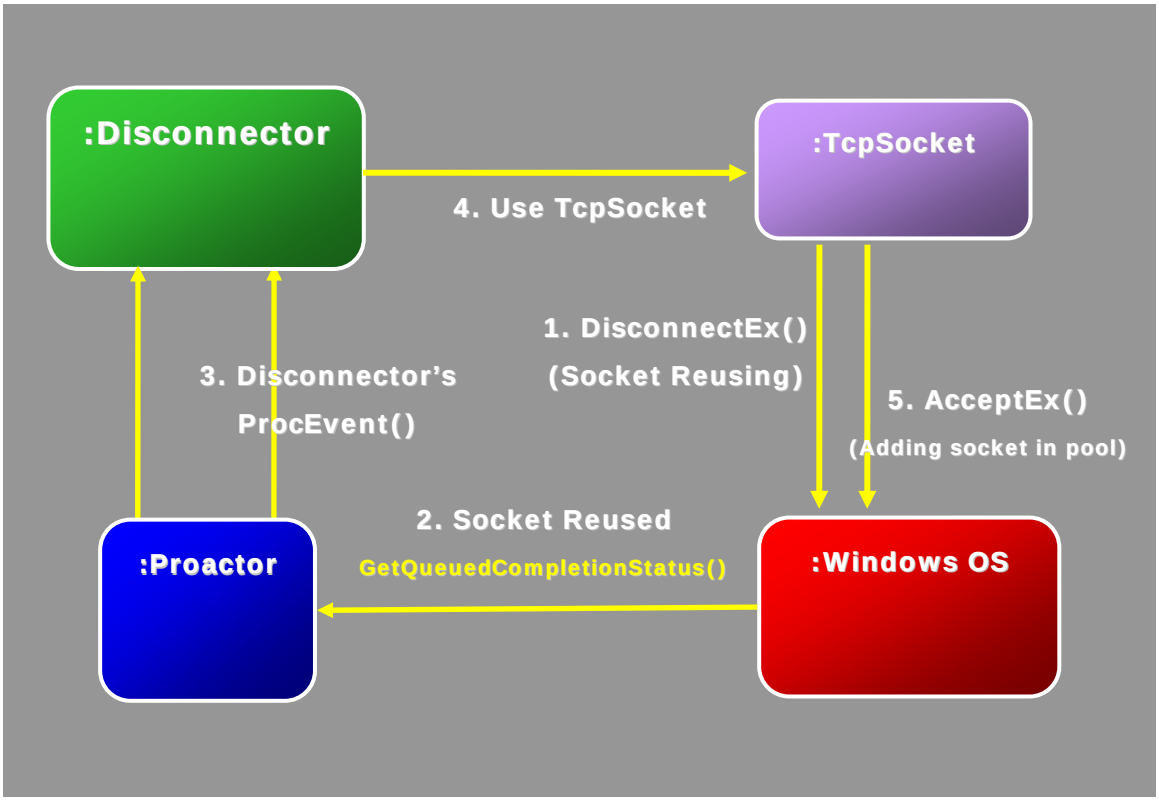


[그림 10] Send 기능 실행도

Sender 는 송신 완료를 처리한다. 데이터 송신의 경우 WSASend() 함수를 호출해 데이터를 전송한다.

1. WSASend() 호출. 비동기 송신 요청을 한다.
2. 원격 사용자가 데이터를 수신한다.
3. Proactor로 송신 완료 통보가 통지된다.
4. Proactor는 Sender의 ProcEvent() 함수를 호출한다.
5. Sender는 송신된 Byte 크기를 체크해서 요청한 크기만큼 송신 됐는지 확인한다.
6. Sender는 송신 완료를 처리하기 위해 TcpSocket을 사용한다.

4.4 Disconnect



[그림 11] Disconnect 기능 실행도

Disconnector 는 원격 사용자의 접속종료를 처리 한다. DisconnectEx() 함수와 IOCP 기능을 이용하면 **소켓을 재사용 할 수 있다.**

1. DisconnectEx() 함수 실행. 해당 소켓에 대해 접속 종료 및 재사용을 요청한다.
2. Proactor로 소켓 재사용 완료가 통지된다.
3. Proactor는 Disconnector의 ProcEvent() 함수를 호출한다.
4. Disconnector는 소켓풀에 재사용된 소켓을 집어 넣기 위해 TcpSocket을 사용한다.
5. AcceptEx() 를 호출해서 재사용된 소켓을 소켓풀에 집에 넣는다.

5 Conclusion

인터넷 환경이 발전하면서 대규모 유저를 수용할 수 있는 대용량 서버 어플리케이션 개발에 대한 요구가 더욱 커지고 있다. 특히 게임산업은 온라인화에 따라 대용량 서버 어플리케이션 개발이 필수가 되었다. 대부분 Windows 기반의 IOCP 서버로 개발 되고 있고 수많은 결과물에 의해 Windows 기반의 IOCP 서버에 대한 성능과 안정성이 검증되고 있다.

본 발표에서는 검증된 IOCP 기술을 바탕으로 다음과 같은 디자인 목표를 세웠다.

대용량 네트워크 I/O 처리를 효율적으로 한다
각 기능의 확장을 쉽게 한다
개념을 쉽게 이해하고 사용할 수 있게 한다
Windows API 를 적극 활용한다.
자원을 효율적으로 사용한다.

이를 위해 Douglas Schmidt 가 제안한 Proactor 패턴을 도입했고, 그에 대한 응용을 통해 'Proactor 패턴 기반의 대용량 IOCP 서버 디자인' 을 제시했다.

Proactor 패턴의 간결함과 명확성으로 인해 확장성 있고 개념을 쉽게 이해할 수 있는 디자인이 가능했고 Window OS의 IOCP 기술을 사용함으로써, 대용량 네트워크 I/O 처리를 효율적으로 할 수 있었다. 또 AcceptEx()와 DisconnectEx() 함수를 이용해 소켓을 재사용할 수 있는 소켓풀을 만들어 자원을 효율적으로 사용할 수 있게 했다.

6 **Reference**

[1] Douglas C. Schmidt

Pattern Oriented Software Architecture Volumn.2

[2] Jonathan Lemon, FreeBSD Project

Kqueue: A generic and scalble event notification facility"

[3] Jeffrey Richter and Jason D. Clark

Programming Server-Side Application for Microsoft Windows 2000

[4] Anthony Jones and Jim Ohlund

Network Programming for Microsoft Winodws 2nd

[5] Kurose and Ross

Computer Networking

[6] GoF

Design Pattern

[6] Jeffrey Richter

Programming Application for Windows Fourth Edition

7 Appendix. Implementation Code

Act

```
#include "Net.h"
//-----------------------------------------------------------
// 이름 : Act ( Async Completion Token )
//
// 역할 : 1) OVERLAPPED 확장
//       2) 추상클래스.
//       3) Proactor의 ProcEvents에서 I/O 완료시 Complete() 와 Error() 콜
//
// 기타 : TcpAct 의 SuperClass
//-----------------------------------------------------------
class Actor;
class Act : public OVERLAPPED
{
public:
       Act()
       {
              hEvent = NULL;
              Internal = 0;
              InternalHigh = 0;
              Offset = 0;
              OffsetHigh = 0;

              Actor_ = NULL;
       }

       virtual void Complete( DWORD bytes_transferred ) = 0;
       virtual void Error( DWORD error ) = 0;
public:
       Actor*  Actor_;
};
```

Actor

```
#pragma once
#include "Act.h"
#include "Proactor.h"
//-------------------------------------------------------
// 이름 :       Actor
//
// 역할 :       1)  ACT의 Complete()에서 ProcEvent() 콜
//              2)  ACT의 Error()에서 ProcError() 콜
//              3)  추상클래스
//
// 기타 :       Acceptor, Disconnector
//              , Receiver, Sender 의 SuperClass
//-------------------------------------------------------
class Actor
{
public:
        Actor()
        {
                Proactor_ = NULL;
        }

        virtual void ProcEvent( Act* act, DWORD bytes_transferred ) = 0;
        virtual void ProcError( Act* act, DWORD error ) = 0;
public:
        Proactor* Proactor_;
};
```

Proactor

```
#pragma once

#include "Act.h"

//-------------------------------------------------------
// 이름 :       Proactor
//
// 역할 :       1)      비동기 I/O 완료 수신
//              2)      I/O 완료를 수신하는 ProcEvents()를
//                      다중쓰레드가 실행할 수 있음
//              3)      Register() 를 이용해 SOCKET을 IOCP에 등록
//              4)      ProcEvents()에서
//                      GeGetQueuedCompletionStatus() 함수 사용
//-------------------------------------------------------

class Proactor
{
public:
        Proactor();

public:
        static UINT    WINAPI ThreadProc( void* pProactor );

public:
        void Init( int numofthreads );
        void Register( HANDLE handle );
        void PostPrivateEvent(DWORD pId, Act* pActor );
        void ProcEvents();

public:
        HANDLE  Iocp_;

        DWORD   TimeOut_;
```

```
        DWORD   NumOfThreads_;

};
```

TcpAct

```
#pragma once

#include "Act.h"
#include "Actor.h"

//-------------------------------------------------------
// 이름 :       TcpAct
//
// 역할 :       1)      OVERLAPPED 확장
//              2)      구체클래스.
//              3)      Proactor의 ProcEvents에서 I/O 완료시
//                      Complete() 와 Error() 콜
//
// 기타 :       Actor 의 SubClass
//-------------------------------------------------------

class TcpSocket;
class TcpAct : public Act
{
public:
        TcpAct()
        {
                TcpSocket_ = NULL;
        }

        void Complete( DWORD bytes_transferred )
        {
                Actor_->ProcEvent( this, bytes_transferred );
```

```
	}

	void Error( DWORD error )
	{
		Actor_->ProcError( this, error );
	}

	void Init( Actor* actor, TcpSocket* tcpsocket )
	{
		Actor_ = actor;
		TcpSocket_ = tcpsocket;
	}

	const TcpSocket* GetTcpSocket()
	{
		return TcpSocket_;
	}

public:
	TcpSocket* TcpSocket_;
};
```

TcpSocket

```
#pragma once

#define BUFSIZE 8192

#include "TcpAct.h"

//-------------------------------------------------------
// 이름 :	TcpSocket
//
// 역할 :	1)	SOCKET을 멤버로 갖는다
```

```
//             2)      구체클래스.
//             3)      소켓의 행위를 Wrapping 함.
//                     Send, Recv, Reuse, Disconnect I/O 행위를 함
//
// 기타 :      각 Actor에서 TcpSocket을 이용해 I/O 행위를 함
//------------------------------------------------------

class Proactor;
class Acceptor;
class Disconnector;
class Sender;
class Receiver;

class TcpSocket
{

public:
       TcpSocket();

public:
       void Init();
       void InitBuf();
       void  InitAct(  Proactor*  proactor,  Acceptor*  acceptor,  Disconnector*
disconnector, Sender* sender, Receiver* receiver );

       SOCKET GetSocket() const;

public:
       void Recv();
       void Send( BYTE* buf, int buflen );
       void Reuse();
       void Disconnect();

public:
       enum ACT_TYPE
```

```
        {
                ACT_SEND = 0,
                ACT_RECV,
                ACT_ACCEPT,
                ACT_DISCONNECT,
                ACT_TYPE_CNT
        };

        TcpAct                  Act_[ACT_TYPE_CNT];

public:
        SOCKET                  Socket_;
        SOCKADDR_IN             Addr_;

public:
        char                    AcceptBuf_[BUFSIZE];

        char                    RecvBuf_[BUFSIZE];
        char                    SendBuf_[BUFSIZE];

        WSABUF                  wsaRecvBuf;
        WSABUF                  wsaSendBuf;

public:
        Proactor*               Proactor_;
        Acceptor*               Acceptor_;
        Disconnector*  Disconnector_;
        Sender*                 Sender_;
        Receiver*               Receiver_;

};
```

Acceptor

```
#pragma once

#include "Actor.h"

#include "TcpAct.h"
#include "TcpListenSocket.h"
#include "TcpSocket.h"

//---------------------------------------------------------
// 이름 :       Acceptor
//
// 역할 :       1)  ACT의 Complete()에서 ProcEvent() 콜
//              2)  ACT의 Error()에서 ProcError() 콜
//              3)  Register()을 통해 AcceptEx() 소켓풀에 넣음
//
// 기타 :       Actor 의 SubClass
//---------------------------------------------------------

class Acceptor : public Actor
{
public:
        Acceptor(){};

public:
        void ProcEvent( Act* act, DWORD bytes_transferred );
        void ProcError( Act* act, DWORD error );

        void Init( TcpListenSocket* tcplistensocket, Proactor* proactor );

        void Register( TcpSocket& acceptsocket );

public:
        TcpListenSocket*      TcpListenSocket_;
};
```

Disconenctor

```
#pragma once

#include "Actor.h"

#include "TcpSocket.h"

//-----------------------------------------------------------
// 이름 :       Disconnector
//
// 역할 :       1)  ACT의 Complete()에서 ProcEvent() 콜
//              2)  ACT의 Error()에서 ProcError() 콜
//
// 기타 :       Actor 의 SubClass
//-----------------------------------------------------------

class Disconnector : public Actor
{
public:
        Disconnector(){}

public:
        void ProcEvent( Act* act, DWORD bytes_transferred )
        {
                TcpAct& tcpact = *dynamic_cast<TcpAct*>(act);

                assert(tcpact.TcpSocket_);
                TcpSocket& tcpsocket = *tcpact.TcpSocket_;

                //printf("...Disconnector s(%d)\n", tcpsocket.GetSocket());

                assert(tcpsocket.Acceptor_);
```

```
                // 소켓 재사용
                tcpsocket.Reuse();
        }

        void ProcError( Act* act, DWORD error )
        {
                assert(dynamic_cast<TcpAct*>(act));

                TcpAct& tcpact = *dynamic_cast<TcpAct*>(act);

                assert(tcpact.TcpSocket_);

                TcpSocket& tcpsocket = *tcpact.TcpSocket_;

        }

        void Init( Proactor* proactor )
        {
                Proactor_ = proactor;
        }
};
```

Sender

```
#pragma once

#include "Actor.h"

#include "TcpAct.h"
#include "TcpSocket.h"

//-------------------------------------------------------
// 이름 :       Sender
//
```

```
// 역할 :       1)  ACT의 Complete()에서 ProcEvent() 콜
//              2)  ACT의 Error()에서 ProcError() 콜
//
// 기타 :       Actor 의 SubClass
//------------------------------------------------------

class Sender : public Actor
{
public:
        Sender(){};

public:
        void ProcEvent( Act* act, DWORD bytes_transferred )
        {
                assert(dynamic_cast<TcpAct*>(act));

                TcpAct& tcpact = *dynamic_cast<TcpAct*>(act);

                assert(tcpact.TcpSocket_);

                TcpSocket& tcpsocket = *tcpact.TcpSocket_;
        }

        void ProcError( Act* act, DWORD error )
        {
                assert(dynamic_cast<TcpAct*>(act));

                TcpAct& tcpact = *dynamic_cast<TcpAct*>(act);

                assert(tcpact.TcpSocket_);

                TcpSocket& tcpsocket = *tcpact.TcpSocket_;
        }

        void Init( Proactor* proactor )
```

```
	{
		Proactor_ = proactor;
	}
};
```

Receiver

```
#pragma once

#include "Actor.h"
#include "TcpAct.h"
#include "TcpSocket.h"

//-------------------------------------------------------
// 이름 :	Receiver.h
//
// 역할 :	1)  ACT의 Complete()에서 ProcEvent() 콜
//		2)  ACT의 Error()에서 ProcError() 콜
//
// 기타 :	Actor 의 SubClass
//-------------------------------------------------------

class Receiver : public Actor
{
public:
	Receiver(){};

public:
	void ProcEvent( Act* act, DWORD bytes_transferred )
	{
		assert(dynamic_cast<TcpAct*>(act));

		TcpAct& tcpact = *dynamic_cast<TcpAct*>(act);

		assert(tcpact.TcpSocket_);
```

```
        TcpSocket& tcpsocket = *tcpact.TcpSocket_;


        // passive 연결끊김
        if( bytes_transferred == 0 )
        {
                tcpsocket.Disconnect();
        }
        else
        {
                //printf("받은 값 = %s\n", tcpsocket.RecvBuf_);

                // Async Recv 포스트
                tcpsocket.Recv();

                //받은만큼 에코
         tcpsocket.Send((BYTE*)tcpsocket.RecvBuf_,bytes_transferred);
        }

}

void ProcError( Act* act, DWORD error )
{
        assert(dynamic_cast<TcpAct*>(act));

        TcpAct& tcpact = *dynamic_cast<TcpAct*>(act);

        assert(tcpact.TcpSocket_);

        TcpSocket& tcpsocket = *tcpact.TcpSocket_;

        tcpsocket.Disconnect();

}
```

```
        void Init( Proactor* proactor )
        {
                Proactor_ = proactor;
        }

};
```