

IOCP서버 구현

개요

게임 서버의 기반이 되는 **IOCP(I/O Completion Port) 기반 비동기 네트워크 서버 코어**를 직접 설계·구현한다. 멀티스레드 워커 풀과 세션 풀(Session Pool)을 통해 효율적이고 안정적인 연결/패킷 처리를 목표로 한다.

목적

- **IOCP 동작 원리 학습과 적용**: Overlapped I/O, Completion Key/Packet 흐름을 코드 수준에서 이해·구현
- **효율적인 서버 코어 개발**: 잠금 경합 최소화, 메모리 재사용(세션/버퍼 풀), 예측 가능한 지연 시간
- **확장 가능한 기반 마련**: 차후 Gate/Game/GameDB 구조로 확장 가능한 공통 네트워크 코어 확보

설계

1) 세션 풀 & 세션 매니저

- 오브젝트 풀링을 통해 `SessionManager` 가 세션을 미리 생성해 관리, `GetEmptySession/Release` 로 최소 비용 할당, 반납.
- 단일 소유권 보장을 위해 싱글톤으로 구성.
- `PostAccept` 시점에만 `pop`, 종료 시 `Release` 후 `push`. 멀티스레드에서도 세션 객체의 생명주기 명확화.

2) IOCP 워커 스레드

- `IoCPCore.Initialize()` 로 IOCP 핸들 생성 → `Run()` 에서 CPU×2 개 워커 스레드 기동.
- `GetQueuedCompletionStatus` 로 완료 이벤트를 받는 **중앙 루프** 운용.

3) 완료 이벤트 분기(WorkerLoop)

- **ACCEPT**
 - `AcceptEx` 완료 → `SO_UPDATE_ACCEPT_CONTEXT` 설정 → 클라 소켓 IOCP 등록 → 세션 초기화 후 첫 `WSARecv` 게시.
 - 다음 연결 대기 위해 즉시 다음 `AcceptEx` 재게시.
- **RECV**
 - 0바이트/오류 시 세션 종료 경로 진입.
 - 바이트 수신 시 수신 링버퍼에 기록 → 패킷 조립(Assembler) → 디스패치.

- `OnRecvCompleted` 는 세션 타입별(클라/게이트/게임)로 오버라이드하여, 검증/라우팅 정책을 분리.
- **SEND**
 - **부분 전송** 고려: 송신 큐에서 조각 단위 전송, 완료 시 pop 후 다음 조각 또는 idle 전환.

4) 패킷 등록 & 디스패치

- 등록: `PacketDispatcher` 에 `REGISTER_HANDLER(ID, Func)` 로 핸들러 정적 등록.
- 검증: 디스패치 전 **헤더 유효성(size 최소/최대, ID 화이트리스트)** 확인.

5) 수신 버퍼(링버퍼) & 패킷 조립

- TCP는 **바이트 스트림**이라 순서는 보장되지만 메시지 경계는 보존되지 않음. 잘려 오거나(분할) 붙어 올 수(결합) 있으므로 **조립** 단계가 필수.
1. 링버퍼에 write
 2. 저장된 바이트가 헤더보다 작으면 보류
 3. `size` 범위 검사 후 전체 길이가 도달하면 1패킷 추출
 4. 남은 바이트는 다음 패킷 조립에 사용

6) 전송 경로(Send)

- 큐잉: `SendPacket` 은 송신 큐에 넣고, **큐가 비어있을 때만** `WSASend` 를 게시(경합 감소).
- 완료 처리: `OnSendCompleted` 에서 현재 조각 pop → 남으면 즉시 다음 `WSASend` , 없으면 대기.

7) 종료/예외 처리(개요)

- 새 Accept 취소, 워커에 종료 키 Post, 워커 join 후 IOCP 핸들 Close.
- **프로토콜 위반**: 비정상 `size/ID` , 과도한 QPS 등은 **즉시 Close + 로그**.
- **오류 분류**: 소켓 오류(재시도 없음) / 일시 오류(재시도) / 프로토콜 위반(차단).

8) 포트 설정

- ini 파일을 통해 설정하도록 설계

주요 구현 내용

세션 매니저

`SessionManager` 에 세션 타입별 `Factory` 를 등록하고 초기 풀을 설정합니다. 각 세션별로 팩토리를 미리 설정해두고 Accept시 타입별 풀에서 해당하는 세션을 할당하고 세션 종료 시 Release하여 반납합니다

다. 풀에 세션이 없을 경우 해당 타입의 Factory를 불러와서 즉시 생성합니다.

이 구조로 타입별 생성 로직을 분리해 유지보수성과 안전성을 확보했습니다. 또한 오브젝트 풀링을 사용하여 **new/delete** 없이 즉시 할당(pop)·반납(push)하며, 지연과 메모리 파편화를 줄였습니다.

```
// 1. Winsock 초기화
WSADATA wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
{
    ERROR_LOG("WSAStartup 실패");
    return -1;
}

//2. SessionManager에 Session 생성 로직 주입
//게이트에서는 클라이언트 세션과 서버 세션을 관리합니다.
SessionManager::GetInstance().RegistFactory(SessionType::CLIENT, []() {
    return new ClientSession();
});
SessionManager::GetInstance().RegistFactory(SessionType::GAME, []() {
    return new ServerSession();
});
```

```
void SessionManager::RegistFactory(SessionType _eType, SessionFactory _factory,
INT32 _i32PoolSize)
{
    // 일단 락 걸고
    std::lock_guard<std::mutex> guard(m_SessionLock);

    // 팩토리 설정
    m_Factories[_eType] = _factory;

    // 세션 타입이 등록되어 있지 않으면 새로 생성
    if (m_SessionPool.find(_eType) == m_SessionPool.end())
    {
        for (int i = 0; i < _i32PoolSize; i++)
        {
            m_SessionPool[_eType].push(m_Factories[_eType]());
        }
    }
}
```

워커 루프

IOCP 큐에서 완료 이벤트를 꺼내 `IoCpContext.eOperation` 기준으로 `ACCEPT/RECV/SEND` 를 세션 메서드에 디스패치하여 처리하고 처리한 컨텍스트를 즉시 해제하도록 했습니다.

TCP는 바이트 스트림이라 메시지 경계가 깨집니다. 그래서 수신 바이트를 링버퍼에 쌓아 두고, 수신버퍼에서 패킷 아이디를 확인하여 Peek→검증→완성되면 Read/Remove 순으로 패킷 단위로 복원합니다.

송신은 Queue에서 하나씩 전송하되 부분 전송은 같은 버퍼의 잔여 구간을 즉시 재전송합니다.

각 이벤트를 처리하고 나면 작업이 끝난 컨텍스트는 즉시 해제하여 메모리 누수를 방지합니다.

```
...
enum class IoCpOperation
{
    ACCEPT,    // 연결 수락
    RECV,      // 데이터 수신
    SEND       // 데이터 송신
};

...

struct PacketHeader
{
    UINT16 size;
    UINT16 id;
};

...

void IoCpCore::WorkerLoop()
{
    while (m_isRunning)
    {
        ...

        // IoCpContext로 캐스팅 (OVERLAPPED 확장 구조)
        IoCpContext* pContext = reinterpret_cast<IoCpContext*>(overlapped);

        // 작업 종류에 따라 분기
        switch (pContext->eOperation)
        {
            case IoCpOperation::RECV:
                pContext->pSession->OnRecvCompleted(pContext, bytesTransferred);
                break;
            case IoCpOperation::SEND:
```

```

        pContext->pSession->OnSendCompleted(pContext, bytesTransferred);
        break;
    case IoCpOperation::ACCEPT:
        pContext->pSession->OnAcceptCompleted(pContext);
        break;
    default:
        break;
    }

    delete pContext;
}
}

```

- **ACCEPT** → `Session::OnAcceptCompleted`

클라 소켓 IOCP 등록 → 첫 `Recv()` 게시 → 다음 Accept 재게시.

```

void Session::OnAcceptCompleted(IoCpContext* _pContext)
{
    NetworkManager& networkManager = NetworkManager::GetInstance();
    SOCKET listenSock = m_Socket;
    // SO_UPDATE_ACCEPT_CONTEXT 설정하는 이유
    // - 커널에게 이 소켓은 이 리스 소켓을 통해 accept된거라고 알려주는 역할
    // - 그래야 커널이 해당 소켓 리소스를 정리할 타이밍을 제대로 계산
    INT32 i32OptResult = setsockopt(m_Socket, SOL_SOCKET, SO_UPDATE_ACCEPT
_CONTEXT, (char*)&listenSock, sizeof(SOCKET));
    SessionManager& sessionManager = SessionManager::GetInstance();
    if (i32OptResult == SOCKET_ERROR)
    {
        ERROR_LOG("setsockopt SO_UPDATE_ACCEPT_CONTEXT 실패 : " << WSAGet
LastError());

        closesocket(m_Socket);
        sessionManager.Release(m_eSessionType, this);
        delete _pContext;
        return;
    }

    IoCpCore& iocpCore = IoCpCore::GetInstance();
    if (!iocpCore.RegisterSocket(m_Socket, reinterpret_cast<ULONG_PTR>(this)))
    {

```

```

        ERROR_LOG("소켓 IOCP 등록 실패");
        closesocket(m_Socket);
        sessionManager.Release(m_eSessionType, this);
        delete _pContext;
        return;
    }
    LOG("클라이언트 접속 완료");

    if (!Recv())
    {
        ERROR_LOG("WSARecv 실패");
    }
    networkManager.AcceptListener(m_eSessionType);
}

```

- **RECV** → 수신된 데이터 길이만큼 RecvBuffer에 담아두고 패킷이 완성됐는지 체크하여 완성된 패킷이 있으면 해당 버퍼에 등록된 핸들러를 실행합니다.

```

void ClientSession::OnRecvCompleted(IocpContext* _pContext, DWORD _dwRecvLen)
{
    // 1. 수신 길이 0이면 클라이언트가 종료한 것
    if (_dwRecvLen == 0)
    {
        // TODO : 소켓 닫기, 세션 해제 필요 - Disconnect함수에서 구현
        Disconnect(); // 연결 종료
        return;
    }

    // 2. 수신된 데이터 길이만큼 저장
    m_RecvBuffer.Write(_pContext->tWsaBuf.buf, _dwRecvLen);

    // 3. 패킷 완성 여부 확인
    while (true)
    {
        // 3-1. 최소한의 데이터 -> 헤더 확인
        PacketHeader header;
        m_RecvBuffer.Peek(reinterpret_cast<char*>(&header), sizeof(PacketHeader));
        // 패킷 사이즈 구하기
        UINT16 ui16PacketSize = header.size;
        // 사이즈 확인
        if (m_RecvBuffer.GetStoredSize() < ui16PacketSize)

```

```

        break;

// 3-2. 버퍼에서 꺼내기
char cPacketBuffer[MAX_RECV_BUFFER_SIZE];
memset(cPacketBuffer, 0x00, sizeof(cPacketBuffer));
// 참조값 전달을 위함
char* pReadPtr = cPacketBuffer;
INT32 i32Len = static_cast<INT32>(ui16PacketSize);
if (!m_RecvBuffer.Read(pReadPtr, i32Len))
{
    ERROR_LOG("패킷 Read 실패");
    break;
}

// 패킷 처리
PacketDispatcher& dispatcher = PacketDispatcher::GetInstance();
dispatcher.Dispatch(this, pReadPtr, ui16PacketSize);

}

// 3. 다시 Recv() 호출
Recv();

}

```

- **SEND** → 전송 완료 시 큐에서 메시지를 pop하고 남은 데이터가 있으면 즉시 다음 WSASend를 이어 보낸다

```

void Session::OnSendCompleted(lopContext* _pContext, DWORD _dwSendLen)
{
    if (_dwSendLen == 0)
    {
        // 상대방이 소켓을 종료함
        Disconnect();
    }
    else
    {
        std::lock_guard<std::mutex> guard(m_SendLock);
        if (!m_SendQueue.empty())
        {
            m_SendQueue.pop(); // 현재 메세지 제거
        }
    }
}

```

```

    }

    // 다음 데이터 있는지 체크
    if (!m_SendQueue.empty())
    {
        // 있으면 send 호출
        auto& next = m_SendQueue.front();
        Send(next.data(), static_cast<INT32>(next.size()));
        return;
    }

    // 없으면 풀어주기
    m_bIsSending = false;

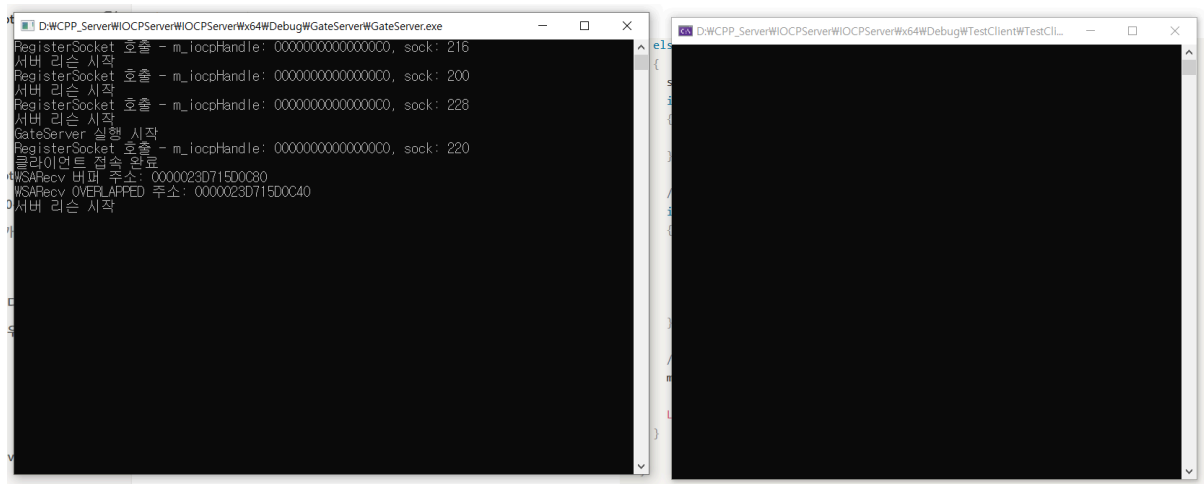
    LOG("Send 완료: " << _dwSendLen << " 바이트 전송됨");
}

}

```

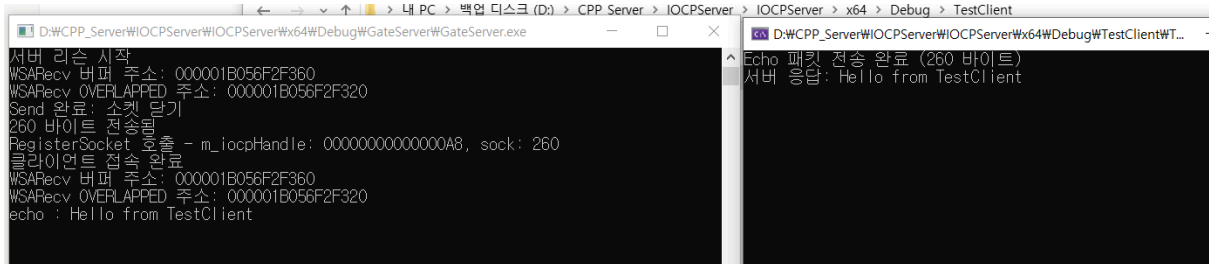
실습

1. 테스트 클라이언트로 게이트 서버에 접속 → **Accept** 에 성공하여 **'클라이언트 접속 완료'** 메시지가 뜨는 것을 확인할 수 있습니다.
또한 클라이언트로부터 메시지를 받을 준비를 마치고 **'서버 리슨 시작'** 메시지가 뜨는 것을 확인 할 수 있습니다.



2. 테스트 클라이언트에서 테스트용 Echo 패킷을 전송 후 서버에서 Echo 패킷 내용을 확인 할 수 있습니다.

또한 서버로부터 Echo 패킷을 받아서 클라이언트에서도 메시지가 출력 되는것을 확인 할 수 있습니다.



```
D:\CPP_Server\IOCPServer\IOCPServer\Debug\GateServer\GateServer.exe
서버 리슨 시작
WSARecv 버퍼 주소: 000001B056F2F360
WSARecv OVERLAPPED 주소: 000001B056F2F320
Send 완료: 소켓 닫기
260 바이트 전송됨
RegisterSocket 호출 - m_ioctlHandle: 00000000000000A8, sock: 260
클라이언트 접속 완료
WSARecv 버퍼 주소: 000001B056F2F360
WSARecv OVERLAPPED 주소: 000001B056F2F320
echo : Hello from TestClient

D:\CPP_Server\IOCPServer\IOCPServer\Debug\TestClient\WT...
Echo 패킷 전송 완료 (260 바이트)
서버 응답: Hello from TestClient
```

결론

워커 루프에서 **ACCEPT/RECV/SEND** 완료 이벤트를 안정적으로 디스패치하고, 컨텍스트 수명(할당·해제)을 일관되게 정리하여 IOCP 코어를 완성했습니다.

AcceptEx 사전 Post와 세션 오브젝트 풀 및 타입별 팩토리로 신규 연결 시 **new/delete** 없이 할당/반납하여 메모리를 효율적으로 관리했습니다.

링버퍼 기반 RecvBuffer로 TCP 스트림을 **[size][id][payload]** 단위로 안전하게 재조립하고, 패킷 디스패처를 통해 도메인 핸들러로 분리했습니다.

IOCP 워커 루프·오브젝트 풀·프레이밍/송신 상태머신의 세 축을 정확히 세워, 안정적 네트워크 코어를 만들었습니다. 다음으로는 해당 코어를 기반으로 실제 mmo rpg 서버처럼 게이트, 게임, 디비 서버를 연결하는 프로젝트를 만들려합니다.