

CarrierStrike

파일 설명

```
1  IOCP
2    - cpp
3      - iocp.cpp
4      - iocpClient.cpp
5      - iocpServer.cpp
6      └ session.cpp
7    - header
8      - iocp.h // 라이브러리의 본체
9      - iocpClient.h
10     - iocpServer.h
11     └ session.h // 연결된 클라이언트 소켓 관리
12   └ utility
13     - cpp
14       - gameObjectManager.cpp
15       - inputMemoryStream.cpp
16       - memoryStream.cpp
17       - messageManager.cpp
18       - objectPool.cpp
19       └ outputMemoryStream.cpp
20     └ header
21       - baseClass.h // 모든 직렬화 클래스가 상속해야하는 기본 클래스
22       - baseMessage.h // 모든 메시지가 상속해야하는 기본 메시지 클래스
23       - gameObject.h // 사용될 게임 오브젝트가 상속해야하는 기본 오브젝트 클래스
24       - gameObjectManager.h // 현재 게임 오브젝트의 상태를 관리
25       - inputMemoryStream.h
26       - memoryStream.h // 메모리 직렬화, 역직렬화를 수행
27       - messageManager.h // 직렬화, 원격 프로시저 호출, 리플리케이션
28       - objectPool.h // 오브젝트풀
29       └ outputMemoryStream.h
```

iocp.h

- 라이브러리의 본체 클래스로, 서버 로직이 구동됩니다.
 - iocpServer
 - iocpClient

session.h

- 연결된 클라이언트 소켓을 관리하고, 해당 소켓과의 통신용 인터페이스를 제공합니다.

Utility

baseClass.h

- 모든 직렬화 클래스가 상속 해야 하는 기본 클래스입니다. 직렬화에 관련된 여러 기능들이 순수 가상함수로 정의 되어 있고, 상속받은 클래스는 매크로를 통해 손쉽게 가상함수를 오버라이딩 할 수 있습니다.

baseMessage.h

- 모든 메시지 클래스가 상속 해야 하는 기본 메시지 클래스입니다. 메시지를 식별할 수 있는 식별자를 정의 할 수 있습니다.

gameObject.h

- 사용될 게임 오브젝트가 상속해야하는 기본 오브젝트 클래스입니다. 오브젝트를 식별할 수 있는 식별자와 팩토리 패턴으로 자기 자신을 생성할 수 있습니다.

CarrierStrike

파일 설명

```
1  IOCP
2    - cpp
3      - iocp.cpp
4      - iocpClient.cpp
5      - iocpServer.cpp
6      └ session.cpp
7    - header
8      - iocp.h // 라이브러리의 본체
9      - iocpClient.h
10     - iocpServer.h
11     └ session.h // 연결된 클라이언트 소켓 관리
12   └ utility
13     - cpp
14       - gameObjectManager.cpp
15       - inputMemoryStream.cpp
16       - memoryStream.cpp
17       - messageManager.cpp
18       - objectPool.cpp
19       └ outputMemoryStream.cpp
20     └ header
21       - baseClass.h // 모든 직렬화 클래스가 상속해야하는 기본 클래스
22       - baseMessage.h // 모든 메시지가 상속해야하는 기본 메시지 클래스
23       - gameObject.h // 사용될 게임 오브젝트가 상속해야하는 기본 오브젝트 클래스
24       - gameObjectManager.h // 현재 게임 오브젝트의 상태를 관리
25       - inputMemoryStream.h
26       - memoryStream.h // 메모리 직렬화, 역직렬화를 수행
27       - messageManager.h // 직렬화, 원격 프로시저 호출, 리플리케이션
28       - objectPool.h // 오브젝트풀
29     └ outputMemoryStream.h
```

Utility

gameObjectManager.h

- 사용할 오브젝트 클래스를 레지스트리에 등록할 수 있습니다.
- 등록된 오브젝트들에 대해서 생성, 갱신, 삭제 등의 관리를 제공합니다.
- 새로 생성되거나 갱신된 오브젝트는 객체 상태 전파를 위한 리플리케이션 대기 큐에 삽입됩니다.

messageManager.h

- 메시지 송수신을 전반적으로 관리합니다.
- 메시지의 직렬화, 역직렬화를 수행하는 인터페이스를 제공합니다.
- 특정 메시지와 함께 동작할 함수를 등록하면, 이후 해당 메시지가 수신되었을 때 자동으로 함수를 호출하는 원격 프로시저 호출(RPC) 기능을 수행합니다.
- gameObjectManager의 리플리케이션 대기큐를 모니터링해 변경점을 감지한 경우 자동으로 관련된 클라이언트에게 전송합니다.

memoryStream.h

- 클래스 직렬화를 위한 도구입니다.

IOCP_lib

IOCP_lib

개인 프로젝트 | 2025.06 ~ 2025.08

IOCP_lib

범용 IOCP 서버 라이브러리

기술 스택

C/C++, socket

Github

<https://github.com/winrinseo/IOCP>

- 기존에 구현했던 iocp 서버 경험을 바탕으로, 프로젝트 범용적인 라이브러리 개발
- 기존 프로젝트보다 스레드 안정성 개선
- 서버-클라이언트간 통신을 위한 클래스 직렬화 시스템 구현
 - 동적 배열이나 클래스 배열같이 복잡한 클래스도 무리없이 직렬화 할 수 있도록 일반화
 - 클래스 직렬화 시스템을 기반으로 원격 프로시저 호출 기능 구현
 - 메세지를 수신하면 역직렬화 후 메세지의 내용을 인자삼아 등록된 함수를 실행
 - 객체 리플리케이션 기능 구현
 - 기존 메세지 시스템에 기반한 동작이지만 메세지 시스템 내에서 클래스 배열은 동일한 객체만 사용될 수 있다는 문제 발생
 - Wrapper 클래스를 만들어 실제 오브젝트 객체를 감싸는 방식으로 문제를 해결함

개요

예전에 수행했던 SpaceQuickDelete라는 프로젝트의 서버는 너무나 문제점이 많은 서버였습니다.

- 스레드 세이프 하지않아 경쟁 상태를 일으켜 잠재적인 문제점이 많았습니다.
- 직렬화/역직렬화는 클래스 멤버 변수로 클래스나 클래스 배열을 처리하지 못해 복잡한 객체를 직렬화 하지 못했습니다.
- 원격 프로시저 호출 시스템이 구현되지 않아 메세지 수신 시 필요한 동작을 IOCP WorkerThread 내부에서 정의 해야 했기 때문에 코드가 매우 길어져 가독성과 유지보수성에 문제가 있었습니다.
- 같은 문제로 객체 리플리케이션 시스템이 구현되지 않아 객체 상태를 동기화 시키는 것에 어려움이 있었습니다.
- 프로젝트에 너무 종속적이라 범용성이 매우 떨어집니다.

시간에 매우 쫓겨 만들었지만 이처럼 매우 많은 문제점을 가지고 있었기 때문에 아쉬움이 많이 남았던 프로젝트였습니다.

그런 이유로, 문제점을 개선하고 새로 다시 한번 구현해 보았습니다. 처음부터 새로 구현하며 추가된 기능이 매우 많고, 라이브러리를 사용하는 사용자 입장에서 큰 편의성을 느끼도록 설계했습니다.

아래는 설명할 클래스의 목차입니다. 해당 클래스에서 핵심적인 함수를 설명하겠습니다.

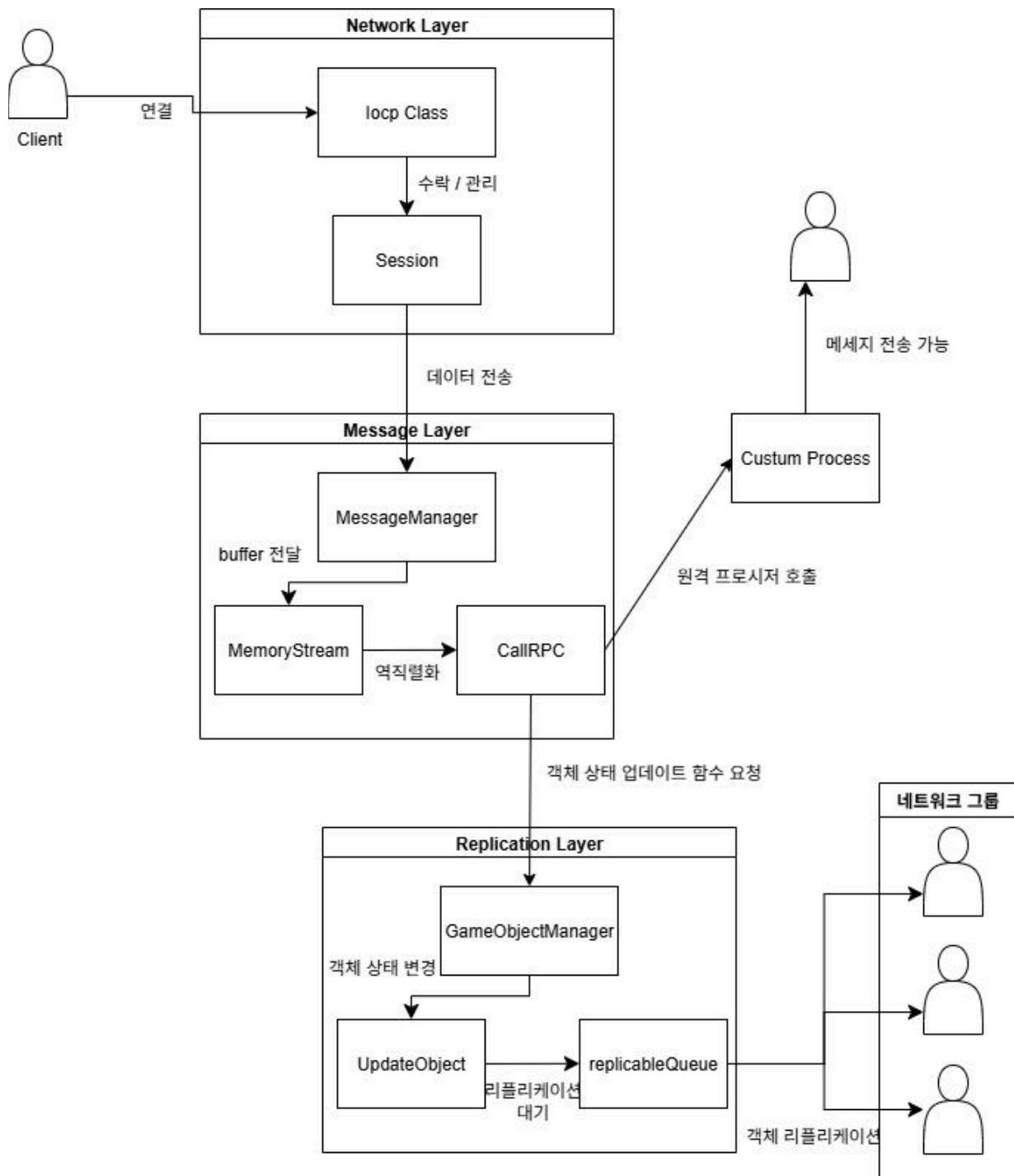
1. IoCP
2. Session
3. BaseClass, GameObject, BaseMessage
4. MessageManager
5. MemoryStream
6. GameObjectManager

IOCP_lib

Architecture

서버 아키텍쳐

클라이언트가 서버에 연결하고 메세지를 전송했을 때의 동작입니다.



IOCP_lib

locp

라이브러리의 본체 클래스로, 파생 클래스를 통해 서버와 클라이언트의 역할을 나눠 각 역할을 수행합니다.

여러가지 유틸리티 클래스를 사용해 메세지 직렬화와 원격 프로시저 호출, 클라이언트 연결 관리 등 프로세스의 본체 역할을 수행합니다.

1. PostAccept

기존엔 Listen socket으로 연결 요청한 클라이언트를 수락하기 위해선 블로킹 상태로 Thread를 하나 점유해야 했습니다. 이를 개선하기 위해 AcceptEX를 활용해 비동기로 IOCP에서 함께 처리할 수 있도록 구현해 Thread 효율을 향상했습니다. PostAccept 함수를 실행하면 클라이언트 접속을 비동기로 대기합니다. 클라이언트가 연결에 성공할 때마다 PostAccept 함수를 재호출해 다시금 대기할 수 있도록 합니다.

2. WorkerThread

IOCP에서 입출력 완료를 처리하는 곳입니다. 처리할 동작은 RECV, SEND, ACCEPT, CONNECT로 4가지지만 가장 중요한 부분은 RECV 동작을 처리하는 부분입니다. 해당 동작은 수신된 메세지를 해석하고 원격 프로시저 호출을 수행합니다. 각각의 동작은 모두 추상화 되어있기 때문에 실제로 처리하는 부분은 매우 간결합니다.

```
1 //ioctp.cpp
2 //void Ioctp::WorkerThread()의 일부.
3
4
5 if (context->operation == OperationType::RECV) { //수신 완료일때
6
7     // 1. 해당 세션은 재수신 준비
8     if(this->connects_.find(sessionId) != this->connects_.end())
9         connects_[sessionId]->Receive();
10
11    // 2. 역직렬화 messageManager.Dispatch는 메세지를 해석해 그에 맞는 객체를 만들어 반환한다.
12    BaseMessage * msg =
13        messageManager.Dispatch(context->buffer, bytesTransferred);
14
15    // 3. 원격 프로시저 호출
16    messageManager.CallRPC(msg);
17
18    // 4. 재귀적으로 delete하거나 추후 스마트 포인터로 변경
19    delete msg;
20
21    // 5. 수신 완료시 수행 할 작업 (외부에서 등록 가능)
22    OnReceiveCompletion(sessionId, context->buffer, bytesTransferred);
23 }
24 }
```

IOCP_lib

Session

클라이언트와의 연결과 비동기 통신을 관리하는 객체입니다.

1. Receive

해당 세션의 수신을 미리 대기합니다. PostAccept와 마찬가지로 해당 세션에서 수신을 받았을 시 재호출해 다시금 대기합니다.

2. Send

해당 세션에 메세지를 발신합니다. MessageManager가 Serialize 함수를 호출해 메세지를 직렬화한 후 Send 함수를 통해 메세지를 발신할 수 있습니다.

```
1 // session.h
2 // 연결된 개체와의 통신을 담당하는 세션 클래스
3 class Session {
4 public:
5     Session(SOCKET socket);
6     ~Session();
7
8     // 소켓 핸들 및 Overlapped 포인터 반환
9     SOCKET GetSocket() const { return socket_; }
10
11
12     //통신용 함수
13     void Receive(); // 데이터 수신 대기
14     bool Send(const char * data , int len); //해당 클라이언트에 메세지 송신
15
16
17 private:
18     SOCKET socket_; // 클라이언트 소켓
19
20 };
```

IOCP_lib

BaseClass, GameObject, BaseMessage

이 프로젝트에서 가장 핵심적인 부분인 기본 직렬화 객체들입니다. 이 객체들은 각각의 기본 단위로, 이 프로젝트에서 사용될 객체들은 용도에 맞게 위의 객체중 하나를 상속받아 사용합니다.

기본 직렬화 객체에서 선언되어있는 순수 가상 함수를 오버라이딩하기 때문에, 통신에 필요한 모든 일반화된 기능은 기본 객체로 캐스팅되어 사용됩니다.

1. BaseClass , BaseMessage

객체를 직렬화하기 위해서는 객체의 구조를 알 수 있도록 리플렉션 시스템이 존재해야 합니다. BaseClass는 이를 위한 리플렉션 시스템의 설계도입니다. 따라서 GameObject나 BaseMessage조차도 반드시 BaseClass를 상속받아야만 사용할 수 있습니다. BaseMessage는 메세지 식별자와 원격 프로시저 호출을 위한 반환 함수밖에 없기 때문에 크게 다르지 않기 때문에 함께 설명하겠습니다.

```
1 //utility/header/BaseMessage.h
2 class Command : public BaseMessage{
3 public:
4     /*
5      *    실질적으로 클라이언트와 통신하기위한 기본 단위임 .
6      *    기본적으로 베이스 클래스를 상속받지만 클라이언트와 통신하기 위한 몇가지 함수가 추가되어있음 .
7
8     */
9     uint32_t num;
10    Player * me;
11    std::vector<Player*> cmdDeck; // 클래스는 항상 포인터형으로 선언
12
13 MESSAGE_IDENTIFICATION(20,Command)
14 // Vector<MemberVariable> DataType을 생성하는 매크로
15 // 일반적으로 변수명과 타입, 오프셋을 입력받지만 클래스 타입인 경우 자기 자신의 객체를 반환하는 함수를
16 // 꼭 추가 해야한다.
17 REFLECTABLE(Command,
18     MemberVariable("num", Type::Int32, OffsetOf(Command, num)),
19     // 클래스 포인터 타입일 경우에는 반환 함수를 꼭 추가해야함
20     MemberVariable("me", Type::Class, OffsetOf(Command, me) , Type::Class ,
21                     [] (BaseClass * cls){return new Player();}),
22     //
23     MemberVariable("cmdDeck", Type::Vector, OffsetOf(Command, cmdDeck) ,
24                     Type::Class , [] (BaseClass * cls){return new Player();})
25 )
26 };
```

MESSAGE_IDENTIFICATION 매크로는 BaseMessage 객체에 선언되어 있는 순수 가상 함수를 오버라이딩하고 식별자를 설정하는 매크로입니다.

REFLECTABLE 매크로는 BaseClass 객체에 선언되어있는 리플렉션 관련 자료구조를 초기화하는 매크로입니다. 기본타입의 변수(정수, 소수, 문자열)는 오프셋 까지만 입력해주면 되지만 Class의 경우엔 동일한 객체를 반환하는 함수를 추가해줘야 합니다. 반환 함수에서 BaseClass의 포인터를 받는 이유는 이후 설명할 GameObject 때문입니다.

IOCP_lib

2. GameObject

우리 게임에서 사용할 오브젝트의 기본 단위입니다. 오브젝트의 상태만을 저장하는 것을 목표로 합니다. GameObjectManager에서 업데이트된 오브젝트를 확인해 실제 오브젝트의 상태를 변화시킬 수 있도록 설계했습니다.

```
1 // utility/herder/gameObject.h
2 class GameObject : public BaseClass{
3 public:
4
5     virtual uint32_t GetObjectId() const = 0;
6     virtual GameObject * CreateInstance() = 0;
7     virtual std::unique_ptr<GameObject> clone() const = 0; // 깊은 복사 후 반환
8     virtual void update(const GameObject& source) = 0; // 깊은 복사
9
10 };
11
12
13
14 class SampleObject : public GameObject{
15 public:
16     int32_t hp;
17     int32_t mp;
18     float x;
19     float y;
20     float z;
21
22     OBJECT_IDENIFICATION(202020 , SampleObject)
23
24     REFLECTABLE(SampleObject,
25         MemberVariable("hp" , Type::Int32 , OffsetOf(SampleObject , hp)),
26         MemberVariable("mp" , Type::Int32 , OffsetOf(SampleObject , mp)),
27         MemberVariable("x" , Type::Float , OffsetOf(SampleObject , x)),
28         MemberVariable("y" , Type::Float , OffsetOf(SampleObject , y)),
29         MemberVariable("z" , Type::Float , OffsetOf(SampleObject , z)))
30     )
31 }
```

OBJECT_IDENIFICATION 매크로는 역시 오브젝트 식별자와 함께 순수 가상함수들을 오버라이딩 하는 매크로입니다.

그 중 clone과 update가 깊은 복사를 수행하는 이유는, 이미 생성되어 있는 오브젝트의 주소값을 받아서 작업을 수행하고 있는 도중에 주소값이 바뀌게 되면 임계영역과 상관없이 Race condition이 발생하게 됩니다. 따라서 성능을 어느정도 희생하더라도 깊은 복사만 수행해 한번 생성된 객체는 주소값이 바뀌지 않도록 설계했습니다.

IOCP_lib

3. GameObjectWrapper

게임 오브젝트 객체를 감싸는 Wrapper입니다. 해당 클래스가 필요한 이유는 다음과 같습니다.

```
1 // utility/header/baseMessage.h
2 class ReplicationData : public BaseMessage{
3 public:
4     uint32_t sessionId;
5     uint32_t networkGroup;
6     std::vector<GameObject*> objList; // 실제 게임 오브젝트는 크기가 다른 별개의 클래스입니다.
7     //...중략
8 };
```

이처럼 리플리케이션할 객체를 모아 전송하는 메세지의 objList 벡터는 크기가 다른, 즉 서로 다른 객체를 담을 수 없습니다. 또한, 메세지를 역직렬화할 때 리스트의 크기를 먼저 맞춘 후 외부에서 객체를 미리 생성해줍니다.

이 방식은 리스트의 요소가 단일 클래스일 때만 가능합니다.

따라서 objList에는 단일 클래스인 GameObjectWrapper를 삽입합니다.

그리고 Wrapper 내부에서 실제 오브젝트를 저장하는 방식으로 기존의 메세지 시스템을 망가뜨리지 않고 서로 다른 객체들을 한번에 리플리케이션 할 수 있게 설계되었습니다.

```
1 // utility/header/baseMessage.h
2 class ReplicationData : public BaseMessage{
3     //...중략
4     std::vector<GameObjectWrapper*> objList;
5     //...중략
6     REFLECTABLE(ReplicationData,
7         //...중략
8         MemberVariable("objList", Type::Vector, OffsetOf(ReplicationData, objList) ,
9             Type::Class , [this](BaseClass* cls){
10                 return new GameObjectWrapper();
11             }),
12         )
13     };
```

Wrapper 클래스 내부에서 핵심적인건 다음과 같습니다.

```
1 // GameObjectWrapper 내부의 오브젝트 반환 함수입니다.
2 MemberVariable("obj" , Type::Class , OffsetOf(GameObjectWrapper , obj) , Type::Class,
3     [this](BaseClass * cls){
4         return GameObjectRegistry::Get()>GetGameObject(
5             ((GameObjectWrapper * )cls)>objectId);
6     }) ,
```

사실 어떤 객체가 사용될 것인지는 미리 알지 못합니다. 그렇기 때문에 앞서 설명드렸던 객체 반환 함수처럼 자기 자신을 반환하도록 작성할 수 없습니다. 따라서, 직전에 역직렬화된 Wrapper→objectId를 키로 GameObjectRegistry 팩토리를 통해 동적으로 해당 타입의 객체를 생성하여 다형성을 확보했습니다.

이렇게 일반화된 코드로 객체 관리와 리플리케이션에 필요한 도구들을 모두 구현하게 되었습니다.

IOCP_lib

MessageManager

메세지의 직렬화, 원격 프로시저 호출, 객체 리플리케이션을 수행하는 핵심적인 클래스입니다. 각 기능 역시 추상화되어있어, 서로 간섭하지 않고 동작합니다. 다만 register함수를 통해 사용할 메세지와 함수를 등록하는 것은 필수입니다. 각 기능에 필요한 함수가 여럿 구현되어 있기 때문에 크게 기능별로 설명하겠습니다.

1. 직렬화 / 역직렬화

MemoryStream 클래스를 활용해 객체를 메모리에 직렬화하거나, 메모리의 데이터를 객체로 역직렬화하는 역할을 합니다. 직렬화를 맡은 Serialize함수와 역직렬화 및 객체 반환을 맡은 Dispatch함수로 외부에 직렬화 인터페이스를 제공합니다. 다만 Dispatch함수는 register함수를 통해 메세지를 등록해야 사용할 수 있습니다.

```
1 //utility/cpp/messageManager.cpp
2 void MessageManager::Serialize(
3     BaseMessage * msg , char ** output_buffer, int * output_length){
4     // 객체 획득 (ObjectPool 클래스 내부에서 동기화 처리 되어있기 때문에 바로 가져와도 됨)
5     ObjectPool<OutputMemoryStream>::ObjectPtr
6         outMemoryStream = outMemoryStreamPool.get();
7     // 버퍼 준비
8     outMemoryStream->Prepare();
9     // 직렬화
10    outMemoryStream->SerializeMessage(msg);
11
12    // 직렬화 정보 반환
13    *output_buffer = (char *) outMemoryStream->GetBuffer();
14    *output_length = outMemoryStream->GetLength();
15 }
```

2. 원격 프로시저 호출 (RPC)

특정 메세지를 수신하면 자동으로 등록한 함수를 실행합니다.

```
1 //utility/cpp/messageManager.cpp
2 //원격 프로시저 호출
3 void MessageManager::CallRPC(BaseMessage * msg){
4     if(RPC.find(msg->GetId()) != RPC.end())
5         RPC[msg->GetId()](msg); // 메세지 식별자를 가져와 함수를 실행함. 직렬화된 메세지는 함수의 인자.
6 }
```

IOCP_lib

3. 리플리케이션

객체의 상태를 연결된 클라이언트에 전파하는 기능입니다. 해당 기능엔 네트워크 그룹이라는 개념이 있는데 이는 리플리케이션할 범위를 사용자가 설정하기 위함입니다. 가령 유저가 P2P그룹이나 동일한 맵에 있다고 가정하면 동일한 그룹인 경우에만 객체의 상태를 전달해주면 될 것입니다.

사용자가 네트워크 그룹의 개념을 설정하고 해당 그룹에 메세지를 발신할 수 있는 함수를 SetSendGroup함수를 통해 등록해주면 준비가 완료됩니다.

```
1 //utility/cpp/messageManager.cpp
2 void MessageManager::replicationThread(){
3     while(1){
4         uint32_t networkGroup;
5         // 없으면 생길때까지 블록
6         auto replication_data = gameObjectManager->PopReplication(&networkGroup);
7
8         std::unique_ptr<ReplicationData> data = std::make_unique<ReplicationData>();
9
10        data->sessionId = 0;
11        data->networkGroup = networkGroup;
12
13        while(!replication_data.empty()){
14
15            auto data_piece = std::move(replication_data.front());
16            replication_data.pop();
17
18            GameObjectWrapper * wrapper = new GameObjectWrapper();
19            wrapper->networkId = data_piece.first;
20            wrapper->objectId = data_piece.second->GetObjectId();
21
22            // 만약 삭제된 객체면
23            if(gameObjectManager->ObjectToAddress(wrapper->networkId) == nullptr)
24                wrapper->objectId = 0; //objectId는 0으로 바꿔 삭제되었음을 알림
25
26            // 메모리 누수 발생 지점(일단 테스트 하고 개선)
27            wrapper->obj = data_piece.second.release();
28            data->objList.push_back(wrapper);
29        }
30
31        // 직렬화
32        char * buffer;
33        int size;
34        Serialize((BaseMessage*)data.get() , &buffer , &size );
35        // 네트워크 그룹에 메세지 발신. SetSendGroup함수를 통해 등록
36        sendGroup(data->networkGroup , buffer , size);
37    }
38 }
```

먼저 GameObjectManager에서 변경된 객체가 없는지 확인합니다. 만약 없다면 변경점이 생길때까지 대기합니다.

변경점이 생겼다면 객체가 담긴 큐를 반환받습니다. 이 때 반환된 객체들은 모두 네트워크 그룹이 같습니다.

객체들을 Wrapper에 감싸 objList에 넣습니다.

모든 객체가 리스트에 담겼다면 직렬화 후, 네트워크 그룹에 메세지를 발신합니다.

IOCP_lib

MemoryStream

실제로 직렬화를 수행하는 클래스입니다. 직렬화와 역직렬화 각각 MemoryStream의 파생 클래스가 담당합니다.

실제로 대부분의 로직은 동일하고 가상함수인 `IsInput`과 가장 기본단위 `Serialize`를 오버라이딩해 각 기능을 구분합니다. 직렬화의 경우 인자로 들어온 객체를 메모리에 기록하고, 역직렬화의 경우 인자로 받은 객체에 메모리의 값을 입력해줍니다.

1. 클래스 직렬화 / 역직렬화

클래스 하나를 직렬화/역직렬화 하는 함수입니다. 동작의 차이를 살펴보면 멤버 변수가 사용자 정의 클래스인 경우, 직렬화일땐 그대로 메모리에 입력하지만, 역직렬화일땐 클래스를 동적할당해 해당 주소를 `classPtr`에 넣어줍니다.

```
1 //utility/cpp/memoryStream.cpp
2 // 클래스 하나를 직렬화 or 역직렬화
3 void MemoryStream::Serialize(BaseClass * data){
4     for(auto& mv : data->GetDataType()){
5         void * mvData = reinterpret_cast<char*>(data) + mv.GetOffset();
6         switch(mv.GetType()){
7             case Type::Int8:
8                 Serialize(mvData , sizeof(uint8_t));
9                 break;
10            //...중략
11            default: // 사용자 정의 클래스
12            {
13                //포인터 변수 mvData도 포인터 변수를 가지기 때문에 역참조를 위해 더블포인터 사용
14                BaseClass** classPtr = reinterpret_cast<BaseClass**>(mvData);
15                // 역직렬화에서는 정보를 담을 객체를 준비해주는 과정이 필요하다.
16                if(IsInput()){
17                    // 탑이 클래스 포인터형일 경우 (GameObject의 경우 wrapper의 주소를 보내줘야함.
18                    *classPtr = mv.CreateInstance(data);
19                }
20                Serialize(*classPtr);
21                break;
22            }
23        }
24    }
25}
26}
27}
```

벡터를 직렬화/역직렬화하는 과정도 위 함수와 다를것이 없습니다. 여기서는 크기가 가변적인 클래스에 깊게 처리되게 때문에 벡터의 크기가 먼저 입력되어야 한다는 차이입니다.

직렬화의 경우 크기를 먼저 메모리에 입력하고, 역직렬화의 경우 얻어낸 크기로 벡터의 크기를 변경합니다.

그 이후 위의 사용자 정의 클래스와 같이 각 요소에 객체를 생성해 넣어 정보를 받을 준비를 해줍니다.

IOCP_lib

GameObjectManager

등록된 게임 오브젝트를 생성하거나 관리하는 클래스입니다. 해당 클래스에는 크게 생성, 업데이트, 삭제 함수를 제공하는데 각 함수는 서버용과 클라이언트용으로 나뉘어있습니다.

서버용 함수는 실행하면 리플리케이션 대기큐에 변경사항을 입력합니다. 입력된 변경사항은 앞서 설명드린 MessageManager의 replicationThread에서 처리됩니다.

클라이언트용 함수는 서버로부터 ReplicationData 메세지를 수신받았을 때 실행됩니다. 변경사항을 업데이트 완료 큐에 입력해 사용자가 변경사항을 실제 게임에 적용할 수 있습니다.

1. 서버 전용 함수

해당 함수들을 통해 객체를 변화시키면 연결된 네트워크 그룹으로 리플리케이션이 가능합니다.

서버가 능동적으로 사용하거나 클라이언트로부터 사용 요청을 받아 사용할 수 있습니다.

```
1 // utility/cpp/gameObjectManager.cpp
2 // 네트워크 그룹에 오브젝트를 생성함
3 void GameObjectManager::CreateObject(uint32_t networkGroup, GameObject * obj){
4
5     uint32_t net_id;
6     {
7         std::lock_guard<std::mutex> lock(idMutex);
8         net_id = ++networkIdCounter;
9     }
10
11    // 네트워크 주소가 다르면 동일한 메모리를 참조할 일이 없기 때문에
12    // 네트워크 그룹이 같은 경우에만 락을 걸어준다.
13    std::lock_guard<std::mutex> lock(groupMutex[networkGroup]);
14
15    objectToAddress[net_id] = obj;
16    actingObject[networkGroup][net_id] = std::unique_ptr<GameObject>(obj);
17
18    // 리플리케이션 대기열에 등록
19    // obj의 clone() 가상 함수를 호출하여 실제 타입에 맞는 복사본을 생성
20    replicableQueue[networkGroup].push(std::make_pair(net_id, obj->clone()));
21    replicationQueue_cv.notify_one();
}
```

다른 서버용 함수 역시 비슷한 로직으로 동작합니다.

IOCP_lib

2. 클라이언트 전용 함수

서버로부터 ReplicationData메세지를 받았을 때 객체의 상태에 따라 자동으로 수행됩니다.

객체가 클라이언트에 없다면 생성, 있다면 업데이트, objectId가 0이라면 삭제를 수행합니다.

```
1 // cpp/iocp.cpp
2 // 필수적인 함수 등록
3 messageManager.regist(new ReplicationData() , [this](BaseMessage * msg){
4     ReplicationData * objectData = (ReplicationData*) msg;
5
6     GameObjectManager * gm = GameObjectManager::Get();
7
8     for(auto & it : objectData->objList){
9         uint32_t & networkId = it->networkId;
10        uint32_t & objectId = it->objectId;
11
12        if(objectId == 0){ // 오브젝트 아이디가 0으로 오면 삭제
13            gm->DeleteObjectFromOther(objectData->networkGroup , networkId);
14
15        }else if(gm->objectToAddress(networkId) == nullptr){ // 받는 쪽에서 없으면 생성
16            gm->CreateObjectFromOther(objectData->networkGroup, networkId, it->obj);
17
18        }else{ // 있으면 업데이트
19            gm->UpdateObjectFromOther(objectData->networkGroup , networkId , it->obj);
20        }
21    }
22});
```

클라이언트 전용 함수도 서버 전용 함수와 비슷하지만 리플리케이션 대기 큐가 아닌, 업데이트 완료 큐에 삽입됩니다.

사용자는 업데이트 완료 큐를 통해 객체의 변경된 상태를 알 수 있으며, 알아낸 변경 사항으로 실제 게임의 오브젝트를 변화시킬 수 있도록 설계되었습니다.

```
1 // utility/cpp/gameObjectManager.cpp
2 // 다른 클라이언트(서버)에 의해 객체가 생성됨
3 void GameObjectManager::CreateObjectFromOther(
4     uint32_t networkGroup , uint32_t networkId,GameObject * obj){
5
6    { // 범위가 너무 넓으면 싱글 스레드와 다를게 없음
7        std::lock_guard<std::mutex> lock(groupMutex[networkGroup]);
8        objectToAddress[networkId] = obj;
9        actingObject[networkGroup][networkId] = std::unique_ptr<GameObject>(obj);
10    }
11
12    // 업데이트 완료 대기큐에 등록
13    {
14        std::lock_guard<std::mutex> lock(updatedQueueMutex);
15        updatedObject.push(networkId);
16    }
17    updatedQueue_cv.notify_one();
}
```

IOCP_lib

3. 동시성 처리 모델

replicableQueue는 네트워크 그룹을 Key로 가지고, 큐를 Value로 가지는 해시맵입니다. 이러한 replicableQueue를 묶는 하나의 거대한 락은 서로 다른 네트워크 그룹을 처리하는 스레드 간의 불필요한 대기를 유발시켜 병렬성을 저해합니다. 이를 해결하기 위해 Fine grained locking 방식을 채택해 네트워크 그룹별로 개별적인 락을 걸었습니다. 해시맵의 특성상 Key가 다르면 서로 다른 메모리를 참조하기 때문에 Race Condition이 발생하지 않을 것입니다.

```
1 // utility/cpp/gameObjectManager.cpp
2 void GameObjectManager::UpdateObject(uint32_t networkGroup,
3                                     uint32_t networkId, GameObject * obj){
4     // 네트워크 그룹 개별 잠금
5     std::lock_guard<std::mutex> lock(groupMutex[networkGroup]);
6
7     //...중략
8 }
```

replicableQueue 전체를 잠궈야 할 때도 있습니다. 리플리케이션 데이터를 꺼내오는 PopReplication 함수에서는 먼저 replicationQueueMutex로 맵 전체의 구조를 보호하며 처리할 대상을 찾고, 대상이 정해지면 해당 그룹의 groupMutex를 잠근 뒤 replicableQueue의 해당 그룹을 안전하게 제거합니다. 이를 통해 락 경합을 최소화하고 시스템 처리량을 극대화했습니다. 락을 두개 사용하긴 하지만 replicationQueueMutex는 PopReplication 함수에서만 사용되므로 데드락의 조건인 원형 대기가 발생하지 않습니다.

```
1 // utility/cpp/gameObjectManager.cpp
2 // 임의의 네트워크 그룹 리플리케이션 큐를 반환
3 std::queue<std::pair<uint32_t, std::unique_ptr<GameObject>>>
4     GameObjectManager::PopReplication(uint32_t * networkGroup){
5
6     // 해시맵 전체 잠금
7     std::unique_lock<std::mutex> replication_lock(replicationQueueMutex);
8     // 비어있다면 대기
9     replicationQueue_cv.wait(replication_lock, [this]{return !replicableQueue.empty();});
10    // 반환할 큐
11    auto it = replicableQueue.begin();
12    // 사용중이라면 대기
13    std::lock_guard<std::mutex> group_lock(groupMutex[it->first]);
14    // replicableQueue의 networkGroup에 유일하게 접근하고 있는 상태
15    *networkGroup = it->first;
16    auto ret = std::move(it->second);
17    // 그룹 삭제
18    replicableQueue.erase(it);
19
20    return ret;
21 }
```

IOCP_lib

4. 리플리케이션 시연

간단히 문자열 오브젝트를 만들어 시연해 보았습니다.

서버에서 ChatMessage를 수신하면 이를 토대로 객체를 만듭니다.

해당 오브젝트를 생성하면 즉시 동일한 그룹에 객체를 리플리케이션합니다.

```
1 // main.cpp
2 iocp.RpcRegist(new ChatMessage() , [&](BaseMessage * msg){
3     ChatMessage * chat = (ChatMessage*) msg;
4     // 오브젝트
5     Chat * chatObject = new Chat();
6     chatObject->sessionId = chat->sessionId;
7     chatObject->chat = chat->chat;
8
9     std::cout<<chatObject->chat<"\n";
10    // 오브젝트 생성, 생성만 하더라도 동일한 네트워크 그룹엔 리플리케이션.
11    GameObjectManager::Get()->CreateObject(chat->networkGroup , chatObject);
12
13});
```

아래는 시연 내용입니다. 출력하는 스레드가 많아 결과가 조금 지저분하게 나왔지만 제대로 동일한 그룹에 객체를 생성하고 리플리케이션하는게 확인됩니다.

```
chatThread 시작
나의 세션 아이디 : 10038
나의 네트워크 그룹 : 20000
> 20000번 네트워크에서 채팅 객체 전송 합니다.
> 전송 완료 !!
> 전송 완료 !!
전송 완료 !!
> 10038> 전송 완료 !!
: > 전송 완료 !!
20000번
10038 : 네트워크에서
10038 : 채팅
10038 : 객체
10038 : 전송 합니다.

시스템 CPU 코어 수 : 16
생성할 스레드 개수 : 33
서버에 성공적으로 연결됨 !
서버 연결 완료 !!
20000
나의 세션 아이디 : 10039 chatThread 시작
전송 완료 !!

나의 네트워크 그룹 : 20000
> 10038 : 20000번
10038 : 네트워크에서
10038 : 채팅
10038 : 객체
10038 : 전송 합니다.

시스템 CPU 코어 수 : 16
생성할 스레드 개수 : 33
서버에 성공적으로 연결됨 !
서버 연결 완료 !!
20001
전송 완료 !!
chatThread 시작
나의 세션 아이디 : 10040
나의 네트워크 그룹 : 20001
> 20001번 네트워크에서, 객체, 전송
> 전송 완료 !!
10040 : 20001번, 네트워크에서, 객체, 전송
10041 : 채팅처럼, 보이지만, 객체, 생성, 후, 리플리케이션입니다.

시스템 CPU 코어 수 : 16
생성할 스레드 개수 : 33
서버에 성공적으로 연결됨 !
서버 연결 완료 !!
20001
전송 완료 !!
나의 세션 아이디 : 10041
나의 네트워크 그룹 : 20001
> chatThread 시작
10040 : 20001번, 네트워크에서, 객체, 전송
채팅처럼, 보이지만, 객체, 생성, 후, 리플리케이션입니다.
> 전송 완료 !!
10041 : 채팅처럼, 보이지만, 객체, 생성, 후, 리플리케이션입니다.
```