**<4.24>**

TCP Fight예제에 AsyncSelect모델 적용하는 예제진행

/////////////////////////////////////////////////////////////////////

// [www.gamecodi.com](http://www.gamecodi.com/) 이주행 master@gamecodi.com

//

//

/////////////////////////////////////////////////////////////////////

/\*---------------------------------------------------------------

패킷데이터 정의.

자신의 캐릭터에 대한 패킷을 서버에게 보낼 때, 모두 자신이 먼저

액션을 취함과 동시에 패킷을 서버로 보내주도록 한다.

- 이동 키 입력 시 이동동작을 취함과 동시에 이동 패킷을 보내도록 한다.

- 공격키 입력 시 공격 동작을 취하면서 패킷을 보낸다.

- 충돌 처리 및 데미지에 대한 정보는 서버에서 처리 후 통보하게 된다.

---------------------------------------------------------------\*/

#ifndef \_\_PACKET\_DEFINE\_\_

#define \_\_PACKET\_DEFINE\_\_

//---------------------------------------------------------------

// 패킷헤더.

//

//---------------------------------------------------------------

/\*

BYTE byCode; // 패킷코드 0x89 고정.

BYTE bySize; // 패킷 사이즈. (실제 페이로드 사이즈, 헤더/End 코드 제외)

BYTE byType; // 패킷타입.

BYTE byTemp; // 사용안함.

\*/

struct st\_NETWORK\_PACKET\_HEADER

{

BYTE byCode; // 패킷코드 0x89 고정.

BYTE bySize; // 패킷 사이즈.

BYTE byType; // 패킷타입.

BYTE byTemp; // 사용안함.

};

//---------------------------------------------------------------

// 패킷의 가장 앞에 들어갈 패킷코드.

//---------------------------------------------------------------

#define dfNETWORK\_PACKET\_CODE ((BYTE)0x89)

//---------------------------------------------------------------

// 패킷의 가장 뒤에 들어갈 패킷코드.

//---------------------------------------------------------------

#define dfNETWORK\_PACKET\_END ((BYTE)0x79)

//---------------------------------------------------------------

// 패킷의 끝 부분에는 1Byte 의 EndCode 가 포함된다. 0x79

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_CREATE\_MY\_CHARACTER 0

//---------------------------------------------------------------

// 0 - 클라이언트 자신의 캐릭터 할당 Server -> Client

//

// 서버에 접속시 최초로 받게되는 패킷으로 자신이 할당받은 ID 와

// 자신의 최초 위치, HP 를 받게 된다. (처음에 한번 받게 됨)

//

// 이 패킷을 받으면 자신의 ID,X,Y,HP 를 저장하고 캐릭터를 생성시켜야 한다.

//

// 4 - ID

// 1 - Direction

// 2 - X

// 2 - Y

// 1 - HP

//

//---------------------------------------------------------------

/\*

struct stPACKET\_SC\_CREATE\_MY\_CHARACTER

{

DWORD ID;

BYTE Direction;

WORD X;

WORD Y;

BYTE HP;

}

\*/

#define dfPACKET\_SC\_CREATE\_OTHER\_CHARACTER 1

//---------------------------------------------------------------

// 1. 다른 클라이언트의 캐릭터 생성 패킷 Server -> Client

//

// 처음 서버에 접속시 이미 접속되어 있던 캐릭터들의 정보

// 또는 게임중에 접속된 클라이언트들의 생성 용 정보.

//

//

// 4 - ID

// 1 - Direction

// 2 - X

// 2 - Y

// 1 - HP

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_DELETE\_CHARACTER 2

//---------------------------------------------------------------

// 2. 캐릭터 삭제 패킷 Server -> Client

//

// 캐릭터의 접속해제 또는 캐릭터가 죽었을때 전송됨.

//

// 4 - ID

//

//---------------------------------------------------------------

#define dfPACKET\_CS\_MOVE\_START 10

//---------------------------------------------------------------

// 10. 캐릭터 이동시작 패킷 Client -> Server

//

// 자신의 캐릭터 이동시작시 이 패킷을 보낸다.

// 이동 중에는 본 패킷을 보내지 않으며, 키 입력이 변경되었을 경우에만

// 보내줘야 한다.

//

// (왼쪽 이동중 위로 이동 / 왼쪽 이동중 왼쪽 위로 이동... 등등)

//

// 1 - Direction ( 방향 디파인 값 8방향 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_MOVE\_DIR\_LL 0

#define dfPACKET\_MOVE\_DIR\_LU 1

#define dfPACKET\_MOVE\_DIR\_UU 2

#define dfPACKET\_MOVE\_DIR\_RU 3

#define dfPACKET\_MOVE\_DIR\_RR 4

#define dfPACKET\_MOVE\_DIR\_RD 5

#define dfPACKET\_MOVE\_DIR\_DD 6

#define dfPACKET\_MOVE\_DIR\_LD 7

#define dfPACKET\_SC\_MOVE\_START 11

//---------------------------------------------------------------

// 캐릭터 이동시작 패킷 Server -> Client

//

// 다른 유저의 캐릭터 이동시 본 패킷을 받는다.

// 패킷 수신시 해당 캐릭터를 찾아 이동처리를 해주도록 한다.

//

// 패킷 수신 시 해당 키가 계속해서 눌린것으로 생각하고

// 해당 방향으로 계속 이동을 하고 있어야만 한다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값 8방향 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_CS\_MOVE\_STOP 12

//---------------------------------------------------------------

// 캐릭터 이동중지 패킷 Client -> Server

//

// 이동중 키보드 입력이 없어서 정지되었을 때, 이 패킷을 서버에 보내준다.

//

// 1 - Direction ( 방향 디파인 값 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_MOVE\_STOP 13

//---------------------------------------------------------------

// 캐릭터 이동중지 패킷 Server -> Client

//

// ID 에 해당하는 캐릭터가 이동을 멈춘것이므로

// 캐릭터를 찾아서 방향과, 좌표를 입력해주고 멈추도록 처리한다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_CS\_ATTACK1 20

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Client -> Server

//

// 공격 키 입력시 본 패킷을 서버에게 보낸다.

// 충돌 및 데미지에 대한 결과는 서버에서 알려 줄 것이다.

//

// 공격 동작 시작시 한번만 서버에게 보내줘야 한다.

//

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_ATTACK1 21

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Server -> Client

//

// 패킷 수신시 해당 캐릭터를 찾아서 공격1번 동작으로 액션을 취해준다.

// 방향이 다를 경우에는 해당 방향으로 바꾼 후 해준다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_CS\_ATTACK2 22

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Client -> Server

//

// 공격 키 입력시 본 패킷을 서버에게 보낸다.

// 충돌 및 데미지에 대한 결과는 서버에서 알려 줄 것이다.

//

// 공격 동작 시작시 한번만 서버에게 보내줘야 한다.

//

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_ATTACK2 23

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Server -> Client

//

// 패킷 수신시 해당 캐릭터를 찾아서 공격2번 동작으로 액션을 취해준다.

// 방향이 다를 경우에는 해당 방향으로 바꾼 후 해준다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_CS\_ATTACK3 24

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Client -> Server

//

// 공격 키 입력시 본 패킷을 서버에게 보낸다.

// 충돌 및 데미지에 대한 결과는 서버에서 알려 줄 것이다.

//

// 공격 동작 시작시 한번만 서버에게 보내줘야 한다.

//

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_ATTACK3 25

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Server -> Client

//

// 패킷 수신시 해당 캐릭터를 찾아서 공격3번 동작으로 액션을 취해준다.

// 방향이 다를 경우에는 해당 방향으로 바꾼 후 해준다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_DAMAGE 30

//---------------------------------------------------------------

// 캐릭터 데미지 패킷 Server -> Client

//

// 공격에 맞은 캐릭터의 정보를 보냄.

//

// 4 - AttackID ( 공격자 ID )

// 4 - DamageID ( 피해자 ID )

// 1 - DamageHP ( 피해자 HP )

//

//---------------------------------------------------------------

#endif

위방법은 구조체를 하나씩 선언하는 방법으로서 완전히 옛날방식이다.

**구조체로 메시지를 만드는 경우, Pragma pack을 1로 선언하여 패딩을 무시한다.**

**(멤버정렬이 되지않아 성능이 떨어질 것이지만 트래픽을 줄이기위함.)**

**메시지를 struct로 보내지않는 이유**

**1. 서버/클라의 bit(32/64)나 언어(플랫폼)가 다를경우 사용불가 (??)**

**2. 메시지가 추가될때마다 헤더가 변화해야한다. (빌드타임증가)**

**3. payload가 고정크기가 된다.**

특히 게임에서 3번은 치명적인데, 채팅/퀘스트/인벤토리/선물함 등은 모두 가변길이로 표현되는데 구조체를 사용할 경우 비효율 적이다.

((

아이템 리스트를 보내고싶을때 구조체 하나로 표현할 수 없으므로, 예전에는 크기가 0인배열을 선언하여 뒤쪽에 붙는 데이터를 배열로서 접근하는 방식을 사용했다.

하지만 현재 C++표준에서는 배열크기가 반드시 0보다 커야한다.

))

**가변길이가 붙게되는 컨텐츠의 경우 사용할수있는 방법**

**1. 비동기 설계 기반**

비동기로 설계하여 게임은 따로 진행되고, 아이템은 알아서 전송되게 하는방법.

(클라가 해주는게 best)

**2. 메시지를 분리하여 시작과 끝을 알려주는 방법.**

아이템인벤토리, 리스트 정보의 시작, 아이템정보 3종류로 보내는것.

길이가 가변인 컨텐츠 (퀘스트 목록, 선물함목록, 경매장목록)은 모두 이와같은 구조이다.

**3. 직렬화버퍼(Serialize buffer) 사용**

**직렬화란 데이터 구조나 오브젝트 상태를, 동일하거나 다른 컴퓨터 환경에 저장(파일, 메모리 버퍼, 또는 네트워크 링크 간 전송)하고 나중에 재구성할 수 있는 포맷으로 변환하는 과정**이다.

**(오브젝트를 직렬화하는 과정은 ‘오브젝트를 마샬링한다’라고 하기도하고 반대로 일련의 바이트로부터 데이터 구조를 추출하는 일은 역직렬화 또는 deserialization이라고 한다.)**

**직렬화 버퍼는 정보들을 순서대로 밀어넣고 전송하는 용도로, 클래스내부에서 버퍼를가지고 있는 형태이며 큐와같이 Inque,Deque와 같은 인터페이스가 존재한다.**

**이 경우 구조체를 선언하는것이 아니라 프로토콜에 대한 설계나 문서로 설명을 적는 것이다.** (서버와 클라가 같은 플랫폼이 아니기 때문에 excel,word등을 이용한다.)

**이렇게 직렬화 설계가 된다면 가변적인 크기가 되기때문에 특정 조건을 통해 추가적으로 데이터가 들어가는것이 가능해진다.**

(어떤 아이템을 들고있다면 뒤에 추가로 아이템이 달라붙는 등 정보를 더 넣을수있게 됨)

**단 직렬화버퍼는 구조체와 다르게 들어가야할 데이터의 순서가 뒤바뀌거나 사이즈를 잘못잡을 경우 문제가 생길 여지가 남아있는데다가, 구조체를 선언하는걸 없애긴 헀지만 데이터를 넣고 빼는 부분에는 결국 하드코딩이 들어가야한다.**

**4. 메시지 생성부를 자동화하는법**

위와같은 직렬화버퍼의 단점을 보완하기위해 별도의 스크립트로 컴파일러를 통해 메시지 생성부를 자동화시킨다.

데이터를 마샬링(marshalling : 분해 분석)해서 수신/송신부가 모두 나오게끔 하는것.

이는 함수만 호출시키면 바로바로 데이터가 나가게끔 하는 방식으로 대부분 이방식을 채택하여 사용한다.

원칙적으로 따지자면 헤더 자료형에는 unsigned, signed도 구분한다.

참고 : SC(Server-> Client) / CS(Clinet -> Server)

/////////////////////////////////////////////////////////////////////

// [www.gamecodi.com](http://www.gamecodi.com/)

//

//

/////////////////////////////////////////////////////////////////////

/\*---------------------------------------------------------------

패킷데이터 정의.

자신의 캐릭터에 대한 패킷을 서버에게 보낼 때, 모두 자신이 먼저

액션을 취함과 동시에 패킷을 서버로 보내주도록 한다.

- 이동 키 입력 시 이동동작을 취함과 동시에 이동 패킷을 보내도록 한다.

- 공격키 입력 시 공격 동작을 취하면서 패킷을 보낸다.

- 충돌 처리 및 데미지에 대한 정보는 서버에서 처리 후 통보하게 된다.

---------------------------------------------------------------\*/

#ifndef \_\_PACKET\_DEFINE\_\_

#define \_\_PACKET\_DEFINE\_\_

//---------------------------------------------------------------

// 패킷헤더.

//

//---------------------------------------------------------------

/\*

BYTE byCode; // 패킷코드 0x89 고정.

BYTE bySize; // 패킷 사이즈. (실제 페이로드 사이즈, 헤더/End 코드 제외)

BYTE byType; // 패킷타입.

BYTE byTemp; // 사용안함.

\*/

struct st\_NETWORK\_PACKET\_HEADER

{

BYTE byCode; // 패킷코드 0x89 고정. 이게 틀릴경우 로그남기고 끊음

BYTE bySize; // 패킷 사이즈. (보통은 2Byte)

BYTE byType; // 패킷타입.

BYTE byTemp; // 사용안함.

};

//---------------------------------------------------------------

// 패킷의 가장 앞에 들어갈 패킷코드.

//---------------------------------------------------------------

#define dfNETWORK\_PACKET\_CODE ((BYTE)0x89)

//---------------------------------------------------------------

// 패킷의 가장 뒤에 들어갈 패킷코드.

//---------------------------------------------------------------

#define dfNETWORK\_PACKET\_END ((BYTE)0x79)

//---------------------------------------------------------------

// 패킷의 끝 부분에는 1Byte 의 EndCode 가 포함된다. 0x79

//

//---------------------------------------------------------------

사실 뒤에 붙이는건 아무의미가없음.

(잘못된 메시지가 오면 정상메시지를 찾아서 처리해줘야하지않을까? X 그냥 끊어버린다. 물려둘이유가없으므로 그 즉시 연결을 끊는다)

#define dfPACKET\_SC\_CREATE\_MY\_CHARACTER 0

//---------------------------------------------------------------

// 0 - 클라이언트 자신의 캐릭터 할당 Server -> Client

//

// 서버에 접속시 최초로 받게되는 패킷으로 자신이 할당받은 ID 와

// 자신의 최초 위치, HP 를 받게 된다. (처음에 한번 받게 됨)

//

// 이 패킷을 받으면 자신의 ID,X,Y,HP 를 저장하고 캐릭터를 생성시켜야 한다.

//

// 4 - ID

// 1 - Direction

// 2 - X

// 2 - Y

// 1 - HP

//

//---------------------------------------------------------------

/\*

struct stPACKET\_SC\_CREATE\_MY\_CHARACTER

{

DWORD ID;

BYTE Direction;

WORD X;

WORD Y;

BYTE HP;

}

\*/

이때 CreateMyCharacter가 두번올것이라는 것은 고려하지않는다. 서버를 믿고가기 때문에 클라는 이러한 예외처리를 따로하지않음.

#define dfPACKET\_SC\_CREATE\_OTHER\_CHARACTER 1

//---------------------------------------------------------------

// 1. 다른 클라이언트의 캐릭터 생성 패킷 Server -> Client

//

// 처음 서버에 접속시 이미 접속되어 있던 캐릭터들의 정보

// 또는 게임중에 접속된 클라이언트들의 생성 용 정보.

//

//

// 4 - ID

// 1 - Direction

// 2 - X

// 2 - Y

// 1 - HP

//

//---------------------------------------------------------------

실제 게임이라면 OhterChar보다 내캐릭터에 훨씬더 많은 정보가 올것이지만 지금은 별내용이 없으므로 똑같다.

#define dfPACKET\_SC\_DELETE\_CHARACTER 2

//---------------------------------------------------------------

// 2. 캐릭터 삭제 패킷 Server -> Client

//

// 캐릭터의 접속해제 또는 캐릭터가 죽었을때 전송됨.

//

// 4 - ID

//

//---------------------------------------------------------------

서버가 DELETE하라고 오면 객체가 살아있든 어쨌든 지운다. (나중에는 섹터처리로 인해 지워지는데 지워지는 상황자체를 구분하지않을것)

#define dfPACKET\_CS\_MOVE\_START 10

//---------------------------------------------------------------

// 10. 캐릭터 이동시작 패킷 Client -> Server

//

// 자신의 캐릭터 이동시작시 이 패킷을 보낸다.

// 이동 중에는 본 패킷을 보내지 않으며, 키 입력이 변경되었을 경우에만

// 보내줘야 한다.

//

// (왼쪽 이동중 위로 이동 / 왼쪽 이동중 왼쪽 위로 이동... 등등)

//

// 1 - Direction ( 방향 디파인 값 8방향 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_MOVE\_DIR\_LL 0

#define dfPACKET\_MOVE\_DIR\_LU 1

#define dfPACKET\_MOVE\_DIR\_UU 2

#define dfPACKET\_MOVE\_DIR\_RU 3

#define dfPACKET\_MOVE\_DIR\_RR 4

#define dfPACKET\_MOVE\_DIR\_RD 5

#define dfPACKET\_MOVE\_DIR\_DD 6

#define dfPACKET\_MOVE\_DIR\_LD 7

#define dfPACKET\_SC\_MOVE\_START 11

//---------------------------------------------------------------

// 캐릭터 이동시작 패킷 Server -> Client

//

// 다른 유저의 캐릭터 이동시 본 패킷을 받는다.

// 패킷 수신시 해당 캐릭터를 찾아 이동처리를 해주도록 한다.

//

// 패킷 수신 시 해당 키가 계속해서 눌린것으로 생각하고

// 해당 방향으로 계속 이동을 하고 있어야만 한다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값 8방향 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

**캐릭터를 동기화 시키는 방법**

좌표를 동기화하는것은 매우 비효율적이므로 **시간과같은 단위를 기준으로 간격을 나눠 보낸다.**

**간격마다의 이동을 부드럽게 하기위해 좌표를 온것을 기준으로 중간과정을 계산하는것을 (Dead reckoing : 추측 항법)이라고 한다.**

**이는 결국 완벽하게 프레임단위로 동기화가 불가능하기 때문에 이는 좌표를 가지고 중간에 존재하지 않는 좌표를 예측해서 만드는 것으로 결국 모든 클라에는 Dead reckoing이 들어가야한다는 얘기가 된다.**

(현재좌표 이동방향 시간)을 가지고있다면 계산이 가능할것이나, 즉각적이고 불규칙적인 이동에 대해서는 정확한 계산이 불가능 할것이다. (항해 게임에 적합할것.)

이때 생기는 오차는 허용범위 안쪽이므로 최대한 열심히 보내면서 오차를 최소화하는것으로 넘어간다.

(지금 예제에서는 8개의 방향이 정해져있어 완벽하게 맞출 수 있으므로 Dead reckoing을 할 필요가 없다.)

만약 MMORPG의 동기화를 요구하는 게임이라면 서버로부터 응답을 받아서 가는게 맞지만 지금 예제는 MO느낌이므로 서버에 통보하는 형태로 가게된다. (액션성이 강한 게임들도 이런방식을 채택함.)

해당예제에서는 자기자신의 플레이어가 내 클라주도하에 움직이고 서버는 경유지역할을 해주는게 전부이므로, 클라 좌표를 서버가 무시하지않고 어느정도 믿어준다.

(예제에서 좌표의 주체는 클라가 되지만 일반적인 서버-클라 구조라면 주체는 서버, 클라는 하나의 Input장치이자 viewer가 될것.)

#define dfPACKET\_CS\_MOVE\_STOP 12

//---------------------------------------------------------------

// 캐릭터 이동중지 패킷 Client -> Server

//

// 이동중 키보드 입력이 없어서 정지되었을 때, 이 패킷을 서버에 보내준다.

//

// 1 - Direction ( 방향 디파인 값 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_MOVE\_STOP 13

//---------------------------------------------------------------

// 캐릭터 이동중지 패킷 Server -> Client

//

// ID 에 해당하는 캐릭터가 이동을 멈춘것이므로

// 캐릭터를 찾아서 방향과, 좌표를 입력해주고 멈추도록 처리한다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

정지메시지를 보내도 latency때문에 (해당 예제는 50프레임이므로) 결국은 모든 메시지는 20m/s후에 오는 구조이다. 이때문에 좌표가 틀어질 수밖에 없으므로 모든 행위를 할때 좌표를 담게되고, 서버에서는 어느정도 이를 믿어준다.

이는 클라의 좌표를 전혀 믿지않는 일반적인 방법과는 다르며, 우리게임상황에 맞춘것.

(게임에서 방향은 중요한것이므로 서버는 항상 기억.)

**필드를 움직이는 MMORPG라면, 메시지 프로토콜 구조를 최소단위로 하게된다.**

**목적지가 같으면 끝이기 때문에 목적지를 공유하는 방식(키보드가 입력장치라면 아주짧은 미래)**

#define dfPACKET\_CS\_ATTACK1 20

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Client -> Server

//

// 공격 키 입력시 본 패킷을 서버에게 보낸다.

// 충돌 및 데미지에 대한 결과는 서버에서 알려 줄 것이다.

//

// 공격 동작 시작시 한번만 서버에게 보내줘야 한다.

//

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_ATTACK1 21

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Server -> Client

//

// 패킷 수신시 해당 캐릭터를 찾아서 공격1번 동작으로 액션을 취해준다.

// 방향이 다를 경우에는 해당 방향으로 바꾼 후 해준다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_CS\_ATTACK2 22

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Client -> Server

//

// 공격 키 입력시 본 패킷을 서버에게 보낸다.

// 충돌 및 데미지에 대한 결과는 서버에서 알려 줄 것이다.

//

// 공격 동작 시작시 한번만 서버에게 보내줘야 한다.

//

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_ATTACK2 23

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Server -> Client

//

// 패킷 수신시 해당 캐릭터를 찾아서 공격2번 동작으로 액션을 취해준다.

// 방향이 다를 경우에는 해당 방향으로 바꾼 후 해준다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_CS\_ATTACK3 24

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Client -> Server

//

// 공격 키 입력시 본 패킷을 서버에게 보낸다.

// 충돌 및 데미지에 대한 결과는 서버에서 알려 줄 것이다.

//

// 공격 동작 시작시 한번만 서버에게 보내줘야 한다.

//

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_ATTACK3 25

//---------------------------------------------------------------

// 캐릭터 공격 패킷 Server -> Client

//

// 패킷 수신시 해당 캐릭터를 찾아서 공격3번 동작으로 액션을 취해준다.

// 방향이 다를 경우에는 해당 방향으로 바꾼 후 해준다.

//

// 4 - ID

// 1 - Direction ( 방향 디파인 값. 좌/우만 사용 )

// 2 - X

// 2 - Y

//

//---------------------------------------------------------------

#define dfPACKET\_SC\_DAMAGE 30

//---------------------------------------------------------------

// 캐릭터 데미지 패킷 Server -> Client

//

// 공격에 맞은 캐릭터의 정보를 보냄.

//

// 4 - AttackID ( 공격자 ID )

// 4 - DamageID ( 피해자 ID )

// 1 - DamageHP ( 피해자 HP )

//

//---------------------------------------------------------------

서버에는 공격이라는 액션이없고, 충돌판정과 데미지

클라는 공격이라는 액션과 애니메이션이 있지만 서버는 함수 호출한번으로 할일이 끝나게 된다. (충돌판정, 데미지계산)

이펙트는 주먹질을 했을때 클라에서 적절한 시기에 터지면 될 것.

#endif

**Send메시지 설계**

그리고 Send할때마다 함수를 하나씩 만들것인데, Send메시지당 함수를 하나씩 만드는 방식을 선택한다.우리가 보내고싶을때 메시지를 만들어서 SendAPI를 호출한다면 전혀 관리가 되지않음.

이때 매개변수는 인자를 최대한 풀어서 해당 함수자체에서 컨텐츠쪽과 의존성을 떼서 변경시 손델일이 적어지도록 설계

**프로토콜 설계시**

메시지별로 하나씩 수신파트 송신파트 프로토콜메시지 껍데기를 잡는다.

그리고 이에대한 Switch Case, 송수신파트를 만들고 서버클라를 만들어 패킷이 분기를 타고 잘 도착하는지 확인.

이것이 확인되었으면 이제 컨텐츠를 넣는다.

**이떄, 오브젝트를 관리하는 관리자부터 염두하여 로직을 짜야한다.**

**<4.29>**

**TCP Fighter Client Network 연동**

윈도우용 게임 클라이언트에서 가장 대중적으로 사용하는 AsyncSelect모델을 사용.

1.키보드 조작의 온라인 네트워크 게임을 구현한다.

2.수신부와 송신부에 링버퍼(스트리밍 큐)를 사용한다.

3.recv와 send과정에 버퍼로 카피하는 기본방식을 사용한다.

4.완성 후 링버퍼를 다이렉트로 사용하도록 변경해본다.

**1.클라이언트 메인함수 구조**

1. Sprite및 GameData Load.

2. IP입력을 위한 다이얼로그 생성 (생략)

3. 윈도우 생성

4. winsock초기화 및 Socket생성

5. WSAAsyncSelect등록

6. 서버로 connect접속 시도

WSAAsyncS

//WSAAsyncSelect에 등록한 소켓은 자동으로 논블럭 소켓이 되므로, connect이후 연결 성공판단이 불가능하다.

따라서 전역변수로 flag를 두어 이를 판단하고 접속된 상황이 아니라면 게임로직으로 들어가지않는다.

MSG msg;

// 기본 메시지 루프입니다:

while (true)

{

if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))

{

if (msg.message == WM\_QUIT)

break;

TranslateMessage(&msg);

DispatchMessage(&msg);

}

else

{

if (**g\_Connect** == TRUE)

GamePlay();

}

}

void Update(void)

{

// 1.키보드 입력처리

// 2.객체 액션 처리

// 3.그리기 & 프레임 스킵

// 4. 플립  
}

Select모델을 사용한다면 Update부분에 select호출 및 네트워크 검사가 들어가야 하지만 AsyncSelect를 사용하기 때문에 네트워크가 연동되어도 게임의 실제 Update부분은 변화가 없다.

**2. 수신 이벤트 처리 및 패킷 처리**

소켓 및 네트워크 처리에 필요한 변수들을 전역으로 선언 한다.

SOCKET g\_Socket;

CRingBuffer g\_RecvQ;

CRingBuffer g\_SendQ;

BOOL g\_SendPossibility //Send 가능여부 판단

윈도우 프로시저의 네트워크 이벤트 발생시 수신 처리부

//

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message)

{

case UM\_NETWORK:

{

if (NetWorkProc(wParam, lParam) == FALSE)

{

MessageBox(hWnd, \_T("NetWorkProc Close."), \_T("연결 끊김"), MB\_OK);

ErrorPrint(-1);

}

}

break;

}

**NetWorkProc**

BOOL NetWorkProc(WPARAM wParam, LPARAM lParam)

{

//WinProc안에서 WSAEGetLastError().

if (WSAGETSELECTERROR(lParam) != 0)

{

DWORD err = WSAGETSELECTERROR(lParam);

return FALSE; //에러 발생, 종료처리 단계로 진행.

}

switch (WSAGETSELECTEVENT(lParam))

{

case FD\_CONNECT: // 연결성공 처리.

g\_Connect = true;

return TRUE;

case FD\_CLOSE: // 연결끊김 처리.

//1234

MessageBox(hWnd, \_T("접속 종료."), \_T("연결 끊김"), MB\_OK);

ErrorPrint(WSAGetLastError());

return FALSE;

break;

case FD\_READ:

if (ProcRead() == FALSE)

return FALSE;

break;

case FD\_WRITE:

g\_Send = true;

if (SendEvent() == FALSE)

return FALSE;

break;

}

return TRUE;

}

FD\_READ와 FD\_WRITE 처리 부분을 별도 함수로 분리.

**FD\_READ에서 호출될 함수**

recv호출(0일경우 종료, SOCKET\_ERROR일경우 예외처리)

데이터를 1이라도 받았다면 RecvQ로 들어가게될것.

완료패킷 처리 부분은 RecvQ에 들어있는 모든 완성패킷을 처리해야한다.

while(1)

{

* **RecvQ에 최소한의 사이즈가 있는지 확인 (조건:헤더사이즈 초과)**
* **RecvQ에서 헤더를 peek**
* **헤더 Code확인(내 패킷이 맞는지)**
* **헤더의 Len값과 RecvQ의 데이터 사이즈 비교**
* **완성 패킷 사이즈 : 헤더 + (PayLoad)Len + 엔드코드**
* **데이터 Peek했던 헤더를 RecvQ에서 삭제(Deque아님)**
* **RecvQ에서 Len만큼 임시 패킷 버퍼로 뽑는다.**
* **RecvQ에서 EndCode인 1byte로 뽑는다.**
* **EndCode확인**
* **헤더의 타입에 따른 분기를 위해 PacketProc를 호출**

}

void PacketProc(BYTE byPacketType, char\* Packet)

{

switch (byPacketType)

{

case dfPACKET\_SC\_CREATE\_MY\_CHARACTER:

netPacketProc\_CreateMyCharacter(Packet);

break;

case dfPACKET\_SC\_CREATE\_OTHER\_CHARACTER:

netPacketProc\_CreateOtherCharacter(Packet);

break;

**.**

**.**

**.**

default:

{

//default로 들어온것은 잘못된 상황이므로 실제 서버일경우 로그를 남기고 끊어야할것.

\_tprintf(\_T("PacketProc default"));

ErrorPrint(-1);

}

}

}

각각의 패킷 프로시저 함수에는 인자로 들어온 패킷버퍼를 해당 구조체 타입으로 변환하여 컨텐츠 처리를 하게된다.

예시)

BOOL netPacketProc\_CreateMyCharacter(char\* pPacketBuffer)

{

stPACKET\_SC\_CREATE\_MY\_CHARACTER\* pPacket = (stPACKET\_SC\_CREATE\_MY\_CHARACTER\*)pPacketBuffer;

Player = new CPlayerObject(pPacket->X, pPacket->Y, pPacket->HP, pPacket->ID, TRUE);

ObjList.push\_back(Player);

SortObjList();

return TRUE;

}

**3. 패킷생성 및 송신처리**

서버로 데이터 Send시에는 항상 SendQ에 먼저 데이터를 넣고, 현재 Send가 가능한 상태인지 확인 후 Send를 호출하여 정상적으로 보내진 사이즈를 SendQ에서 지우는 형식으로 진행한다.

위에 언급했듯 패킷데이터 생성은 각각의 패킷마다 생성 함수를 별도로 만들어 내부에서 데이터를 셋팅하고, 매개변수는 최대한 외부코드에 종속되지않도록 전달한다.

void mpMoveStart(st\_NETWORK\_PACKET\_HEADER header, stPACKET\_CS\_MOVE\_START\* pPacket, BYTE Direction, WORD X, WORD Y)

{

header.byCode = dfNETWORK\_PACKET\_CODE;

header.bySize = sizeof(stPACKET\_CS\_MOVE\_START);

header.byType = dfPACKET\_CS\_MOVE\_START;

header.byTemp = 0;

stPACKET\_CS\_MOVE\_START Packet;

Packet.Direction = Direction;

Packet.X = X;

Packet.Y = Y;

SendPacket(&header, (char\*)&Packet);

}

SendPacket의 경우 사용자가 데이터를 보낼때도 호출하지만 **FD\_WRITE**이벤트 발생시에도 호출하여 보내지못하고 남아있는 SendQ의 데이터를 보내주게 된다.

void SendPacket(st\_NETWORK\_PACKET\_HEADER\* pHeader, char\* pPacket)

{

//접속상태 예외처리

if (g\_Connect == FALSE)

return;

//SendQ에 헤더, 패킷, EndCode를 차례로 넣는다.

SendQ->Enque((char\*)pHeader, sizeof(\*pHeader));

SendQ->Enque((char\*)pPacket, pHeader->bySize);

BYTE EndCode = dfNETWORK\_PACKET\_END;

SendQ->Enque((char\*)&EndCode, sizeof(EndCode));

//실제 Send를 시도하는 함수호출

SendEvent();

}

SendPacket에서는 Send를 세번하고있는데, 이는 빠른이해를 위해 구현된것.

차후에 직렬화 버퍼를 사용하여 해결한다.

BOOL SendEvent()

{

if (g\_Send == FALSE)

return FALSE;

while (true)

{

//SendQ에 데이터가 없을 경우 루프를 멈추게될것.

//SendQ에서 보내기용 임시버퍼로 Peek.

char Sendbuff[1500];

INT SendQpp\_ret = SendQ->peek((char\*)Sendbuff, sizeof(Sendbuff));

if (SendQpp\_ret == 0)

break;

//send시도

INT SendAPI\_ret = send(Socket, Sendbuff, SendQpp\_ret, NULL);

if (Socket == SOCKET\_ERROR)

{

DWORD err = WSAGetLastError();

//우드블럭이 아닌 에러일경우 이는 치명적에러

if (err != WSAEWOULDBLOCK)

return FALSE;

//우드블럭이 나왔을경우 지금 보내지못하는 상황

else

{

g\_Send = FALSE;

break;

}

}

//Send한만큼 SendQ에서 데이터 제거

SendQ->MoveFront(SendAPI\_ret);

}

return TRUE;

}

**Send구현시 주의사항**

클라이언트의 조작에 대한 변화를 서버로 통지하는 개념이므로 키보드 눌림여부에 따라서 Send를 하는것이 아니고 액션이 바뀔때마다 이에대한 패킷을 서버로 보내야한다.

또한 이동중 공격키를 누를씨 MOVE – STOP - ATTACK으로 전환시켜 주어야 한다.

MOVE중에 바로 ATTACK으로 넘어갈 시 서버에서 정상적으로 처리되지 않을 수 있다.

데미지 패킷 수신시 이에대한 이펙트 처리는 클라이언트가 알아서 진행해야 하며, 공격자의 스프라이트 애니메이션을 검사하여 적절한 순간에 이펙틀를 터뜨려 주어야 한다.

---------------------------------------------------------------------------------

**프라우드넷의 RPG개념 자료**

**빠르고 효율적인 원격 프로시저 호출 구현**

**추측항법, 분산게임 서버, 스로틀링, 부하균등화, Smart seeds등의 측면에 초점을 둔 네트윅 게임개발 관련 기법들은 하나이상의 메시지를 보내고 받는다는 공통점이 있고 이는 결국 네트웍 코드에서 중복이 가장 많이 생겨나는 주요원인중 하나이다.**

네트워크 메시지를 주고받기위한 코드들은 대부분 비슷비슷하고, 따라서 분리할 수 있다. 메시지를 주고받는 코드에는 수많은 switch-case문들과 struct 정의도 포함되며, 이런 코드의 중복은 개발 과정에서 메시지가 추가되거나 변함에 따라 더욱더 심해진다.

**Legacy RPC와 우리가 접근할 RPC의 개념**

RPC는 한 컴퓨터의 프로그램이 다른 컴퓨터의 함수를 호출하는 과정을 추상화 하는 것이다.

RPC구현은 여러가지로, 일부는 몇몇 주요 기업에서 제공한다.

(MS-RPC, DEC-RPC, DCOM, CORBA, Java RMI……)

이러한 legacy RPC시스템은 안정적이며, 보안, 인증, 여러 프로토콜 호환성 등 다양한 기능을 지원한다.

하지만 게임 프로그래밍에는 적합하지않은 단점들을 가지고있다.

1. 이해하기 어렵고, 복잡하며 무겁다.

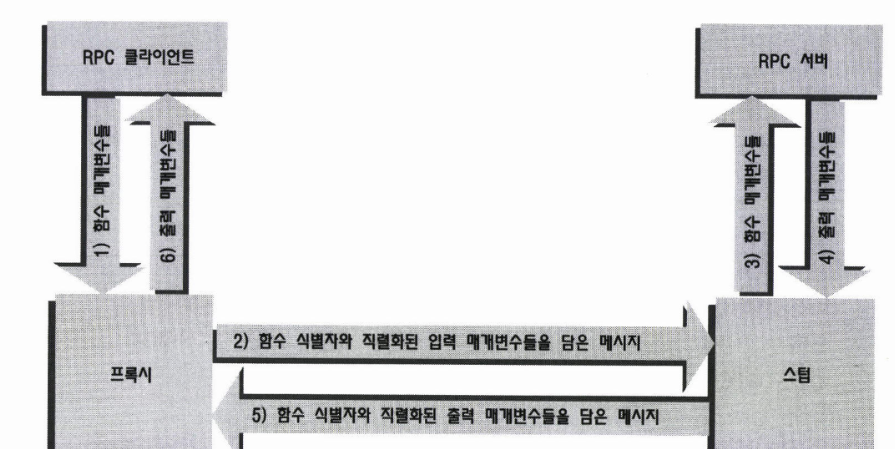
2. 메시지 형식을 마음대로 제어할 수 없다.

3. 스로틀링을 적용하는등, 메시지 전송을 마음대로 제어하는 것이 불가능하다.

4. 이들이 사용하는 구식 비동기 모형은 동기모형에 비해 훨씬 더 복잡하다.

여기서 소개할 접근방식은, RPC컴파일러와 실행시점 엔진으로 구성되어있다.

이 RPC시스템은 전적으로 응용프로그램의 제어 하에 있기 때문에, 필요에 따라 최적화가 가능.



**RPC클라 - RPC서버**

일반 클라-서버의 개념과는 다르다.

보통 하나의 게임서버와 하나의 게임클라는 각자 두개의 이질적인 RPC들을 가진다.

(클라-> 서버, 서버-> 클라)

반면 동위간(p2p) 게임의 경우 게임당 하나씩의 동질적인 RPC를 이용해서 요청을 주고받는다.

**RPC동기 - RPC비동기**

**동기 RPC는 출력 매개변수가 존재하여, RP클라가 동기 RPC요청을 RPC서버로 보내면 RPC서버가 함수호출 결과를 돌려줄때까지 기다린다.**

**비동기 RPC는 출력매개변수가 없고 반환호출을 기다리지않으며, 신뢰성없는 메시징을 허용하며, 네트워크 전송 과정에서 유실될 수 있다.**

**(네트워크 전송과정에서 메시지 유실은 흔한 경우이므로 <- ? TCP단이라면 반드시 성공할것인데..)**

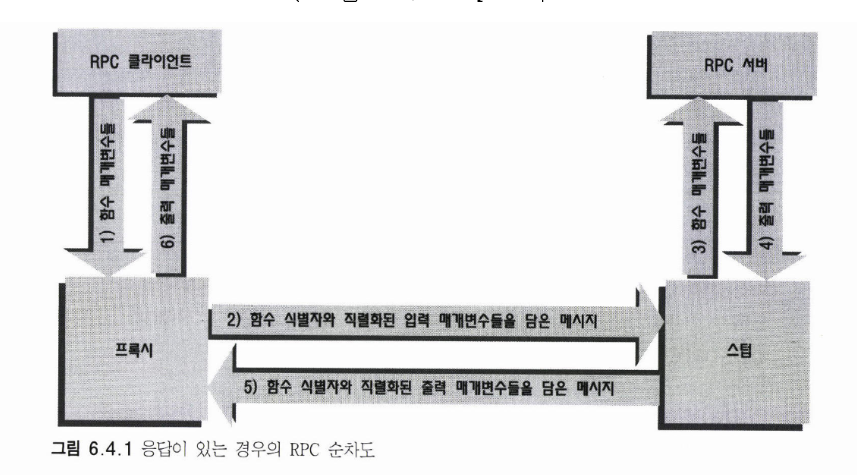
**게임쪽 대부분의 RPC서버 프로그램은 RPC클라의 실행을 기다려야하는 상황을 설계단에서부터 피하기 때문에, 게임쪽에서는 동기RPC를 사용하는 경우는 거의없다.**

**(이는 병목이나 교착상태를 야기)**

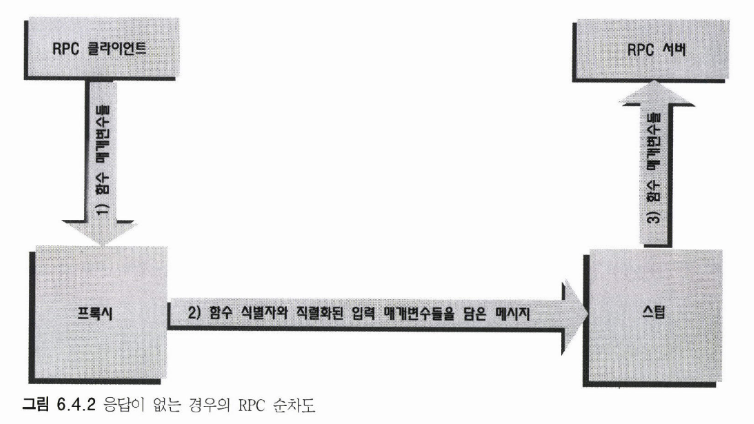
대신 RPC서버로의 요청과 그 응답을 개별적인 메시지로 정의하는 방법을 사용한다.

많은 게임들은 RPC로부터 응답을 기다려야하는데, 이 경우 적절한 대기 애니메이션을 표시하고 사용자가 취소버튼을 눌러 과정을 취소할 수 있게 한다.

**RPC설계 - 응답이있는 RPC (동기)**



**RPC설계 - 응답이없는 RPC (비동기)**



RPC클라가 어느 RPC함수를 호출할 때, RPC클라가 RPC서버에 있는 함수의 본문을 직접수행하는것은 아니다.

**1. 프록시**

RPC클라이언트는 함수 매개변수들을 모으고 그것들을 하나의 메시지에 추가한다.

호출할 함수를 식별하는 함수식별자는 메시지 헤더에 담아둔다. 이 단계를 수행하는 코드를 프록시라고 부른다.

(실제 함수를 수행하는 것이 아니라 식별자와 매개변수들을 하나의 스트림으로 직렬화하여 실제 호출이 일어나게 하는 모양만 같은 대리자 역할이기 때문.)

**2. 스텁**

그러면 RPC서버는 메시지를 받아서, 해당 메시지를 해석하여 어떤 RPC함수를 실행할 것인지 결정하고 매개변수들을 추출하여 RPC함수 본문, 또는 또다른 함수에 전달한다.

**이처럼 여러 함수들 중 하나를 적절히 선택해서 호출하는 역할을 담당하는 함수를 스텁이라고 부른다.**

**비동기RPC에서는 이러한과정에서 RPC응답단계는 무시된다.**

하나의 RPC호출이 적절한 메시지로 변환되고, 또 반대쪽에서 제대로 해석되게 하려면 모든 RPC메시지가 하나의 특정된 방식으로 생성되어야 한다.

프록시 쪽에서 하나의 RPC호출이 일어나면 생성된 메시지에는 그 함수 호출에 필요한 모든 정보가 포함되어야 한다.

- 함수 식별자 (미리 정의된 숫자. RPC함수가 선언될때마다 증가)

- 함수 매개변수들의 직렬화 결과 (직렬화는 프록시가 수행한다.)

오류검출과 복구 기능을 생략한다면, 매개변수개수나 각각의 형식은 함수 식별자에서 파악할 수 있으므로 직렬화만 하면 끝난다.

다양한 자료형식을 가진 매개변수들을 직렬화하는 방법은 여러 개다.

여기서는 메시지 클래스와 이름은 같으나 매개변수 형식들이 다른 중복적재된 직렬화 함수를 예시로 든다.

void Message\_Read(CMsg& m, double &val);

void Message\_Read(CMsg& m, std::string, &val);

void Message\_Write(CMsg& m, double &val);

void Message\_Write(CMsg& m, std::string, &val);

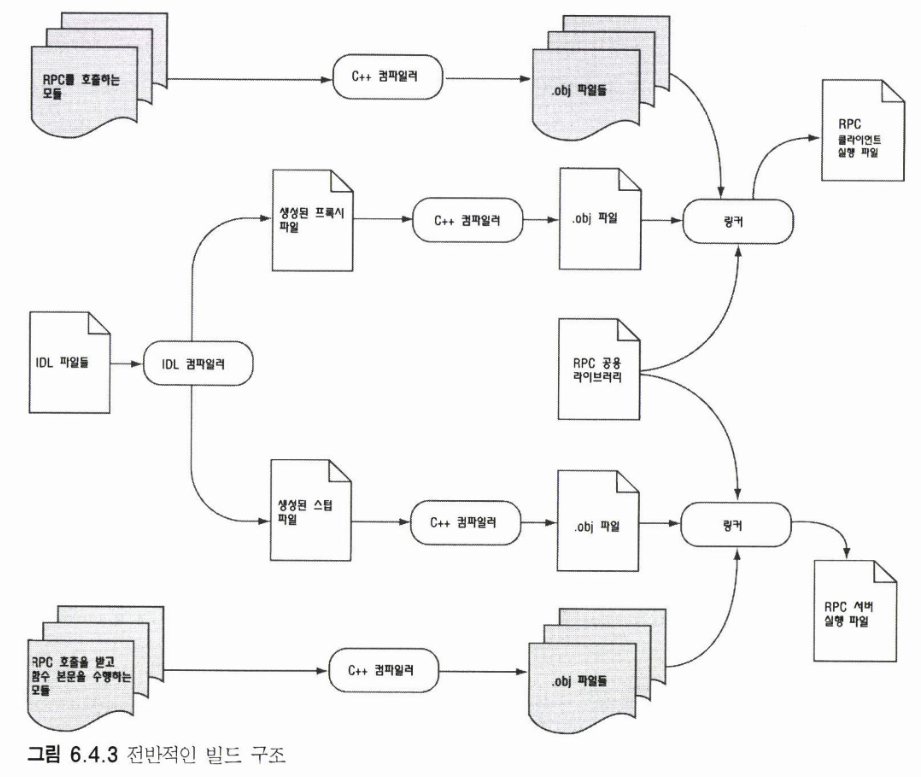
이러한 함수들을 일일이 코딩하는 것은 지루하고 실수하기 쉽기 때문에, 적절한 자동화가 필요.

우리가 원하는 것은 네트워킹 코드를 간단한 한줄짜리 선언들로부터 생성하는 것이다.

이를위해 RPC는 인터페이스 정의언어(Interface Definition Language : IDL)을 사용.

IDL파일이 응용 프로그램의 빌드에 직접적으로 쓰이는 프로그램 소스파일은 아니며, IDL파일은 IDL컴파일러라는 도구로 컴파일한다.

이 후 프록시를 정의하는 소스파일과 스텁을 정의하는 소스파일이 생성되며, RPC를 원격 호출하고자 하는 프로그램은 그 프록시 소스 파일을 링크하고, 실제 RPC함수를 가진 프로그램은 해당 스텁파일을 링크하면 된다.



프록시와 스텁 모두에서 적어도 하나의 보조함수 또는 클래스가 필요하다.

이런 함수또는 클래스는 공용 런타임이라고 부르는 하나의 라이브러리에 넣어둔다.

단, 생성된 프록시와 스텁코드를 소스제어 시스템에 포함회서는 안된다.

IDL컴파일러는 수정된 IDL파일을 컴파일하면서 프록시아 서버 코드를 다시 덮어 씌우므로 프록시 서버와 스텁코드가 체크아웃되지 않은 (그래서 읽기전용) 경우에는 빌드가 실패할 수 있다.

**RPC의 구현**

RPC함수는 본질적으로 다른프로세스에 있는 코드이므로, RPC클라에서 직접 호출할 수 없다.

그러나 RPC클라에 RPC함수와 동일한 함수 원형(서명)을 가진 지역 함수를 두는 것은 가능.

그러한 함수는 주어진 함수호출을 하나의 메시지로 만들고 그것을 실제 함수 본문이 있는 원격 프로세스에 보냄으로서 호출을 대신한다.

이러한 지역 버전함수를 프록시 코드라고 부른다.

IDL컴파일러는 그러한 프록시 코드를 생성하는 역할을 담당한다.

IDL컴파일러는 명시된 코드 패턴에 근거해서 적절한 프록시 코드를 생성한다.

그러한 코드패턴 지정을 정의하는

<자료 742>

----------------------------------------------------------------------------------

**RPC(Remote procedure call) - 수업**

**어떤 메시지를 만들어서 Send를하면 수신부쪽에서 매시지에 따른 분기를 타고 그행위를한다.**

**이를 자동화를 시키고 나서보니 마치 원격지에 있는 함수를 호출시키는것과 같다라는것.**

예제로 본다면, MOVE, START메시지를 생성해 이것을 Send한다.

상대방은 이를받아 Recv하여, PROC\_MOVESTART로 호출이 들어간다.

이 형태가 마치 상대방의 무빙스타트 함수를 호출시키는것과 같은 개념. (RPC)

**안쪽에서보면 send하여 분기문받아 호출시키는 것이지만, 이를 랩핑하여 자동화 시킨다면 RPC의 개념으로 볼수있다.**

**이를 자동화시켜 어떤 함수를 호출하면, 반대쪽에서도 자동으로 함수호출이 되는 개념으로 본다는 것.**

(메시지 하나당 함수 하나)

**RPC 클라 - RPC 서버**

RPC개념은 여기저기 존재한다.

윈도우자체에도 RPC서비스가 존재하며, 업체들마다 RPC프로토콜이 존재한다.

이를 써먹으려면, 우리는 우리게임의 프로토콜 규격에맞는 우리만의 RPC를 만들어 봐야할 것이다.

RPC를 호출하면 요청하는 쪽이 RPC클라, 받는쪽이 RPC서버 (일반적인 클라-서버개념 X)

쌍방이 동등한 입장으로, 원격지에 있는 함수를 호출하는 개념이다.

**새로운 메시지 추가 시**

예제에서 사용한 고전적인 방법에서, 새로운 메시지를 추가하려면 4가지의 변경이 필요하다.

**1. 세메시지 ID**

**2. 구조체 (직렬화 버퍼 사용시 생략가능)**

**3. 구조체를 전송하는 함수**

**4. 메시지를 받아 처리하기위한 case절.**

위와 같은 작업들을 모두 자동화 하겠다는 얘기가 된다.

따라서 클라도 RPC클라가 될수있고, 서버도 RPC클라가 될수있다.

**RPC - 동기 / 비동기**

동기 RPC는 출력 매개변수가 있고, RPC클라 프로그램이 동기 RPC요청을 RPC서버로 보낸다.

RPC클라는 RPC서버가 결과를 돌려줄때까지 기다린다.

**rpc 서버 ㅡ rpc 클라**

RPC에서 A함수를 호출시킨다면, 내부에서는 데이터를 만들어 Send한다.

그리고 상대는 데이터를 받아 프로시져까지 호출하고, 그 결과를 다시 보낸다.

**동기 RPC**

게임입장에서는 네트워크 Send / Recv의 코드파트가 완전히 분리되어있다.

**따라서 Select를 쓰든 IOCP를 쓰든 이는 구현하기 엄청나게 난해해진다.**

(Send한다음 Select루프..? 아니면 IOCP응답..? 난해함. 만든다고 해도 병목이나 교착현상 야기)

네트워크 Send이후에 Recv결과까지 받아서 처리하는것까지 함수에 들어가야 하기 때문이다.

**게임에서는 동기RPC개념이 적합하지않으며 구현도 어렵기 때문에 생략한다.**

**비동기 RPC**

**비동기RPC는 리턴, 파라미터가 없다. 결과를 모르므로 출력도 없다.**

비동기 RPC는 신뢰성 없는 메시지를 허용한다는 얘기가 있는데 (프라우드넷 자료), 이는 네트워크 전송과정에서 메시지가 사라지는 것을 말한다.

이것은 TCP에서 메시지가 유실되는 것을 말하는 것이 아니라, Send를 했을 때 상대방이 받았는지 받지 못했는지 알 수 없다는 얘기이다.

TCP를 썼다고해도 Send를 한게 상대방한테 도착했는지는 알수 없다.

(우리는 단지 송신버퍼의 카피만을 확인함)

**RPC 컴파일러 연동**

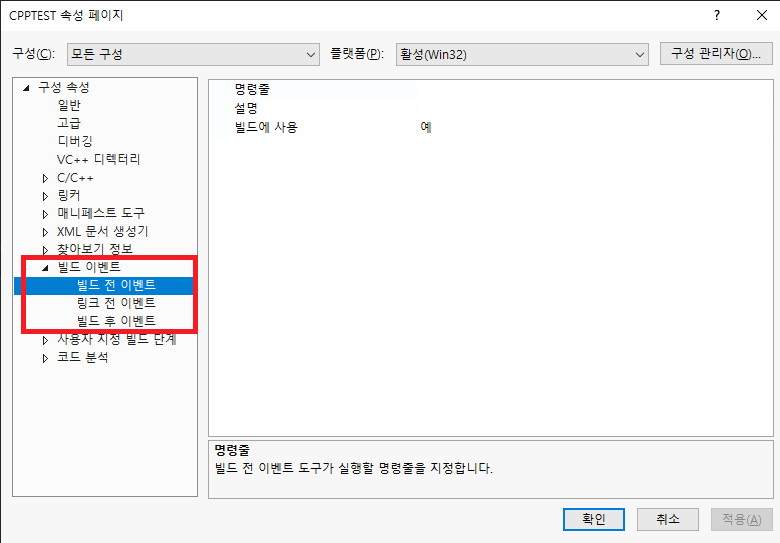
RPC클라입장에서 매개변수를 넣으면, 프록시가 메시지를 만들어 Send까지 담당한다.

이후 RPC서버쪽에서 수신받아,메시지가 분기를 타고 함수호출까지 되는것을 스텁이 담당한다.

결국 메시지를 만들어서 샌드하는 행위, 수신을해서 분기타고 함수호출시키는것 까지를 RPC라이브러리가 컴파일단계에서 자동으로 소스코드를 만들어 준다. (우리는 수동)

여기서 만든 RPC의 개념은 컴파일러에 어떻게 연동하는가?

프로젝트 속성에 아래와 같은 메뉴들이 있다.



프라우드넷 RPC가 적용되기 위해선 위항목에 프라우드넷 컴파일을 넣어야 한다.

그러면 스크립트로 짠 메시지의 정의파일을 먼저 컴파일 하게된다.

**컴파일 결과로서 CPP/헤더 파일이 나오게되고, 그다음 으로 이 파일들을 visual C컴파일러가 컴파일한다.**

위에 프라우드넷자료에 게시된 자료들은 프라우드넷에서 실제 사용한 방법을 얘기해 준거싱 아니라 초안이며, 실제 프라우드넷에서 사용하는 것은 수업시간에 보여드림.

문법대로 스크립트 파일을 짜는것으로 함수를 하나씩 정의한다.

Request, Login, Version, MemNo, Parameter….

그러면 이 함수들이 메시지로서 정의되어 소스코드부분이 자동으로 나오게된다.

만들어진 스크립터를 프라우드넷 PIDL컴파일러(프라우드넷이 붙인 이름)로 컴파일하면 CPP, 헤더 소스파일이 나오게된다.

이 소스파일에는 메시지를 생성하는 함수(Msg정의), 메시지를 보내는함수(Msg Send), 수신해서 스위치하는 함수(Recv, Switch), 수신해서 ProCall하는 함수까지 만들어져있다.

**RPC의 단점**

RPC의 약간의 단점이라고 한다면 쌍방에 똑 같은 함수가 허용되게끔 만들어 진다.

우리는 Request, Response로 클라-서버간의 메시지방향을 정의했다.

하지만 RPC의 경우 쌍방에서 원격지로 호출할 수 있는 준비가 끝나게되고, 서로에게 호출시킬 수 있는 입장이 되게 된다.

**실제 현업에서 사용되는 RPC**

실제 현업역시 하나하나 코딩하기 너무번거롭고, 실수할 확률도 올라가기 때문에 switch-case 분기까지 타고 가는 함수까지는 만든다.

RPC를 대체하는것이 프로토콜 버퍼(구글)이고, RPC라는 오픈소스 라이브러리(프로토콜 버퍼, 플랫버퍼)도 존재한다.

이 라이브러리들은 쌍방이 어떤 언어를 사용하든간에 RPC로 통일시켜주는 개념이다.(웹개발자들이 많이 사용한다.)

(프라우드넷같은 경우는 CPP와 C#만이 지원된다.)

**이러한 RPC를 우리가 적용할 수 없는 이유**

RPC의 개념이 모두 반영되려면 네트워크 송수신부가 내장되어야하고, 어떤 함수를 호출시키면 메시지를 만들어서 Send를하고 상대방이 Recv를 받아 분기를 나눠 호출해야 한다.

네트워크 송수신 파트가 내부에 들어갈수밖에 없는 구조이다.

하지만 우리는 네트워크 송수신부를 Overlapped I/O의 IOCP를 사용할 것이므로, 오픈소스로 존재하는 구글의 프로토콜 버퍼 같은 것을 사용할 수가 없다.

(설령 쓴다고해도 메시지를 생성시키는 직렬화버퍼 역할로만 사용이 가능)

NC는 자체적으로 만들며, 프라우드넷도 네트워크 엔진이므로 이런기능을 내장시키는 것이다.

**RPC 의 성능**

RPC는 당연히 우리가 직접 하드코딩하는것보다 느려진다.

모든부분이 라이브러리 입장이되고, 외부에서 기능을 확장시켜야 하므로 모두 가상함수가 될수밖에 없고, 이는 함수호출 부분에서 성능하락을 뜻한다.

하지만 직렬화버퍼를 쓴다고 하면 마샬링부분에 대한 성능하락은 우리가 직접짜도 있을 수밖에 없을 것. (Copy가 한번 더 늘어나게되는 것)

이걸직접 완성시킨다면 포폴에 넣을 가치가 있을 것이다.

**<5.1>**

직렬화 버퍼클래스를 만들때는 무엇으로 해도 크게 상관없으나, 엄밀히 따지면 L7이기때문에 클래스이름을 지을때 Packet보다는 msg가 더 어울린다

버퍼의 크기는 프로젝트의 프로토콜을 기준으로 판단할 것인데, 게임이라면 대게 1000바이트를 넘지않는다.

**최대크기 규칙을 설정해도 될것이고, 너무 가변적인 경우 직렬화버퍼의 크기를 가변으로 만들 수 있게 개량해야한다.(기본크기에서 부족할경우 사이즈를 더 확보하는 등)**

**일반적으로 C나 C++에서 사용하는 기본타입은 연산자 오버로딩하여 정의하게되며, 큐와 동일한 FIFO형식이나 1회용이기 때문에 환형으로 만들지 않는다.**

실제로 사용할때는 Recv를 받아 헤더에 기록된 길이만큼을 직렬화버퍼에 해당 길이만큼을 직렬화버퍼에 넣어주는 것이다.

(직렬화버퍼 안에는 header없이 payload만 들어있다는 전제이다.)

이후 직렬화버퍼 안에서 메시지 프로토콜 규격대로 하나씩 뽑아내는 방식이다.

실제 데이터를 생성할때 헤더같은 경우는 개발자의 편의에 따라 진행하지만, 밑에 예시는 header와 payload를 분리시켰다.

/\*---------------------------------------------------------------

Packet.

네트워크 패킷용 클래스.

간편하게 패킷에 순서대로 데이타를 In, Out 한다.

- 사용법.

CPacket cPacket;

넣기.

clPacket << 40030; or clPacket << iValue; (int 넣기)

clPacket << 1.4; or clPacket << fValue; (float 넣기)

빼기.

clPacket >> iValue; (int 빼기)

clPacket >> byValue; (BYTE 빼기)

clPacket >> fValue; (float 빼기)

!. 삽입되는 데이타 FIFO 순서로 관리된다.

환형 큐는 아니므로, 넣기(<<).빼기(>>) 를 혼합해서 사용하지 않도록 한다

\* 실제 패킷 프로시저에서의 처리

BOOL netPacketProc\_CreateMyCharacter(CPacket \*clpPacket)

{

DWORD dwSessionID;

short shX, shY;

char chHP;

BYTE byDirection;

// \*clpPacket >> dwSessionID >> byDirection >> shX >> shY >> chHP;

\*clpPacket >> dwSessionID;

\*clpPacket >> byDirection;

\*clpPacket >> shX;

\*clpPacket >> shY;

\*clpPacket >> chHP;

...

...

}

\* 실제 메시지(패킷) 생성부에서의 처리

void mpMoveStart(CPacket \*clpPacket, BYTE byDirection, short shX, short shY)

{

st\_NETWORK\_PACKET\_HEADER stPacketHeader;

stPacketHeader.byCode = dfNETWORK\_PACKET\_CODE;

stPacketHeader.bySize = 5;

stPacketHeader.byType = dfPACKET\_CS\_MOVE\_START;

clpPacket->PutData((char \*)&stPacketHeader, dfNETWORK\_PACKET\_HEADER\_SIZE);

\*clpPacket << byDirection;

\*clpPacket << shX;

\*clpPacket << shY;

\*clpPacket << (BYTE)dfNETWORK\_PACKET\_END;

}

----------------------------------------------------------------\*/

헤더는 고정이므로 구조체로 진행한다.

이런 모양새로 간다면 보낼때는 패킷(직렬화버퍼)에 헤더를 넣은 꼴이되고, 받을때는 패킷에 헤더가 없는꼴이된다.

엄밀히 따지면 사실 여기서 헤더는 없어야하지만 지금은 헤더들은 네트워크쪽에 치중한 헤더도 아니고 컨텐츠 쪽에 있는 헤더도 아니므로 경계가 애매한 상태.

(실제로 네트워크 라이브러리를 만들때는 암호화와 체크섬, 헤더셋팅까지 라이브러리가 감당할 것이다. 단, 컨텐츠 헤더 타입정도는 따로 존재해야 함.)

#ifndef \_\_PACKET\_\_

#define \_\_PACKET\_\_

class CPacket

{

public:

/\*---------------------------------------------------------------

Packet Enum.

----------------------------------------------------------------\*/

enum en\_PACKET

{

eBUFFER\_DEFAULT = 1400 // 패킷의 기본 버퍼 사이즈.

};

//////////////////////////////////////////////////////////////////////////

// 생성자, 파괴자.

//

// Return:

//////////////////////////////////////////////////////////////////////////

CPacket();

CPacket(int iBufferSize);

virtual ~CPacket();

//////////////////////////////////////////////////////////////////////////

// 패킷 파괴.

//

// Parameters: 없음.

// Return: 없음.

//////////////////////////////////////////////////////////////////////////

void Release(void);

//////////////////////////////////////////////////////////////////////////

// 패킷 청소.

// Parameters: 없음.

// Return: 없음.

//////////////////////////////////////////////////////////////////////////

void Clear(void);

Clear()는 지금당장은 쓸일이없으나 나중에 메모리풀로 사용(재사용하는 구조)할때 호출된다. **메모리풀을 사용하는 이유는 동적할당/지역변수 선언이 느릴뿐더러 여러개의 스레드에 전달하는 형식이 멀티스레드 환경을 지원하기 위함.**

**생성자 호출자안에서 동적할당이 진행되는데, 생성자/소멸자를 반복적으로 호출하는 경우는 불필요한 작업이 되기때문에 clear는 front/rear만 0으로 체크하는방식으로 가게된다.**

//////////////////////////////////////////////////////////////////////////

// 버퍼 사이즈 얻기.

//

// Parameters: 없음.

// Return: (int)패킷 버퍼 사이즈 얻기.

//////////////////////////////////////////////////////////////////////////

int GetBufferSize(void) { return m\_iBufferSize; }

//////////////////////////////////////////////////////////////////////////

// 현재 사용중인 사이즈 얻기.

//

// Parameters: 없음.

// Return: (int)사용중인 데이타 사이즈.

//////////////////////////////////////////////////////////////////////////

int GetDataSize(void) { return m\_iDataSize; }

GetBufferSize()는 Payload의 크기를 얻어주는 것.

//////////////////////////////////////////////////////////////////////////

// 버퍼 포인터 얻기.

//

// Parameters: 없음.

// Return: (char \*)버퍼 포인터.

//////////////////////////////////////////////////////////////////////////

char\* GetBufferPtr(void) { return m\_chpBuffer; }

//////////////////////////////////////////////////////////////////////////

// 버퍼 Pos 이동. (음수이동은 안됨)

// GetBufferPtr 함수를 이용하여 외부에서 강제로 버퍼 내용을 수정할 경우 사용.

//

// Parameters: (int) 이동 사이즈.

// Return: (int) 이동된 사이즈.

//////////////////////////////////////////////////////////////////////////

int MoveWritePos(int iSize);

int MoveReadPos(int iSize);

/\* ================================================================= \*/

// 연산자 오버로딩

/\* ================================================================= \*/

CPacket& operator = (CPacket& clSrcPacket);

//////////////////////////////////////////////////////////////////////////

// 넣기. 각 변수 타입마다 모두 만듬.

//////////////////////////////////////////////////////////////////////////

CPacket& operator << (BYTE byValue);

CPacket& operator << (char chValue);

CPacket& operator << (short shValue);

CPacket& operator << (WORD wValue);

CPacket& operator << (int iValue);

CPacket& operator << (DWORD dwValue);

CPacket& operator << (float fValue);

CPacket& operator << (\_\_int64 iValue);

CPacket& operator << (double dValue);

//////////////////////////////////////////////////////////////////////////

// 빼기. 각 변수 타입마다 모두 만듬.

//////////////////////////////////////////////////////////////////////////

CPacket& operator >> (BYTE& byValue);

CPacket& operator >> (char& chValue);

CPacket& operator >> (short& shValue);

CPacket& operator >> (WORD& wValue);

CPacket& operator >> (int& iValue);

CPacket& operator >> (DWORD& dwValue);

CPacket& operator >> (float& fValue);

CPacket& operator >> (\_\_int64& iValue);

CPacket& operator >> (double& dValue);

위 연산자 오버로딩을 템플릿으로 대체하는 경우 어떨지 고민해볼것.

**거의대부분이 성능을 높이기위해 프로파일러로 많은 시간이 걸린 것들을 고민하지만 호출량이 더 중요하다.** (호출량이 얼마없다면 성능을 줄여도 티가안남)

네트워크 프로그래밍을 할 때 이 직렬화버퍼 클래스가 호출량이 제일 많다.

((찾아본 결과

**1.STL을 사용하는 것만으로 빌드 시간이 늘어날 수있다.**

**프리 프로세스에서 행해지는 템플릿 해석은 빌드시간에서 템플릿 전개에 의한 코드 사이즈의 증가로 링크 시간에 지대한 영향을 끼치므로, 용법과 용량을 지키고 올바르게 사용해야한다.**

프리 프로세스 (**고급언어를 또다른 고급 언어로 번역하는 고급언어 번역기. 컴파일보다 먼저 실행되어 미리 처리하는 뜻으로 선행처리기 라고도 부른다.** 프리프로세서는 컴파일 된 바이너리 파일을 링크하는 단계가 아닌 컴파일 이전의 텍스트파일을 처리한다. 프리프로세서는 대표적으로 C언어에서의 #이 들어간 명령어가 있다.(#include:파일대치, #define 매크로 정의, #undef(매크로 해제)

추가로, 템플릿과정에서 느려지는것이아니라 템플릿타입을 쓰면 T타입으로 모든 자료형을 퉁치는 것이므로 하나하나의 전용코드를 사용하는것보다 하나로 퉁칠려면 느려지는 코드가 100퍼센트 나온다. 느낌있는 분은 감이 오셨을것.

-> 이걸 템플릿으로 사용하고자한다면 들어온 자료형타입이 어떤 자료형인지 분석해야하는데,

여기서 느려지는 것이 아닌가 생각.

-> 이렇게 하기싫다면 사용자측에서 하나하나 인자를 정해줘서 써야하는데…이렇게 못씀.;

))

//////////////////////////////////////////////////////////////////////////

// 데이타 얻기.

//

// Parameters: (char \*)Dest 포인터. (int)Size.

// Return: (int)복사한 사이즈.

//////////////////////////////////////////////////////////////////////////

int GetData(char\* chpDest, int iSize);

//////////////////////////////////////////////////////////////////////////

// 데이타 삽입.

//

// Parameters: (char \*)Src 포인터. (int)SrcSize.

// Return: (int)복사한 사이즈.

//////////////////////////////////////////////////////////////////////////

int PutData(char\* chpSrc, int iSrcSize);

protected:

...

int m\_iBufferSize;

//------------------------------------------------------------

// 현재 버퍼에 사용중인 사이즈.

//------------------------------------------------------------

int m\_iDataSize;

...

};

#endif

이때 직렬화버퍼가 모두 찼다면, 내부적으로 사이즈를 늘려준다.

(사이즈를 다시잡아서 실제 카피하는것까지) 대부분이 여기까지만 생각한다.

**여기서 동적할당하여 메모리를 확보하는것은 예측하지못한 상황을 대비한 로직이기 때문에 빈도수가 매우적어서 성능면에서 크게 이슈가 되지않음.**

때문에 메모리를 확보해서 사이즈를 늘리는것이 나쁜 선택은 아니나, 이런상황이 일어났다는 것을 알고 있어야 한다.(ex로그)

**이러한 조치를 취하지않는다면 개발자는 이와같은 상황을 전혀 눈치채지못할 것이기 때문에, 내부에서 resize가 반복적으로 호출되어 성능을 저하시킬 수도있음.**

(resize가 자주호출된다면 최대크기를 처음부터 늘려야할것)

그리고 프로토콜을 아예쪼갤지, 버퍼를 키우는걸로 할지 결정을 해야할것.

**서버의 생존(?)유무만으로는 정보가 부족하기에, 로그를 매일매일 확인할것.**

**네트워크로 메시지가왔는데, 기대한만큼 오지않았을 경우 어떻게든 바깥에 알려주지않으면 쓰레기값을 정상적인 값으로 판단하고 로직이 돌게 될 것이다.**

(**TCP에서 오지않은게 아니라 클라쪽에서 실수로 잘못 보낸 상황.** 이와같은 상황은 디버깅할때 다 잡아낼것이지만 가변적인 프로토콜이라면 라이브서비스 중에도 나타날수있음. 10개를 준다고했는데 9만 온다면 나머지 하나에는 쓰레기값이 들어가게될것. 이경우 모르고 그냥 넘어간다.)

Queue에서는 Dequeue를 시도할때 공간이없는 경우 상황자체를 return값으로 알려줬으나 직렬화 버퍼의 경우는 리턴이나 파라미터로 결과를 알려줄 방법이 없다.

이때 선택할 수 있는 방법.

**1. WSAEGetLastError와 같이 errcode flag를 둔다.**

어디가 문제인지 알면 더좋지만 뽑는데 문제가있는것만 알려줘도 상관없다. 멤버에 에러코드를 넣어놓고 뽑기를 시도한다음에 에러가 존재하는지 바깥에서 체크하도록 하는것.

**단, 이 경우 300개가 넘는 proc에 계속해서 똑같이 확인하는 코드가 들어가야하므로, try catch구문을 활용하면 용이하다.** (단순 throw는 위 방식과 의미없음)

**try catch의 가장 큰 특징중 하나는 Stack Unwinding으로, 콜스택을 뛰어넘어서 나간다는 점이다.** 이게 호출되는 구간은 어딘가에서 하나의 패킷프로시져 분기가 나오는 구조일수밖에 없으며, (함수 포인터 배열, switch case) 이 구조에 try catch가 한번에 들어가게 될것이다.

이안에서 문제가 발생된다면 튀어나갈것. 우리가 실제 코드를 어딘가에서 recv함수를 호출을 했으면

recv(....)

RecvQ.enque..

헤더검사

CPacket Packet; (직렬화버퍼 선언)

RecvQ -> Packet 완성된 메시지를 복사

try

{

PacketProc(헤더.type, &Packet);//이 함수를 try catch문으로 묶어버리는것.

{

Switch(..) //여기서 정보를 로그남김

}

}

catch(CPacket::Ex )

{

}

catch(...)는 모든 객체 throw를 다받겠다는뜻.

우리는 직렬화버퍼의 Output(꺼내오는 경우)하다가 발생할 예외만 받을 것이므로, 우리만의 예외객체를 하나 만들어야 한다.

**강사의 경우에는 인풋용 아웃풋용이 따로 존재하긴하지만, Input에서 buffer를 resize한다면 예외가 발생할 일이 없을 것.**

**반대로 최대메시지 길이를 정해놓고 그걸 넘었다면 Input예외를 던지는 방법도 상관없음.**

dequeue하다가는 패킷 output용의 어떤 예외객체를 던질 것.그럼 여기는 해당 타입만 받아내면된다.(이 역시 디버깅에 어려움이 있다)

최대한 분석하기 쉽게끔 고민해볼것.

**나중에는 간단한 암호화/복호화와 체크섬까지 추가한다. 이 경우 클라가 실수로 보내는게 아닌이상 누군가 의도적으로 조작을 해서 메시지 보낸것이다.**

임의의 클라이언트를 만드는 과정에서 메시지프로토콜을 잘못분석해서 잘못보내는상황이 흔한 경우는 아니므로, 이런로그가 수도없이 나와서 이걸 분석하느라 걱정할 필요는 없다.

직렬화버퍼를 보면 데이터를 만들때는 문제없지만 뺄때는 번거로워 보인다.

**구조체의 경우 버퍼를 구조체형태로 캐스팅하여 쓰면 그만이지만, 직렬화버퍼의 경우에는 로컬지역변수를 하나씩 만들어서 하나씩 카피해서 뽑은다음에 사용하므로 결과적으로는 카피가 1회 늘어나는것이다.**

따라서 직렬화 버퍼는 구조체보다 오히려 성능이 떨어지고 불편하다.

하지만 직렬화버퍼의 장점은 (rpc의 개념을 들어가기 직전까지 개념), 위의 예제에는 연산자 오버로딩의 대상을 기본타입으로만 넣었다.

**이 type을 객체단위로 가게되는것.(Character, Item...)**

**따라서 객체내부에 인자들을 하나씩 빼서 보내는것이 아니라, 그냥 아이템을 보내는것이다.**

채팅 메시지를 예시로 든다면,

**채팅메시지를 보내고싶다면 문자열을 보내야하므로 프로토콜 설계시 길이가 들어가야한다.**

((

C언어에서는 NULL-TerminateString을 사용하고있다. 이 방법은 포인터의 개념으로 길이없이 문자열을 개념을 만들기위해 C언어가 선택한 방법으로, 일반적인 방법은 아니다.

어떤언어도 NULL-Terminate스트링으로 표현하지않는다. C언어에만 포인터가 존재하기때문에..

))

결국 채팅메시지도 메시지의 길이, 문자열의 길이와 문자열을 보내야한다.

만약에 이렇게 채팅메시지를 만드는 어떤 캐릭터형태 포인터의 문자열이 들어왔다고 치자.

채팅메시지를 직렬화버퍼에 뽑는 메시지를 만든다고 치면?

ChatMessage(\*Packet, char\*str)

Short Len = strlen(str);

Packet << Len; //길이를 먼저넣고

Packet << char\* //문자열을 넣는다. 버퍼니까 PutData를 써야함.

위와같은 형태가 될것이다. 이는 뽑을때도 마찬가지인데, **두번씩 넣고 빼기 번거롭기 때문에 직렬화버퍼 클래스에 char\*를 오버로딩 해둔다.** char\*포인터가 오면 문자열이라고 인식하고 가는것. (누군가 캐릭터 형태 포인터의 주소값을 보낸다는 것은 말이안되므로...문자열로 인식.)

때문에 위 구문이 클래스 내부로 들어가게되는것이다. 이러한 형태로 확장시키는것..

**결국 직렬화버퍼의 장점을 정리하자면, 변수를 하나씩 얻어 쪼개 넣는 구조가 아닌 컨텐츠에 있는 객체를 밀어넣는 것으로 대체가 가능하게 포장하는것이다.**

하지만 이 객체를 직렬화버퍼는 알지못하므로 상속받아서 해당 컨텐츠 그게임의 필요로하는 타입들을 연산자 오버로딩을 통해 추가하는것이다.

(보통 문자열정도는 기본형에 넣는것을 허용하는데, 이형태는 우리가 정한것이다. 어떤곳에서는 1byte, 또 다른곳에서는 4byte로 하고싶을때도 있음. 이것은 문자열을 표현하는 규칙자체도 어떤 컨텐츠가 존재하는 서버-클라 프로토콜중 하나이므로, 부모는 기본타입만을 가진다.)

실제 프라우드넷도 이와같은 형태이다. 기본메시지라는 클래스에는 기본타입만 있고 필요한게 있다면 상속받아서 마음대로 넣으라는 뜻이다. 여기서 좀더 생각해본다면...

아이템이라는 객체를 전달하기위해 다음과같이 정의했다고 가정한다.

CItem

ID

Type

Attack1 (4개의 능력치)

Attack2

Attack3

Attack4

직렬화버퍼에 연산자 오버로딩으로 아이템을 던지면 내부 요소들을 하나씩 뽑아서 넣는걸 만들어놨다고 가정한다.

**이때, 아이템의 목록을 보내고 싶다면?**

**1.아이템이라는 리스트가 있을때 (list<CItem\*>) 직접 순회하면서 넣는 것.**

**더 나아가서, 아이템 list타입자체를 오버로딩 받아 바깥쪽에서 좀더 수월하게 코딩하는것도 가능하다.**

이는 프로토콜이 변경됐을때 특정 클래스에 캐릭터를 생성할때 어떤 요소를 추가해야 하는경우에도 오버로딩한 부분만 바꾸면 캐릭터만 생성하는 경우도 모두 바뀌게 될것.

**이런식으로 진행될때, 퀘스트목록, 선물함목록 등등 다양한 것을 보낼때 받는쪽에서는 용량이나, 어떤 형태인지 전혀 알수 없게 되지만,** (처음 설계하고 만든사람은 알겠지만 게임이 오래되어 사람이 바뀔경우 그냥 이렇게 쓰면 간다라는것만 알것.) **요즘은 대부분 이러한 형태로 사용한다.**

그래서 지난번에 스크립트로 프로토콜 정의를하고 RPC의 개념으로 Send파트와 분기파트와 Recv파트까지 만들어주는것까지 설명을 했던 것.

NC같은 경우는 RPC까지는 구현하지않고 오버로딩 까지는 자동화시킨다.

**우리는 프로토콜을 설계할때 먼저 아이템의 정보, 캐릭터의 정보 이동의정보를 먼저 define을 걸고,** (define이라기 보다는 스크립트차원에서 먼저 정의를 내려놓는것이 정확한 표현) **실제 프로토콜에서는 아이템을 보냅니다, 테스트를 보냅니다 라고 해결을 본다. 이때 이 부분에 해당하는 연산자 오버로딩 파트의 코드들은 자동으로 만들어지는것으로, 이를 rpc의 전단계까지 가는것으로 본다.**

rpc의 경우 네트워크 송수신부분까지 다 묶여야되기 때문에 약간 애매하므로, 전단계까지 자동화하는것.

네트워크 엔진이라는 입장에서는 그 취지에맞게 묶어 버릴수있겠지만 보통 네트워크 엔진/ 라이브러리 따로 있는 상황에서는 rpc의 개념까지는 가지않고 분류한다.

**따라서 보는시각에따라 그래서 패킷생성기 정도로 볼수도 있다.**

NC의 리니지2의 경우 Jason이라는 테스트 포맷이 있다. (이는 Key-Value형태로 이루어져 있으므로 크게 어렵지않음. )

**이와같은 것들로 프로토콜을 정의한다. 구조체설계하듯 설계하면 자동으로 넣고빼고할수있게끔 코드를 생성시켜 주는것이다.**

**이는 분명히 비효율적인 부분이 생길 수 밖에 없는 구조이다.단순히 코드용량을 모르는것을 떠나서, 불필요한것까지 더 보내는 형태가 된다.**

//책갈피\_001

구체적으로는, 소유자한테는 해당 아이템의 모든 정보를 보내줘야하는 상황과 다른 유저가 장착한 아이템을 가볍게 확인할 수 있을 정도로 들고 있는 경우 똑같이 처리하게되면 불필요한 것까지 보내는 형태이다. (이와같은 상황을 위해 세분화한다면 코드가 너무 복잡해지므로, 대부분은 1,2개로 모두 커버한다.)

이렇게되면 불필요한 데이터가 같이 번갈아서 들어가게된다. 근데 이거를 보내는사람도 잘모르고, 받는사람도 잘모르는 상황(위에 언급한)이 만들어 질 수 있다.

**구조체도 옛날방법이긴 하지만 단점만 존재하는것이 아니기 떄문에 여전히 현역적인 방법이다.**

일단은 직렬화버퍼를 만들고, 타입별로 컨텐츠의 어떤 객체들을 상속받아 자동으로 큐잉하는것까지는 안해도 상관없다. 경우에 따라 다르지만 단계만 많아지고 카피만 많아져서 더 복잡해질 수 있다.

타입기준으로 해서 간단한 예외처리 할수있을때까지만 진행하고, 간단하게 그냥 넣고빼고 하는것을 테스트해보고 어느정도 완성이 됐다 싶으면 우리클라를 싸그리 교체하는것이다. 그리고 아무문제없다면 믿고 진행한다.

네트워크 프로그래밍시 직렬화버퍼 클래스가 호출량이 제일 많으므로, 성능면에서 많은 고민을 해봐야 할것.

**<알고리즘 진행후 다시복귀>**

**<5.29>**

**앞으로의 진행방향**

예전 TCP Fight 3D game을 스크롤이 되는 형태로, MMORPG에서 필드처리하듯이 하는것으로 가게된다. 이때 채팅서버랑, 친구관리하는 간단한 서버랑 간단한 것을 만들어 볼텐데 이동안은 Select모델로 검증된 코드를 만들어야한다.

**이때 네트워크 오류가 난다면 매우 복잡해지기 때문에, 제일 마지막 TCP FIGHT서버를 만들떄는 네트워크파트에 대한 오류는 없다고 기대하고 가는것이다.**

**단, Select모델은 모듈화가 깔끔하게 나오지 않기 때문에 클래스화는 하지않는다.**

전역함수, 전역배열로 진행하고 이 코드를 최대한 재사용하여 컨텐츠만 바꿔 진행하는 방향으로 간다. (스트레스 풀을 만들어서 검증을 한번 거칠 것)

**결국 나중에 네트워크 파트만 복붙하여 게임서버의 컨텐츠를 가져다 붙이는 형태로 진행을 하겠다는 얘기.**

**채팅서버 예제 진행방향**

채팅서버에는 로비, 방, 방에 있는 유저, 채팅상대 유저, 유저의 방생성, 방파괴등의 컨텐츠가 있다. Select모델에 대한 완벽한 검증 코드를 만드는것도 목적이긴 하지만, 컨텐츠에 대한 연습이기도 하다.

실제 현업에서는 컨텐츠를 만드는 일이 더 많을 것이다.

**클라이언트를 Console로 만들어서 채팅을 할 경우에는 키보드 입력함수가(scanf\_s, get\_s) Block 함수이기 때문에, 아예 네트워크 코드가 돌지 못한다.**

기본적으로 네트워크는 네트워크대로 돌아야하고, 키보드입력은 키보드 입력대로 받아야하므로 멀티쓰레드로 해결을 하던가, Win API기반의 Async Select모델에서 키보드입력을 editer console을 사용하여 만들어야 한다.

(클라의 경우 Win API에서 시간이 너무많이 쓰이기 때문에 생략한다)

**채팅서버 컨텐츠 규칙**

- 로비창에는 방목록 존재.

- 방을 만들면 다른 유저에게도 방이 전달되어야 한다.

- 로그인은 존재하지않고 닉네임만을 입력한다.(닉네임은 중복X)

- 방 더블클릭하면 방에 입장된다.

- 채팅창을 닫으면 방에서 나가게되고, 마지막 방에 있는 사람이 나가면 방이 삭제된다.

(위 규칙으로 인해 방을 만들고 아무도 들어가지않으면 방은 삭제되지 않는다.)

위 사항들은 기획의 문제이지만 제공되는 클라와 규칙을 맞춰야 하기 때문에 위와 같은 규칙을 지켜야 한다.

**PacketCode / EndCode.**

TCP FIght만들 때 사용했던 EndCode는 언급했듯 사용하지않는다.

맨앞에 1byte를 넣는 PacketCode는 포함하여 진행하지만 일반적인 상황에서는 사용하지않는다. **PacketCode의 목적은메시지의 쓰레기 데이터인지를 판별하기 위함이나, 체크섬을 사용하여 패킷이 위변조가 되었는 확인하는 동시에 위와 같은 판별을까지 커버가 가능하다.**

(이러한 쓰레기 데이터가 오는 상황이 자주 일어나는것도 아니므로..)

**(이때 의도적으로 패킷을 위변조 하는것과 도중에 패킷이 손실된 것과는 다른얘기이며, 패킷안에 데이터가 깨졌는지 확인하기위해 체크섬을 사용한다는 것은 잘못된 답변이다.)**

(데이터가 깨졌냐 안깨졌냐(노이즈, 랜선, 열화)는 걱정하지않는다. 애초에 아래 계층에서 해결이 끝났기 때문이다. UDP에도 체크섬이 포함되어있음.)

**더나아가서, 패킷코드도 일치하고 체크섬규칙도 일치할 경우에도 컨텐츠 차원에서 정상적인 요청인지 한번더 검증을 거쳐야 할것이다.**

**requeset와 response**

**request : 클라이언트가 서버에 정보를 요청하는 것.**

**response : 서버가 클라이언트에 보내는 것.**

(TCP Fight의 CS, SC에 해당한다)

단, response가 꼭 요청된 응답에 해당하는 것은 아니다.

프로토콜 설계 자체가 무조건 클라가 먼저 요청을하면 그 결과를 주는 서버일 경우(ex 웹)에는 requset / response 개념이 정확히 일치할 것.

stateful서버에서는 클라의 요청이 없더라도 서버가 먼저 보내야 하는 경우가 존재하므로,

**request와 response의 개념이 100% 맞는것은 아니므로, 여기서는 단지 방향을 따지는데 그치게 된다.**

**문자열 전송**

문자열을 보낼 때 어떻게 보낼것인가?

**채팅메시지의 경우 가변길이 이므로, 프로토콜은 [문자열 길이 + PayLoad(문자열)]이 된다. 여기에는 길이가 포함되었으므로 당연히 NULL문자가 들어가지않는다.**

ID같은 경우는 일반적으로 고정길이로 간다.

**고정길이일 경우 길이가 포함되지않으므로 끝이 어딘지 알수있게끔 NULL문자가 포함되어야한다.**

**채팅서버 프로토콜 설계**

제일먼저 프로토콜을 설계한다.

//--------------------------------------------------

// 채팅 프로토콜

//

//

//--------------------------------------------------

#define dfNETWORK\_PORT 6000

#define dfNICK\_MAX\_LEN 15 // 유니코드 글자 길이 NULL 문자 포함

#define dfPACKET\_CODE 0x89

//------------------------------------------------------

// 패킷헤더

//

// | PacketCode | CheckSum | MsgType | PayloadSize | \* Payload \* |

// 1Byte 1Byte 2Byte 2Byte Size Byte

//

// checkSum - 각 MsgType, Payload 의 각 바이트 더하기 % 256

//

//------------------------------------------------------

**위 체크섬에 PayLoadsize는 포함시키지 않고, MsgType과 PayLoad를 바이트 단위로 더해서 이렇게 한 값에 %256을 해서 체크섬에 넣을 것이다.**

단순한 규칙일 뿐 PayLoadSize를 체크섬에 넣지 않는 큰 의미는 없다.

#pragma pack(push, 1) //**무조건 구조체로 네트워크 메시지 선언할때는 pragma pack을 간다.**

struct st\_PACKET\_HEADER

{

BYTE byCode;

BYTE byCheckSum;

WORD wMsgType;

WORD wPayloadSize;

};

#pragma pack(pop)

**클라가 제일처음 하게되는 것은 로그인이다.**

//------------------------------------------------------------

// 1 Req 로그인

//

//

// WCHAR[15] : 닉네임 (유니코드)

//------------------------------------------------------------

#define df\_REQ\_LOGIN 1

//------------------------------------------------------------

// 2 Res 로그인

//

// 1Byte : 결과 (1:OK / 2:중복닉네임 / 3:사용자초과 / 4:기타오류)

// 4Byte : 사용자 NO

//------------------------------------------------------------

#define df\_RES\_LOGIN 2

#define df\_RESULT\_LOGIN\_OK 1

#define df\_RESULT\_LOGIN\_DNICK 2

#define df\_RESULT\_LOGIN\_MAX 3

#define df\_RESULT\_LOGIN\_ETC 4

**사용자에 대한 구분은 고유번호를 사용하며, (닉네임인 string을 들고 다니며 검색할 수는 없기 때문) 서버에서는 이 사용자 고유번호를 모두 기억해놓고 있는다.**

**이때, 로그인이 성공했을 경우 방목록을 받도록 로직을 결합시키지 않아야 한다.**

**이런형태로 로직을 결합시키면 구조가 매우 복잡해지므로, 독립적으로 구성하여, 로그인이 완료되면 클라쪽에서 방목록을 요청하게끔 만들어주면 되는 것.**

(이렇게 독립시키는것은 다른곳에서도 통용되는 개념.)

**추가로, 우리가 프로토콜을 설계할때는 실패라는 결과만으로는 부족하기 떄문에, 우리가 판단할 수 있는 모든상황들에 대해서 최대한 상세하게 모든 타입을 지정해 둔다.**

따라서 (4. 기타오류)는 사용할수도, 안할수도있음.

//------------------------------------------------------------

// 3 Req 대화방 리스트

//

// None

//------------------------------------------------------------

#define df\_REQ\_ROOM\_LIST 3

**대화방 목록을 요청할때, 들어갈 데이터가 없으므로 PayLoad가 0이되고, 헤더만 송신된다.**

//------------------------------------------------------------

// 4 Res 대화방 리스트

//

// 2Byte : 개수

// {

// 4Byte : 방 No

// 2Byte : 방이름 byte size

// Size : 방이름 (유니코드)

//

// 1Byte : 참여인원

// {

// WHCAR[15] : 닉네임

// }

//

// }

//------------------------------------------------------------

#define df\_RES\_ROOM\_LIST 4

방 역시 닉네임과 같이 방의 고유번호가 존재하며, 가변길이가 되어야 한다.

**우리는 직렬화버퍼를 쓰는 방식을 택했으므로 메시지 하나에 끝낸다.**

**따라서 방개수부터 들어가서 [ 방이름 / 방번호 ]로 들어가게 된다.**

참여인원은 1byte로 넣고 그다음에 닉네임이 들어가는 형태로 설계되었다.

**이때, 방이 엄청나게 많이 나와서 10000바이트가 된다고 한다면 생길 수 있는 문제는?**

**1. 직렬화버퍼에 넣게 될것인데, 이때 직렬화버퍼가 고정길이 었다면 문제가 생길 것이다.**

**고정길이를 초과했을 때 동적으로 길이를 늘리는 방식을 택했다라면 일단은 직렬화버퍼 상에서는 문제는 없다.**

2. **그다음 생길수있는 문제는 Send용링버퍼에 인큐가 안되는 상황. 우리는 당연히 Send용 링버퍼를 충분하게 잡아놔야 할 것.**

(TCP는 자기가 알아서 끊어서 보내기 때문에 L4계층에서는 전혀 문제가 없음.)

**3. 결국엔 페이징 방식을 사용하는 방법을 선택한다.**

이는 개수에 대한 MAX를 정해 보내는 방법으로, 일방적으로 서버측에서 n개로 끊어 보낸다고하면 클라측에서 문제가 생길 수있으므로 추가적으로 논의가 되어야 한다.

**3-1. 게시물 웹페이지처럼 단순히 방목록을 요청하는 것이 아니라, 몇 페이지 방목록을 요청하는 형식으로으로 사용.**

**이 경우 클릭하는순간 (n)m/s안으로 들어오기 때문에 사용자 측에서 딱히 느리다고 생각하지는 않는다.**

**3-2. 스크롤바 형식일 때**

이 경우 일단 클라에서 방목록이 다 올라와야한다. 방목록을 20개씩 끊어서 페이지 번호를 부여해서 보여주는 것으로, 클라입장에서는 다음페이지가 온 것을 인식하고 add를 계속하면 될것이다.

**이경우는 클라이언트의 행동과 방목록의 추가가 비동기로 설계가 되어야 할것이다.**

요즘은 유니티자체가 설계를 비동기 자체로 해야되기 때문에 상관없으나 옛날 Win API 자체를 이렇게 만들었다고 한다면 방목록 갱신을 다받고나서 갱신하는 설계는 애초에 막았어야한다.

(방목록이 5번와서 추가했는데, 잠깐사이에 6번째 것이 오면 또 추가해야한다.)

이외에도 방법은 여러가지가 존재하며, 적절한 프로토콜로 꼼꼼하게 설계를 해야할 것이다.

우리 헤더설계상 표현범위가 2byte이므로 65535이 PayLoad의 최대크기 일것이다.

TCP/UDP에도 길이가 2byte로 존재하나, MTU(1500)사이즈 문제 때문에 실상 MTU사이즈 이상으로 보내지 않는다.

우리도 마찬가지로 최대 65535까지의 크기를 보낼수는 있지만, 지금은 그렇게 크게 보내지않는다.

**현재 제공된 클라의 직렬화버퍼는 512으로, 그이상으로 보내면 그냥 누락된다.**

때문에 위 클라는 방을 계속 만드는것에 대해 안전장치가 없는 상태이나 실제 상용화를 위한 채팅서버를 만든다면 위와같이 다양한 방안을 생각해봐야 할 것.

클라가 서버로부터 방생성 완료 response를 받았으므로 이제 대화방 생성이 될 차례.

이경우 방이름만 들어가면되므로, 그래서 가변길이로 방에대한 제목길이와 방에대한 제목이 들어온다.

//------------------------------------------------------------

// 5 Req 대화방 생성

//

// 2Byte : 방제목 Size 유니코드 문자 바이트 길이 (널 제외)

// Size : 방제목 (유니코드)

//------------------------------------------------------------

#define df\_REQ\_ROOM\_CREATE 5

방마다 고유번호가 있기 때문에 사실 방이름이 겹쳤다고해서 문제될 것은 없으나, 중복처리를 해도 상관없다. 그리고 방제목이 너무 긴경우도 예외처리가 있으면 좋을 것.

이전에 방목록이 엄청나게 많을 경우를 고민하긴했지만 **서버입장에서는 현재 플레이하는 사람들의 원할한 서비스를 위해서 입장제한을 시켜야 한다. 이는 컨텐츠의 제한으로, 이러한 규칙은 우리가 직접 선택하는 것.**

(지금 우리가 확정해서 만들어야하는 것은 프로토콜 규격맞추는것이고, 그외에 컨텐츠나 코드는 개발자가 적당히 알아서 결정하는 것.)

**현재 대화방생성은 두가지의 목적을 가지고있다.**

**1. 방 생성을 요청한 사람한테 요청성공을 알려주는 것**

**2. 다른유저들한테 새로운 방이 생겼다고 알려주는 것**

**메시지크기를 낮추기위해 이러한 종류를 분류하는것도 상관없다.**

방생성에 대한 response는 그 response대로가고, 다른유저들 한테 새로운 방이 생긴 것은 따로 처리하는 것.(성공실패와 방번호만 가면된다. 방이름은 알고있으니까.)

다만 이 경우 클라쪽에서 상당히 귀찮아 질 수도 있기 때문에, 잘 협의하여 선택할 것.

(내가 방금 요청했던 방의 이름을 보관해놨다가 결과가 비교하여 해당 방이 생성이 된 것을 판단하기 때문에 까다로워 진다. 방생성 버튼을 눌렀는데 결과오기전에 방생성을 또 누르는 경우 이전의 결과가 도착했다면 당연히 이전 것이 맞겠지만 덮어써지기 때문에 이걸 보관해야하고, 이를 또 리스트로 보관하게된다. 때문에 이걸 사전에 막기위해 결과가 오기전에 방생성 버튼을 막아버리는 작업이 들어가야한다. 이경우 결론적으로 로직이 복잡해지는 것이기 때문에…)

결국 프로토콜에 대한 초안은 서버개발자가 잡고, 이후에 클라개발자랑 논의하면서 보정을 잡아 적절한 합의점을 찾아야 할것이다.

지금상황에서는 요청자에게 보낼 메시지와 다른 사용자에게 갈 메시지를 구분하지 않는다.

위에있던 메시지들(로그인..)은 뭐든지 requeset를 한 유저한테만 response가 유니캐스트로 가게되지만, 방생성을 했다면 broadCast로 가는 것.

//------------------------------------------------------------

// 6 Res 대화방 생성 (수시로)

//

// 1Byte : 결과 (1:OK / 2:방이름 중복 / 3:개수초과 / 4:기타오류)

//

//

// 4Byte : 방 No

// 2Byte : 방제목 바이트 Size

// Size : 방제목 (유니코드)

//------------------------------------------------------------

#define df\_RES\_ROOM\_CREATE 6

#define df\_RESULT\_ROOM\_CREATE\_OK 1

#define df\_RESULT\_ROOM\_CREATE\_DNICK 2

#define df\_RESULT\_ROOM\_CREATE\_MAX 3

#define df\_RESULT\_ROOM\_CREATE\_ETC 4

**대화방 생성이 성공했다면 방에 대한 번호가 가갈것이고, 실패했다면 응답이 없어도 된다.**

**이는 직렬화버퍼를 사용했을 때 장점이 발휘되는 것으로, 상황에 맞춰서 보낼까 안보낼까 결정할수 있는것이다.**

방 생성이 실패했다면 결과는 요청자한테만 가는 것이다.

떄문에 ‘수시로’라는 표현은 내가 요청한적이 없지만 올수 있다는 뜻이다.우리는 어차피 지금 클라를 만드는 입장이 아니기 때문에 클라쪽 설명인 이 문구는 무시해도 된다.

//------------------------------------------------------------

// 7 Req 대화방 입장

//

// 4Byte : 방 No

//------------------------------------------------------------

#define df\_REQ\_ROOM\_ENTER 7

**대화방 입장 메시지가 왔다면 먼저 로그인했는지 확인해야 한다.**

(로그인 요청이 왔을때는 요청자가 로그아웃 상태인지 확인)

위와 같은 상황을 하나씩 확인해야 한다.

**만약 절차에 맞지않는 행위를 했을경우 해당 클라는 서버에서 끊어야 한다.**

(로그인 하지않았는데 방생성을 하는 등)

구체적인 상황으로는 채팅메시지가 온 경우 해당클라가 방에있는지 검사. 방에 있다면 로그인 했는지 검사된 것일 것.

**이런식으로 어떤 컨텐츠가 있을 때, 해당 컨텐츠에 들어올수 있는 조건만 확인한다.**

//------------------------------------------------------------

// 8 Res 대화방 입장

//

// 1Byte : 결과 (1:OK / 2:방No 오류 / 3:인원초과 / 4:기타오류)

//

// OK 의 경우에만 다음 전송

// {

// 4Byte : 방 No

// 2Byte : 방제목 Size

// Size : 방제목 (유니코드)

//

// 1Byte : 참가인원

// {

// WCHAR[15] : 닉네임(유니코드)

// 4Byte : 사용자No

// }

// }

//------------------------------------------------------------

#define df\_RES\_ROOM\_ENTER 8

#define df\_RESULT\_ROOM\_ENTER\_OK 1

#define df\_RESULT\_ROOM\_ENTER\_NOT 2

#define df\_RESULT\_ROOM\_ENTER\_MAX 3

#define df\_RESULT\_ROOM\_ENTER\_ETC 4

**이때 결과는 성공/실패(방번호 존재하지않음)/인원초과/기타로 나뉘어져있는데**

(인원 초과에 관해서는 예외처리가 되어있지 않으나, 시간남으면 진행해볼 것.)

**기타오류 같은 경우 방이 로비에 존재하지않는다거나, 방에 이미 들어가 있다거나, 아니면 로그인을 안했다거나 등의 오류를 현재 로비에 있는가? 라는 예외처리 하나로 커버하는 것이다.**

**여기서 OK가 떨어진경우, 이제 대화방의 번호, 참가인원이 추가로 전달될 것이다.**

(사실상 방 고유번호는 클라가 요청한 정보가 있으므로 필요 없다. 이는 프로토콜을 설계할 때 적절히 선택을 해볼 것. 일단 여기서는 보내주는 형태를 취한다.)

방번호를 안다는 것은 방제목을 안다는 뜻이므로, 어떤방에 입장하는지 까지는 (클라는 목록을 다 들고있기 때문) 안줘도 된다.

**참가인원 같은 경우는 표현크기가 1byte이므로 결과적으로 255명으로 제한이 걸린 것.**

참가인원에 대한 각각의 닉네임과 사용자 번호를 줄 것이다. 채팅을 할 때는 닉네임을 주고받는게 아니라 몇번(고유번호)이 채팅, 메시지등을 하는것으로 보기 때문에 불필요하게 닉네임을 가지고 다니지 않는다.

((

**회원번호 정보 은닉**

보통 회원고유번호라는것은 가입할 때 확정되는 것이다. (절대 바뀌지않음)

**이때 대부분의 서버 개발자가 이 회원번호를 감추고싶어 하기 때문에 서버에서만 회원번호를 알게끔 하고 현재 접속에 대한 번호(일회용)를 따로 구현을 하는것이다.**

하지만 이 번호는 노출되어도 딱히 문제는 없을 것 같다는 것이 학원쪽 생각.

어차피 고유한 ID랑 동급인데 ID는 노출이되고 회원번호는 불안하다는 것이 앞뒤가 맞지 않는 것 같음.

하지만 이런 경우가 많기 때문에 사내원칙을 따라간다.

내 고유번호를 서버에 보내서 다른사람의 데이터를 조작할 수 있다라는 상황자체가 말이되지 않는다. 서버는 이 해당 클라가 누구인지(고유번호가 뭔지) 다 알고있기 때문에 클라가 서버한테 메시지를 보낼 때 굳이 내가 누군지(세션) 알려줄 필요 자체가 없기때문이다. (이는 로그인이 존재하는 이유이기도 하다.)

))

//------------------------------------------------------------

// 9 Req 채팅송신

//

// 2Byte : 메시지 Size

// Size : 대화내용(유니코드)

//------------------------------------------------------------

#define df\_REQ\_CHAT 9

채팅송신을 요청했는데 해당클라가 방에 입장되지 않은 상태라면?..

**이는 예외처리를 했음에도 일어나서는 안되는 상황이 되어버린것으로 정상적인 클라라고 판단하지않고 끊어버린다.**

//------------------------------------------------------------

// 10 Res 채팅수신 (아무때나 올 수 있음) (나에겐 오지 않음)

//

// 4Byte : 송신자 No

//

// 2Byte : 메시지 Size

// Size : 대화내용(유니코드)

//------------------------------------------------------------

#define df\_RES\_CHAT 10

내가 채팅을 보내고 다시금 나한테도 돌아오게 할수도 있는것이고, 나한테 오지않고 다른애들한테만 가게 할 수도 있을 것이다.

이는 클라랑 상의해서 진행해야할 문제이다.

이 예제에서는 자기것은 자기가 창에 띄우고, 굳이 자기자신한테는 다시 보내지않는다.

**클라이언트는 송신자에 대한 고유 번호가 오면, 자기 방에있는 사람들 목록에서 찾아서 그 닉네임을 붙여서 스트링(채팅)을 써줄 것.**

//------------------------------------------------------------

// 11 Req 방퇴장

//

// None

//------------------------------------------------------------

#define df\_REQ\_ROOM\_LEAVE 11

여기서는 클라가 다이얼로그를 닫으면 방퇴장을 요청한다.

요청이오면 서버에서는 퇴장을 진행한다.

//------------------------------------------------------------

// 12 Res 방퇴장 (수시)

//

// 4Byte : 사용자 No

//------------------------------------------------------------

#define df\_RES\_ROOM\_LEAVE 12

방퇴장의 결과전달 유무도 만들기 나름이다.

해당 예제의 구조라면 다이얼로그를 닫는것으로 방퇴장을 요청하기 때문에 결과의 유무에 상관없이 퇴장시켜 버린다.

단, 다른 유저들한테는 퇴장한 유저를 삭제해야 되기 때문에 반드시 보내줘야한다.

이 예제의 설계상으로는 자기자신한테도 보냄.

//------------------------------------------------------------

// 13 Res 방삭제 (수시)

//

// 4Byte : 방 No

//------------------------------------------------------------

#define df\_RES\_ROOM\_DELETE 13

이 예제에서는 방삭제 요청이없다. 서버입장에서 판단해서 삭제해야 겠다면 그냥 삭제하고,

전체 유저한테 뿌린다. 방에대한 목록(방에대한 유저, 방의 생성과 삭제..)은 무조건 전체유저를 대상으로하며 소켓이 disconnect 된것도 나간걸로 처리한다.

**연결이 끊어져 버렸다면 해당 세션이 어떤 행위 중이었는지 찾아내서 정리해야 한다.**

**구체적으로는, 유저가 방에서 퇴장하는 함수를 만들어 방에서 퇴장할때도 호출할 것이고, dissconnect가 됐어도 얘가 들어있던 방에서 얘를 호출하는것이다.**

**추가로, 항시 언급했듯 게임에서는 정상적인 종료요청과 비정상종료를 구분하지않는다. 만들 이유도 없음.**

//------------------------------------------------------------

// 14 Res 타 사용자 입장 (수시)

//

// WCHAR[15] : 닉네임(유니코드)

// 4Byte : 사용자 No

//------------------------------------------------------------

#define df\_RES\_USER\_ENTER 14

클라들한테 주는 것. 서버입장에서는 들어온애가 당연히 누군지 아는것이고. 클라들한테 이방에 이 닉네임을 가진 사람이 들어왔다고 쫙 뿌려주는것이다.

문서로도 이미 끝난것이긴한데. 추가적으로 텍스트 설명도 들어간다.

# 방 개념이 들어간 채팅 프로그램 만들기.

클라이언트 입장에서의 대표 기능.

- 로그인

- 방 생성

- 방 입장

- 채팅

- 방 나가기

그 외의 기능은 protocol.h 파일의 프로토콜 참조.

\* 사용자 No : 서버에서 클라이언트 접속 시 마다 부여하는 고유 번호.

\* 방 No : 방 생성시 서버가 부여하는 방 고유 번호.

# 개발조건

0. 프로토콜 무조건 지키기. 임의로 프로토콜을 고치거나 추가하지 않는다.

1. 클라이언트 : 윈도우 GUI / AsyncSelect 모델

서버 : 콘솔 / select 모델

클라,서버 모두 STL 사용. (직접만든 자료구조 금지 / 버퍼큐,직렬화버퍼 제외)

2. 클라,서버 모두 패킷 생성,사용시 직렬화 버퍼 사용. (헤더를 제외하고 패킷 구조체 없음)

3. 메시지 송수신 부에 스트림버퍼(큐, 링버퍼)를 거쳐서 사용. (클라,서버 공통)

각 클라이언트 마다 SendQ, RecvQ 가짐.

4. 서버는 select 모델 이지만 64명 이상 처리 가능하게끔 구현.

클라이언트를 64명씩 끊어서 select 호출을 반복적으로 해줌.

지금부터 모든 자료구조는 STL의 컨테이너로 대체할것이며, 직렬화버퍼도 사용한다.

또한 Select모델을 사용하지만 한번에 64명이상이도 커버가 가능하게끔 로직을 짜야한다.

(이는 64명씩 끊어서 처리하고, 다음거 처리하고.. 이런얘기.)

**세션을 어떤 자료구조로 하든 동적으로 할당된 포인터로 관리할것이고, 네트워크 프로시져를 호출하면 그안에서는 현재 리스트에 있는 세션목록들에 begin부터 end까지 싸그리 돌면서 64명씩 끊어서 호출 시키는 것이다.**

**이는 한번 루프돌 때 전체에 대한 select검사가 이루어져야 하는 것을 뜻함.**

(리슨소켓이 포함되므로 이것도 염두하고 한다.)

추가로, 예전에 Select모델로 MoveStar예제를 할때는 보내고싶을 때 마구잡이로 보냈으나 이제부터는 보내고싶다면 SendQ에 큐잉하여 Select를 거쳐 send한다.

**이는 결국 모든 유저를 대상으로 한프레임에 한번에 send가 이뤄진다는 것이다.**

**추가로 염두해 둘 것**

이 예제가 완성되면 해당 코드를 그대로 들어다가 다른 프로젝트에 붙여 넣는다.

이후 스트레스 테스트가 될수있게끔 에코서버 같이 만들면서 대규모로 송수신하는것을 테스트 하하 될 것.

그리고 세션이 어떤 행위를 할때마다 무조건 눈에 보이게 모두 출력할 것. 지금 현재 제공된 서버는 send하는 것은 찍고있지 않은데 이것도 찍을 것.

내서버가 어떻게 돌아가는지, 적은 코드가 정상적으로 작동하는지 정도는 반드시 확인할것.

그냥 모든 로그를 찍고 확인하는 것을 권장한다. (완전히 문제 없다고 판단되면 필요없는 것은 그때 지우더라도 )Send를 호출할때도 몇바이트 보냈는지, 받을때도 뭘 받았는지…etc..

**<6.1>**

**채팅서버 / 클라이언트 추가.**

**(프린트물 프로토콜 부는 위 내용과 중복되기에 생략한다.)**

**직렬화 버퍼 사용시 장점**

**1. 유동적인 프로토콜 디자인 가능**

**2. 구조체 바이트 정렬에 걱정이 없음**

**3. 구조체가 선언된 헤더가 없으므로 특정 프로토콜 수정 시 컴파일 시간 단축.**

**직렬화 버퍼 사용시 단점**

* **프로토콜을 문서나 주석으로 상세히 기록해야 한다.**
* **데이터를 입력과 처리하는 부분에서 순서가 정확해야 한다.**
* **중간에 프로토콜이 변경되는 경우 실수가 발생할 가능성이 있다.**
* **불필요한 Copy가 한번 더 일어나게 된다.**

**Select모델, Nonblocked Socket**

**Select모델을 사용하더라도 Non-BlockSocket으로 하는것을 권장한다.**

**Select모델을 사용할때는 Block Socket을 사용하더라도 왠만해서는 상관없겠지만 가능성이 아예 없는것은 아니다.**

**이는 우리가 송신버퍼에 Copy를 넣다가 block걸리는 상황(상대방의 윈도우사이즈가 0이라 보낼수없는경우.)을 뜻하는데, 이경우는 극히드물다.**

**Recv는 Select모델을 사용할때 block걸리는상황은 없다고 봐도 무방.**

**part1.서버부**

**구조체 설계 클라이언트, 채팅방.**

// 클라이언트 구조체 (세션을 겸한 클라이언트.)

// 세션과 유저라는 개념을 분리하지 않고 하나로 간것.

struct CLIENT

{

SOCKET socket;

SOCKADDR\_IN addr;

CRingBuffer SendQ;

CRingBuffer RecvQ;

DWORD UserNo; //유저 고유번호

DWORD RoomNo; //방 번호

WCHAR NickName[NICK\_MAX\_LEN];

};

유저가 로비에 있는지, 방에 들어가있는지에 대한 것을 flag로 관리해도 상관없으나, 여기서는 어떤 방에 들어가 있지 않으면 로비에 있다고 보고, 유저 고유번호가 부여가 된 상태는 로그인 된 상태이므로 굳이 변수를 두지않고 관리해도 상관없다.

**이는 실제 MMORPG에서** (ex 거래제한(거래가 되는 구역에서만 거래가능), PVP(PVP가 가능한 구역에서만 PVP가능) **관련 메시지가 와서 검증을 할때도 통용되는 개념이다.**

**이때 변수가 많아지면 많아질수록 유지보수 과정에서 버그가 생길 가능성이 많아 지기 때문에 연결되는 개념으로 커버하는 경우도 많다.**

//채팅방 구조체

struct CHAR\_ROOM

{

DWORD RoomNo;

WCHAR Title[256];

std::list<DWORD> UserList;

};

따로 채팅방 이름길이에 대해 제한은 생략했는데, 제대로 만든다면 기획단계에서 나와야 할것. (보통 클라이언트에 표현할수있는 한줄의 길이로 가기 때문에 클라이언트랑 상의한다.)

**유저들을 List로 관리한다. 이때 여기서 ‘어떤것’을 다룰지 고민을 해봐야 한다.**

* **위 예제와 같이 유저 고유번호를 다룬다.**
* **유저 클라이언트에 대한 포인터객체 자체를 관리한다.**

**유저번호를 기준으로 관리한다면 결국 (컨텐츠 진행시)번호를가지고 클라이언트를 찾아내야 하는 작업이 더 들어가기 때문에 성능면에서는 2번이 낫다.**

**하지만 안정성 면에서는 1번이 더 뛰어날것. (멀티 스레드환경에서는 다양한 문제가 발생)**

((

추가적으로, 안정성을 높이기 위해서는

**포인터에 대한 데이터를 한곳에서 관리하고, 나머지는 이것을 참조할수있는 키만 관리하는것.**

단 안정성이 올라가는대신 성능면에 손해가 있으므로 초반에만 이렇게하는것을 권장한다.

))

**현재 예제에서는 로비에서 할 수 있는것도, 알수있는 정보도 없기 때문에 로비에 대한 정보(구조체)는 없다.**

**컨텐츠 관리 전역 데이터**

로그인에 성공했을때 세션을 관리하는 자료구조에 insert되어 관리될 것이다.

**Connect하면 Accept하여 무조건 우리쪽에서 관리해야하므로**, (Accept만 진행하고 로그인을 진행하지 않은 연결들이 서버에 상주하면 안될 것) **위 예제같은 경우 Accept하고 바로 고유번호 부여한다.**

**이때 만약 Select 모델이 아니거나, 네트워크 부(L5, L6)와 컨텐츠부(L7)를 분리될때..**

**stateful서버를 만들때는 데이터를 처리하고 로직을 주는것에 추가로** (Stateless는 연결이라는 대상이 없기 때문에 고려대상이 아님)**timeout이 필수적으로 들어가야하고, 이에 대한 기준은 컨텐츠 파트(L7)에서 내려야 한다.**

**네트워크 파트(L4)의 KeepAlive는 TCP의 연결이고, 우리가 필요한것은 내 클라이언트가 맞는가 확인하는 것이기 때문에 적절치 않다.**

**L5, L6에서 timeout을 처리할 경우 해당 네트워크 라이브러리에 범용성이 떨어질 것이다.**

우리는 궁극적으로 컨텐츠에 종속되지않는 네트워크 라이브러리 형태를 원하기 때문.

**결국, 로그인 유무상관없이 connect하여 Accept하는 행위 자체만으로 관리대상에 포함시켜 어떤 소켓이 얼만큼의 송신이 있는지 항시 파악할 수 있어야 한다.**

std::map<DWORD, CLIENT\*> g\_ClientMap; //사용자 관리

std::map<DWORD, CHAR\_ROOM> g\_RoomMap; //채팅방 관리

SOCKET g\_ListenSocket = INVALID\_SOCKET;

DWORD g\_UserKey = 1; //유저 고유번호

DWORD g\_RoomKey = 1; //방 고유번호

위 예제에서는 map을 사용하지만, 정렬해야될 이유가 없으므로 unordered\_map을 권장한다.

**이때 고유키 부분은 실제 게임서버에서 이처럼 사용할 수 없다. 계정 생성시 실제로 영원히 고유한 번호가 부여되어야 한다.**

**(서버 통합을 고려하여 서버마다 구분이 되게끔 해야할것)**

(여기서 아이템은 희귀한 아이템을 말하며, 소모성이나 잡다한 아이템은 type으로 몇개 가지고 있다고 간단하게 관리한다. 아이템에 고유한 번호를 부여하는이유는 추적(복사문제 등)을 위함)

**고유키를 만드는 여러가지 방법**

* **유니크한 값을 만들어내는 공식을 만들어 사용한다. (ex 서버번호 + 날짜 + 넘버링)**
* **일반적으로 DB에서 생성을 시킨다.**
* **카운팅하는 서버를 따로 둔다.**
* **미리 구역을 나눈다. (ex 1~ 10억, 10억1~ 20억..)**

**Setlocale**

#include <locale.h>

char \*setlocale(int category, const char \*locale);

프로그래밍은 일반적으로 영어를 기반으로 하기 때문에, 언어 설정을 해줘야 한다.

이 같은 설정을 하지 않을 경우 가장 기본적으로 파일 입출력시 한글폴더명을 인식하지못한다.

setlocale(LC\_ALL, "");

일반적으로 LC\_ALL(모든인자 적용) 이 선호되며, 위와같이 ""를 넣어준다면 OS를 기반으로한 지역언어 설정을 적용하게 된다.

**MAIN(NetworkInital())**

int main()

{

setlocale(LC\_ALL, ""); //

if (NetworkInital() == FALSE)

{

wprintf(\_T("네트워크 초기화 에러\n"));

return 0;

}

while (true)

{

NetworkProcess();

//특정 키 및 상황에 따른 종료 처리

if (/\*체크\*/false)

{

closesocket(g\_ListenSocket);

break;

}

}

//서버마무리 종료 작업.

return 0;

}

**NetWorkProcess()**

Select로 한번에 64개(FD\_SETSIZE)까지밖에 처리하지못함.

하지만 클라이언트가 몇명이든 Select를 호출하는 부분에서 모두 처리해야 하기때문에, 이와같이 코드를 작성한다.

void NetworkProcess(void)

{

//FD\_SET을 64개씩 끊어서 하기위한 작업

CLIENT\* pClient;

DWORD UserTable\_NO[FD\_SETSIZE]; //FD\_SET에 등록된 UserNo를 저장

SOCKET UserTable\_SOCKET[FD\_SETSIZE]; //FD\_SET에 등록된 소켓을 저장.

int SocketCount = 0;

FD\_SET ReadSet;

FD\_SET WriteSet;

FD\_ZERO(&ReadSet);

FD\_ZERO(&WriteSet);

memset(UserTable\_NO, -1, sizeof(DWORD) \* FD\_SETSIZE);

memset(UserTable\_SOCKET, INVALID\_SOCKET, sizeof(SOCKET) \* FD\_SETSIZE);

//리슨소켓 넣기

FD\_SET(g\_ListenSocket, &ReadSet);

UserTable\_NO[SocketCount] = 0;

UserTable\_SOCKET[SocketCount++] = g\_ListenSocket;

//리슨 소켓 및 접속중인 모든 클라이언트에 대해 SOCKET을 체크한다.

std::map<DWORD, CLIENT\*>::iterator iter;

for (iter = g\_ClientMap.begin(); iter != g\_ClientMap.end();)

{

pClient = iter->second;

**++iter;**

//해당 클라이언트 ReadSet등록

//SendQ에 데이터가 있다면 WriteSet에 등록.

UserTable\_NO[SocketCount] = pClient->UserNo;

UserTable\_SOCKET[SocketCount] = pClient->socket;

FD\_SET(pClient->socket, &ReadSet); //ReadSet은 전체대상.

if (pClient->SendQ.GetUseSize() > 0) //WriteSet은 보낼게 있는 클라만.

FD\_SET(pClient->socket, &WriteSet);

SocketCount++;

//select최대치 도달, 만들어진 테이블 정보로 select호출 후 정리.

if (FD\_SETSIZE <= SocketCount)

{

SelectSocket(UserTable\_NO, UserTable\_SOCKET, &ReadSet, &WriteSet);

FD\_ZERO(&ReadSet);

FD\_ZERO(&WriteSet);

memset(UserTable\_NO, -1, sizeof(DWORD) \* FD\_SETSIZE);

memset(UserTable\_SOCKET, INVALID\_SOCKET, sizeof(SOCKET) \* FD\_SETSIZE);

SocketCount = 0;

}

}

//전체 클라이언트 for문 종료 후 SocketCount수치가 남아있다면,

//추가적으로 마지막 Select호출을 해준다.

if (SocketCount > 0)

{

SelectSocket(UserTable\_NO , UserTable\_SOCKET, &ReadSet, &WriteSet);

}

}

**여기서 나올수 있는 문제**

지금은 연결에대한 요청과 연결해제가 빈번하게 나타나지 않기때문에 큰 문제를 경험하지 못했을 수도있음.

**결국 세션을 끊어내는 Disconnect는(소켓정리, Client자료구조 에서 제거)Select의 결과로서만 알수 있다.**

**Select결과로 Recv, 또는 Send를 했는데 ERROR가 발생하여 Disconnect를 호출하면 해당 세션이 삭제될 것이다. 이때 iterater가 날아가는 문제가 생길 수있음.**

((

별이동 예제에서도 비슷한 문제가 존재했었다.

이는 iter를 가지고 순회하는 도중에 내부에서 또 iter로 순회하는 경우, (이때는 Send를 하고싶을때 아무곳에서나 했음) 코드 일부가 재귀적으로 도는 현상이 있는 경우. (누군가가 연결이 끊겨서 종료 로직으로 들어갔는데, 내부에서 또 Send가 있고 또 그 내부에서 연결이 끊겼고... 무한루프.)

))

지금 예제에서는 어쨌든 Select를 지나쳐야지만 끊겼다라는 판단이 서기 때문에. 컨텐츠 파트와 여기가 일단 분리되어있는 상태이다.

**그래서 이 로직에서 선택한 방법은 iter가 가리키는 값을 백업해놓고 iter를 증가시켰다. 이는SelectSocket함수 내부에서 List 요소를 삭제하는 경우를 방지한것.**

위와같은 문제는 어떤 방식으로 해결하든 해결만 하면된다.

**생각해 볼수있는 방법**

* **FD\_SET자체를 보관하여 연결을 끊을때 FD\_SET에서 뺴고, 내부에서 모두 카피해 놓는다. 이는FD\_SET에 하나씩 카피하는 행위에 대한 부담을 덜자는 의미.**
* **FD\_SET을 하나씩 등록하여 한명당 Select를 하도록 한다. (지금처럼 모아서가 아니라 10명이면 싶으면 Select가 10번씩 나오는것.) 단, 이방법은 성능이 매우 저하됨.**
* **멀티쓰레드로 64개씩 끊어서 가는것. (640명일경우 쓰레드 10개) 이경우 어떤 쓰레드인지 적절하게 배분을 해줘야 할것이다. (예전에 자주 사용되었으나, 지금에 와서는 이렇게 할바에 IOCP를 사용)**

Select모델 자체를 라이브러리화 하지는 않을 것이나, 학원에서 제공되는 더미들은 모두 위와같은 Select모델의 코드를 가지고 갈것이다.

**더미로 테스트를할때, 정상적인 테스트를하려면 하나의 더미 프로세스가 몇천명은 커버해야 한다**.(많으면 5천명, 최소한 천명이상) **더미를 붙일때 몇만명의 유저를 가상으로 만들어야 하기 때문에 더미 수준이 몇백명 이라면 더미를 수백개를 켜야하기때문...**(이경우 감당안됨)

**때문에 더미를 만들때는 한쓰레드당 5~60명정도로 가고 쓰레드를 100개 정도로 가져간다.**

(더미같은 경우 유저끼리의 상호작용은 너무 어렵기때문에 만들지 못함)

때문에 위와같이 쓰레드로 따로 분리하면 간단하게 구현이 될 것.

**SelectSocket**

위에서 FD\_SETSIZE(64개)만큼 호출된다. 유저테이블에 대한번호, 소켓테이블, 그리고 readSet과 Writeset을 인자로 넣어주고있음.

**TIME\_OUT값은 (0,0). 64개 이상을 커버해야 하기때문에, 만약에 값이 들어가서 기다리게 될 경우 뒤에있는 클라는 처리가 늦어지기 때문.**

(별이동 예제에서는 TIME\_OUT값을 0(무한)으로 했다. 유저가 접속도하지않고 메시지도 보내지 않는다면 할일이 없기 때문.)

0이기 때문에 이 예제를 실행시킨다면 CPU를 100%먹게될것. 지금 구조에서는 이를 해결하려면 잠시 쉬게하는 수밖에 없음. 이 서버에 CPU부하를 낮추기위해 timeOut을 거는것은 적절한 선택이 아님.

void SelectSocket(DWORD\* TableNO, SOCKET\* pTableSocket, FD\_SET\* pReadSet, FD\_SET\* pWriteSet)

{

//select함수의 대기시간을 입력한다.

timeval Time;

Time.tv\_sec = 0;

Time.tv\_usec = 0;

//접속자 요청과, 현재 접속중인 클라이언트들의 메시지 송신을 체크한다.

int res = select(0, pReadSet, pWriteSet, 0, &Time);

//리턴값이 0이상일경우 누군가의 데이터가 왔다는 의미

if (0 < res)

{

//Table Socket을 돌면서 어떤 소켓에 반응이 있었는지 확인한다.

for (int i = 0; i < FD\_SETSIZE; ++i)

{

if (pTableSocket[i] == INVALID\_SOCKET)

continue;

//write체크

//readset과 writeset에 대한 순서는 상관없지만, 정상적인 연결종료라면 readSet에서 판단되는게 우선.

if (FD\_ISSET(pTableSocket[i], pWriteSet))

NetProc\_Send(TableNO[i]);

if (FD\_ISSET(pTableSocket[i], pReadSet))

{

//ListenSocket은 접속자 수락용도 이므로 별도처리 / UserNo가 0이면 ListenSocket.

if (TableNO[i] == 0)

NetProc\_Accept();

else

NetProc\_Recv(TableNO[i]);

}

}

}

else if (res == SOCKET\_ERROR)

wprintf(\_T("select socket error\n"));

}

**STL컨테이너를 활용할때 권장하는 방법**

**삽입/검색/삭제(+순회)에 관련하여 랩핑된 함수를 우선적으로 만들어놓고, 이것을 가져다 쓰는 형태로 사용하는것을 권장한다.**

**이는 해당함수만 손보면 컨테이너를 바꾸기 용이하기 때문이다.**

**NetProc\_Recv**

void NetProc\_Recv(DWORD UserNO)

{

CLIENT\* pClient;

int res;

char RecvBuff[512];

//해당 사용자 세션 찾기.

pClient = FindClient(UserNO);

if (NULL == pClient)

return;

//받기 작업

res = recv(pClient->socket, RecvBuff, sizeof(RecvBuff), NULL);

//에러 발생, Recv가 0일경우 종료.

if (SOCKET\_ERROR == res || res == 0)

{

closesocket(pClient->socket);

DisconnectClient(UserNO);

return;

}

//받은 데이터가 있다면

if (res > 0)

{

//일단 RecvStreamQ에 넣는다.

pClient->RecvQ.Enque(RecvBuff, res);

//패킷이 완료되었는지 확인.

//패킷처리중 문제가 발생한다면 종료.

//패킷은 하나이상이 버퍼에 있을 수 있으므로 반복문으로 한번에 처리.

while (true)

{

res = CompleteRecvPacket(pClient);

//더이상 처리할 패킷없음.

if (res == 1)

break;

if (res == -1)

{

wprintf(\_T("PacketError UserNO %d \n", UserNO));

return;

}

}

}

}

이때 문제가 생겨서 클라이언트가 종료되어야 할시, 클라이언트 관리맵에서 빼야할것이고 컨텐츠 적인 부분으로 본다면 방에있었다면 방에서퇴장(다른클라에게 Send), 방에 혼자있었다면 방삭제.

추가적으로,

로그를 찍을때는 필요한 부분을 로깅해야한다. 패킷처리시 오류가 났다면 상황을 아는것에 더해왜 오류가 났는지 상세한 데이터를 다 찍어내야한다. (어떤 유저가 어떤상태에서 뭐가왔는지, 어떤 데이터가 잘못됐는지..)

정안되면 링버퍼/직렬화버퍼 까지 다 뽑아야할 것.

**NetProc\_Send**

void NetProc\_Send(DWORD UserNO)

{

CLIENT\* pClient;

int res;

int SendSize;

char SendBuff[512];

//해당 사용자 세션 찾기

pClient = FindClient(UserNO);

if (NULL == pClient)

return;

//SendQ에 있는 데이터들을 보낸다.

SendSize = pClient->SendQ.GetUseSize();

SendSize = min(512, SendSize);

//큐에 보낼 데이터가 있을 경우에만 보내도록 한다.

if (SendSize <= 0)

return;

//일단 Peek함수를 사용하여 빼낸후, 전송이 제대로 마무리 되었을때 해당 내용을 지운다.

pClient->SendQ.peek(SendBuff, SendSize);

//전송

res = send(pClient->socket, SendBuff, SendSize, NULL);

if (res == SOCKET\_ERROR)

{

DWORD err = WSAGetLastError();

if (err == WSAEWOULDBLOCK)

{

wprintf(\_T("Socket WOULDBLOCK - UserNO:%d, \n"), UserNO);

return;

}

wprintf(\_T("Socket Error - UserNO: %d ErrorCode : %d \n"), UserNO, err);

closesocket(pClient->socket);

DisconnectClient(UserNO);

return;

}

else

{

//송신작업을 완료했다.

//패킷이 완전히 전송되었다는것이 아니라 소켓버퍼에 복사를 완료했다는 의미.

pClient->SendQ.MoveFront(res);

}

}

현재 서버부의 Recv, Send는 별도의 RecvQ, SendQ가 있으며, Recv, Send시에 임시 버퍼를 사용하고있다.

Recv > 임시버퍼 > RecvQ버퍼.

SendQ버퍼 > 임시버퍼 > Send.

중간에 임시 버퍼를 사용하므로 코드나 로직적으로는 간결하지만 불필요한 메모리 복사가 존재한다.

recv > RecvQ버퍼

SendQ버퍼 > Send

위의 방식으로 개선되어야 할 것.

**CompleteRecvPacket**

int CompleteRecvPacket(CLIENT\* pClient)

{

PACKET\_HEADER header;

int RecvQSize = pClient->RecvQ.GetUseSize();

//받은 내용을 검사해야한다. 이때 패킷헤더 크기 이상으로 뭔가 받은 경우가 아니라면

//검사해볼 필요가 없음.

if (sizeof(PACKET\_HEADER) > RecvQSize)

return 1; //더이상 처리할 것이 없으므로 중단.

//1. PacketCode검사.

//Peek으로 검사하는 이유는, 헤더를 얻어 사이즈를 비교하고 하나의 완성된 패킷 만큼의 데이터가

//있는지 확인하기 위해서이다. 패킷을 마저 얻을지, 그냥 중단할지 결정해야함.

//Get으로 얻는 경우 검사 후 사이즈가 맞지 않으면 헤더를 다시 큐의 자리에 //돌려놓을수가없음.(FIFO구조이므로)

pClient->RecvQ.peek((char\*)&header, sizeof(PACKET\_HEADER));

if (PACKET\_CODE != header.Code)

return -1;

//2.큐에 저장된 데이터가 얻고자 하는 패킷의 크기만큼 있는지 확인.

if (header.PayLoadSize + sizeof(PACKET\_HEADER) > (DWORD)RecvQSize)

return -1;

//위에서 헤더부분은 Get이 아닌 PeeK으로 빼왔으므로 Q안에서 지운다.

pClient->RecvQ.MoveFront(sizeof(PACKET\_HEADER));

CMsg msg; //불필요해 보인다. 나중에는 메모리풀을 사용하여 재사용하는 형식으로 갈 것.

//PayLoad부분 패킷버퍼로 빼옴.

if (header.PayLoadSize != pClient->RecvQ.Deque(msg.GetBufferPtr(), header.PayLoadSize))

return -1;

//패킷 클래스에 버퍼 포인터를 얻어 임의로 데이터를 넣었으므로 클래스 내부의 이동처리를 해준다.

msg.MoveWritePos(header.PayLoadSize);

//체크섬 확인

BYTE CheckSum = MakeCheckSum(&msg, header.MsgType);

if (CheckSum != header.MsgType)

{

wprintf(\_T("CheckSum Error[UserNO:%d]\n", pClient->userNO));

return -1;

}

//실질적인 패킷 처리 함수를 호출한다.

if (PacketProc(pClient, header.MsgType, &msg))

return -1;

return 0; //패킷 1개 처리완료. 본 함수 호출부에서 Loop를 유도한다.

}

**PacketProc**

bool PacketProc(CLIENT\* pClient, WORD MsgType, CMsg\* msg)

{

wprintf(\_T("PacketProc [UserNO:%d]\n", pClient->userNO, MsgType));

switch (MsgType)

{

case REQ\_LOGIN:

return NetPacket\_ReqLogin(pClient, msg);

break;

case REQ\_ROOM\_LIST:

return NetPacket\_ReqRoomList(pClient, msg);

break;

case REQ\_ROOM\_CREATE:

return NetPacket\_ReqRoomCreate(pClient, msg);

break;

case REQ\_ROOM\_ENTER:

return NetPacket\_ReqRoomEnter(pClient, msg);

break;

case REQ\_CHAT:

return NetPacket\_Req\_CHAT(pClient, msg);

break;

case REQ\_ROOM\_LEAVE:

return NetPacket\_Req\_ROOM\_LEAVE(pClient, msg);

break;

}

return 1;

}

메시지 (패킷) 실제 처리부는 위와같이 종류별로 함수를 나누며, 인자는 하나로 통일시키는것 을 권장한다.

차후에 함수포인터 형식으로 변경을 시킬수 있게되고, 메시지별 클래스 방식으로 변경되는 경우에도 가상함수로 변경하기 쉽다.

**또한 각 처리 함수로 전달되는 패킷의 데이터(위 코드에서는 msg)은 절대로 전역변수, 클래스 멤버변수같이 공용부분을 사용하여 전달하지 말 것.**

**함수에서 다른 함수로 데이터를 전달하고자 할때 멤버변수, 전역변수등의 공용 공간을 사용하면 차후 스레드 구조에서 데이터 공유의 문제가 발생한다.**

함수와 함수가 중복적으로 호출되는 경우에도 문제가 발생할 요소가 많음.

**요청작업1 A -> Func**

**요청작업2 B -> Func**

**요청작업3 A -> Func**

**요청작업4 A -> Func**

위와같은 로직에서 A, B에서 Func로 데이터를 전달 할 때 **외부의 공용공간을 사용하여 전달되면 요청작업 1,2,3,4는 절대 병렬로 동시에 돌아갈 수 없게 되며, 공용메모리가 언제 변경,삭제 될지 불안한 상태로 진행된다.**

**NetPacket\_ReqRoomCreate**

bool NetPacket\_ReqRoomCreate(CLIENT\* pClient, CMsg\* msg)

{

WCHAR RoomTitle[256] = { 0, };

WORD TitleSize;

//256 이상의 글자인 경우 예외처리를 해야한다.

\*msg >> TitleSize;

msg->GetData((char\*)RoomTitle, TitleSize);

//방 생성 작업

CHAT\_ROOM\* pRoom = new CHAT\_ROOM;

memset(pRoom->Title, 0, sizeof(WCHAR) \* 256);

pRoom->RoomNo = g\_RoomKey;

wcscpy\_s(pRoom->Title, RoomTitle);

g\_RoomMap.insert(std::pair<DWORD, CHAT\_ROOM\*>(pRoom->RoomNo, pRoom));

//방생성 로그

wprintf(\_T("방생성 [UserNO:%d][Room:%s] [TotalRoom:%d]\n", pClient->UserNO, pRoom->Title, g\_RoomMap.size()));

//방생성 결과를 클라이언트로 전송

Send\_ResRoomCreate(pClient, RESULT\_ROOM\_CREATE\_OK, pRoom);

return TRUE;

}

**Send\_ResRoomCreate**

void Send\_ResRoomCreate(CLIENT\* pClient, BYTE res, CHAT\_ROOM\* pRoom)

{

PACKET\_HEADER header;

CMsg msg;

MakePacket\_ResRoomCreate(&header, &msg, res, pRoom);

if (res == RESULT\_ROOM\_CREATE\_OK)

SendBroadcast(&header, &msg);

else

SendUnicast(pClient, &header, &msg);

}

패킷(메시지) 처리부는 한가지만 예시를 제공한다.

**MakePacket\_ResRoomCreate**

void MakePacket\_ResRoomCreate(PACKET\_HEADER\* header, CMsg\* msg, BYTE res, CHAT\_ROOM\* pRoom)

{

WORD TitleSize;

msg->Clear();

msg->SetData((char\*)&res, sizeof(BYTE));

if (res == RESULT\_ROOM\_CREATE\_OK)

{

\*msg << pRoom->RoomNo;

TitleSize = wcslen(pRoom->Title) \* sizeof(WCHAR);

\*msg << TitleSize;

msg->SetData((char\*)pRoom->Title, TitleSize);

}

header->Code = PACKET\_CODE;

header->CheckSum = MakeCheckSum(msg, RES\_ROOM\_CREATE);

header->MsgType = RES\_ROOM\_CREATE;

header->PayLoadSize = msg->GetDataSize();

}

직렬화 버퍼를 사용하여 메시지를 생성하는 경우 위와같이 사용.

패킷 메시지를 만드는 방법중 가장 간단하게 생각해볼수 있는 방법은,

char msg[1000];

\*(DWORD\*)(msg + 0) = 10;

\*(DWORD\*)(msg + 4) = 20;

\*(short\*)(msg + 8) = 20;

위 방식은 코드 작성도 어렵고 대형프로젝트의 유지보수가 불가능하다.

실무에서 이런식의 개발은 권고퇴사(...)

구조체를 만들어서 사용하거나, 별도의 큐 버퍼를 만들어서 최대한 간결한 사용법으로 가능하도록 만들어야 한다.

**SendUnicast**

void SendUnicast(CLIENT \*pClient, PACKET\_HEADER\* header, CMsg\* msg)

{

if (pClient == NULL)

{

wprintf(\_T("SendUnicast Client is NULL\n"));

return;

}

pClient->SendQ.Enque((char\*)header, sizeof(PACKET\_HEADER));

pClient->SendQ.Enque((char\*)msg, msg->GetDataSize());

}

위부분은 헤더와 Payload파트를 두번에 걸쳐서 Enque하고있다.

지금은 PayLoad가 만들어져야지만 헤더를 만들수 있기 때문에 지금 구조로서는 어쩔수가 없다.

헤더를 먼저 만들 방법이없다.(길이와 체크섬때문)

나중에는 직렬화버퍼안에다 헤더를 꽂을 것이다. 직렬화버퍼 설계 자체를 헤더공간을 비워둘것.

직렬화버퍼가 하나가 독립적인 헤더까지 포함시킨 메시지가 만들어지게 할것이다.

**SendBroadcast**

void SendBroadcast(PACKET\_HEADER\* header, CMsg\* msg)

{

CLIENT\* pClient;

std::map<DWORD, CLIENT\*>::iterator iter;

for (iter = g\_ClientMap.begin(); iter != g\_ClientMap.end(); ++iter)

{

pClient = iter->second;

**SendUnicast(pClient, header, msg);**

}

//만들어진 패킷을방 유저에게 보내기.

}

**SendBroadcast\_Room**

void SendBroadcast\_Room(CHAT\_ROOM\* pRoom, CLIENT\* pClient, PACKET\_HEADER\* header, CMsg\* msg)

{

CLIENT\* pDestClient;

std::list<DWORD>::iterator iter;

for (iter = pRoom->UserList.begin(); iter != pRoom->UserList.end(); ++iter)

{

pDestClient = FindClient(\*iter);

if (pClient != pDestClient)

SendUnicast(pDestClient, header, msg);

}

}

**MakeCheckSum**

BYTE MakeCheckSum(CMsg\* msg, WORD Type)

{

int size = msg->GetDataSize();

BYTE\* ptr = (BYTE\*)msg->GetBufferPtr();

int CheckSum = Type;

for (int i = 0; i < size; ++i)

{

CheckSum += \*ptr;

ptr++;

}

return (BYTE)(CheckSum % 256);

}

**결국 메시지를 보내는함수는 총 3개가 나오게된다.**

* **유니캐스트**
* **브로드캐스트**
* **방에 있는 사람에게만 뿌림.**

단, 위 항목처럼 BroadCast를 할때 자료구조를 돌면서 UniCast함수를 호출하는것은 함수가 간결해지지만 Call횟수가 늘어남.

단, 상용화된 MMORPG에서는 한번에 모든유저에게 보내는것이 아닌 끊어서 보낼 것.(몇명단위로.)

가장 좋은방법은 병렬로 보내는것이 좋지만.. 난이도가 상당하기 때문.

중간에 추가된 사람에게도 보낼것인지..등.

**프린트물(Client part) 생략**

DialogBox로 DialogBox를 만들면 블락. 이걸 모달 DialogBox라고 부른다.

이 DialogBox가 종료되기 전까지는 return하지않는다.

근데 모달리스 같은경우는 해당 Dialog를 띄워놓고 따로 동작을 한다.

해당 Dialog는 자식윈도우 개념이다. 자식윈도우를 띄워놓고 바로 리턴한다.

그럼 메시지 루프에 같이 처리하는게 모달리스.

포토샵같은거나 이런곳에서 자식윈도우를 띄우는걸, 자식윈도우가 아닌 다이얼로그로 많이 쓴다.

그래서 지금 이 프로그램자체가 윈도우가 없고 다이얼로그 기반의 어플리케이션의 내용이다.

때문에 해당내용에는 CreateWindow가 없다.

CreateDialog하면 핸들이 반환될것이고, 이핸들을 AsycSelect에 등록하면 똑같이 수신되는것.

지금 해당클라의 WinProc는 우리가 보던 로직과는 조금다르다.

우리는 지금까지

while (true)

{

if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))

{

if (msg.message == WM\_QUIT)

break;

TranslateMessage(&msg);

DispatchMessage(&msg);

}

else

{

if (g\_Connect == TRUE)

GamePlay();

}

}

이러한 형태로 사용했다.

모달리스 다이얼로그는 다이얼로그가 여러개 떠있는 상황일 수있다....(생략)

**<6.1>**

**1:44:00**

**<6.4>**

지금껏 만들었던 서버는 클라이언트를 받아 컨텐츠가들어가는 서버를 만들었다.

여기서 말하는 서버는 **서버끼리의 분할역할**을 하는 서버를 뜻한다. (ex친구관리 서버)

**친구관리 서버를 만든다고 가정하면 친구관리만을 전담하는 서버로, 친구관리 서버의 입장의 클라이언트는 서버가 될것이다.**(서버 내부 구조에서 친구관리에 대한 파트를 별도 서버로 분리)

**그리고 우리서버가 친구서버에 조회하여 데이터를 얻어간다.**

**지금 방식의 JSON은 학습목적으로 억지스럽게 사용해보는것이며, 아무도 이런형태로 사용하지 않음**

**(원래는 친구관리데이터의 경우 DB에 저장이 되어야할것.)**

우리의 목표는 메모리의 데이터를 DB가 아닌 JSON에 저장했다가 서버가 켜질때 JSON에서 읽어오는것이다.

**텍스트와 바이너리**

텍스트파일은 문자를 기반으로 하는 코드값이 저장된 파일이며, 바이너리 파일은 텍스트파일을 제외한 모든 파일을 의미한다.

**바이너리**

**바이너리 파일은 데이터를 읽거나 쓸때 파일 구성 형식에 특별한 조건이 없기 때문에 대부분 데이터의 크기로 판단한다. 따라서 ‘n바이트 만큼 읽어라’ 같은 명령을 사용한다.**

**따라서 바이너리 파일은 이런 특징 때문에 데이터만 가지고는 내용이 무엇을 의미하는지 확인하기 어렵고, 해당 파일을 볼수있는 프로그램이 별도로 있어야 한다.**

**(PNG형식의 그림파일을 보려면 그림판과 같은 호환프로그램이 설치되어 있어야 가능하다. 호환되지않는 메모장으로 사용하면 알수없는 문자로 표시될것)**

**텍스트**

텍스트 파일은 문자를 기반으로 하는 코드값이 저장된 파일이다. 문자 기반이기 때문에 전용 프로그램이 아니더라도 **문자열만 확인할 수 있는 프로그램이나 명령을 사용하면 파일의 내용을 확인할 수 있다.** (ex 콘솔창에서 type명령어 사용)

**퍼블리셔와 통신**

만약 퍼블리셔에 계약이 될 경우 퍼블리셔의 플랫폼위에 우리의 게임이 들어가는 방식이 된다.

이경우 플랫폼 쪽에서 계정로그인을 해서 우리쪽 게임을 플레이 하게 될것이다.

**(즉 퍼블리셔가 계정관리를하고 우리는 게임만을 관리한다.)**

**이때 해당 퍼블리셔의 인증을 우리쪽에서 알아야하므로, 우리서버와 퍼블리셔 플랫폼과 통신이 이뤄져야 한다. (로그인유무, 결제)**

**결제 같은 경우 구글플레이에서 서비스하고, 유저가 구글플레이에서 결재했다면 서버쪽으로 결제코드를 가져오고, 우리는 이 결제코드를 구글플레이에 조회한다.**

**예전같은 경우는 해당 작업을 우리가 지금까지 했던(포트,IP, 프로토콜 지정후 바이너리 송신)방식과 같았으나 지금은 완전히 사라진 방법이다. (보안상)**

지금은 모든 통신이 HTTP의 텍스트라는 환경으로 통일되고 있으며(호환성이 굉장히 좋음) ,게임만이 그나마 바이너리 프로토콜을 사용하고 있는 중이다.

**HTTP구조**

hyper text구조에서는 바이너리가 들어가지않고 모든게 다 text이므로 호환성이 좋고 개발도 쉽다. 애초에 HTTP라는 hyper text구조에서는 바이너리가 들어가지않는다. 이미 정해진 프로토콜이고, 서버도 웹서버로 만들면 되므로... 호환성 증가.

**(Hypertext : 참조를 통해 독자가 한 문서에서 다른 문서로 즉시 접근할수 있는 텍스트)**

**지금은 모든 플랫폼의 API는 (구글, 페이스북, 트위터, 넥슨 등) 플랫폼에서 제공하는 기능과 연동하기 위해 UI를주고, 이 데이터가 모두 JSON이다.**

**결국 바이너리 구조는 애초에 없고, XML은 구시대적, JSON거의 대부분.**

**JSON(JAVA Script Object Notation)**

**JSON은 자바스크립트에서 객체를 표현하는 텍스트 데이터 문법 (경량화된 데이터 표현방법)이다.**

자바스크립트에서 쓰는 문법을 하나의 포맷으로 빼버린것.

또는 우리가 만들었던 텍스트파서를 대체하는것도 가능하다.

추가로, PHP에는 이미 JSON이 내장되어있다.

**구글,페이스북,트위터,네이버 등등의 공개 OpenAPI 에서도 모두 JSON 을 기본으로 하며 여러 게임 네트워크 프로토콜 등에서 JSON 이 많이 사용된다.**

(예전에는 XML을 많이 사용했으나, XML은 데이터 하나를 표현하기 위해서 너무많은 태그가 들어가는 등의 문제로(덩치가 크다.) 모든 부분에서 JSON이 활용이 되고있다.)

**JSON 문법**

중괄호{}로서 어떤 데이터를 표현할것이고, :로 Key-Value를 표현할 것이다.

배열은[]로서 표현.

**(1)**

**{ “MemberNo”: “123456”, “SessionTicket”: “R2X5JVYhq4Q==”, “RuneID” : 1001, “CardList”:[1,2,3,4,5,6], “s\_Ver”: 1, “s\_Time”: 123123123 }**

(2)

MemberNo = 123456

SessionTicket = R2X5JVYhq4Q==

RuneID = 1001

CardList = Array[] 1,2,3,4,5,6

s\_Ver = 1

s\_Time = 123123123

**키밸류 형태이므로 쓰기도편하고, 데이터 순서도 의미가 없으므로 중간에 새로운 데이터를 넣어도 아무런 상관이 없다.**

**Stateless 기반의 게임을 만든다고해도 모두 100%기반의 JSON이** 들어간다.

**JSON의 장/단점**

**장점**

표현하기도 편하고, 간단하고, 호환성이 좋다.

* **순서가 바뀔때 바이너리코드가 엉망진창이 되는반면, JSON은 Key-Value type이므로 아무런 상관이 없다.**
* **또 프로토콜을 추가할때 어디에 넣든 상관이 없기 때문에 하위 호환성도 좋아진다.**
* **모든 언어로 라이브러리가 만들어져 있다.**

**단점**

**바이너리 데이터와 비교한다면 JSON은(text기반이므로) 용량도 더 크고 성능도 떨어질것이다.**

**결론적으로 성능과 용량부분의 단점만 제외한다면 매우 큰 이점을 가지고있기 때문에,**

**우리가 CPP기반의 stateful서버를 만들었다고 해도 JSON은 계속해서 활용이 될것.**

**클라이언트 패치**

**게임 로직 내 바이너리코드는 설정파일로 변경할 수 있는 성질이 아니다.**

**여기서 언급하는 것은 설정파일로 따로 빼는 것이 가능한 스크립터 데이터를 말한다.**

PC의 경우 클라패치 전체를 제어할수 있고, 패치를 자주한다고하여 사용자가 불만을 가지는 경우는 적다.

하지만 모바일의 경우는 PC와 같은 패치를 진행하려면 앱스토어에서 추가적인 패치를 진행해야한다. 이런패치를 싫어하는 유저들이 존재한다.

**따라서 앱스토어 패치를 피하기위해, 설정파일로 게임내 로직을 변경시키는 방법을 택한다.**

**이 경우 옛날 클라는 새로운 값을 무시하고 사용하고, 새 클라는 새로운값을 적용시켜 사용.**

(메인 페이지에 UI기능이 새로이 추가되었다는 등..)

이 경우 호환성도 뛰어나고 유저이탈도 막을 수 있음.

**JSON예시**

# 아래는 모바일 게임 윈드러너의 첫 로그인시 서버로부터 받는 JSON 데이터를 캡쳐한 내용이다.

(카카오톡 친구들의 윈드러너 플레이 정보를 받고 있는 것.)

# {"agreed":0,"bestScore":7029,"bestDist":5561,"bestOpts":"C:CHAR1,R:,P:,P2:,CL:0,RL:-1,RE:-1,ITMS:","weekScore":7029,"weekDist":5561,"weekOpts":"C:CHAR1,R:,P:,P2:,CL:0,RL:-1,RE:-1,ITMS:","bestDistRecord":5561,"weekDistRecord":5561,"friendDatas":{"88354349483960817":{"bscr":6272,"bdst":4571,"bopts":"C:CHAR1,R:,P:,CL:0,RL:-1","twscr":0,"twdst":0,"twopts":"","lwscr":0,"lwdst":0, "lwopts":"","nmls":0,"msgblk":"N", "wrClass":"","rank":0},"88334806680033728":{"bscr":868354,"bdst":101962,"bopts":"C:CHAR5,R:RIDE3,P:PET\_L\_004,P2:PET\_S\_010,CL:25,RL:25,RE:1,ITMS:ITM1;ITM2;ITM3","twscr":868354,"twdst":101962,"twopts":"C:CHAR5,R:RIDE3,P:PET\_L\_004,P2:PET\_S\_010,CL:25,RL:25,RE:1,ITMS:ITM1;ITM2;ITM3","lwscr":559906,"lwdst":80546,"lwopts":"C:CHAR5,R:RIDE3,P:PET\_L\_004,P2:PET\_S\_010,CL:25,RL:25,RE:1,ITMS:ITM1;ITM2;ITM3;ITM4","nmls":0,"msgblk":"Y","wrClass":"A","rank":10},

**rapidjson**

소스를 옮기고 lib붙이는 작업없이 헤더만 include해서 사용하면 되는 장점때문에 학원에서는 rapidjson을 사용한다. (제공하는 어떤것을 사용해도 상관없음)

<http://rapidjson.org/>

<https://github.com/Tencent/rapidjson/>

rapidjson-master/include/rapidjson 만 프로젝트에 복사해서 넣으면 될것.

**(stateless로 비동기 온라인게임을 만들어 웹서버로 진행을 한다면 프로토콜은 100% http기반의 JSON이 될것이다. )** (http기반이기 때문에 바이너리는 쓸수가없는상황)

**웹서버 - stateless**

**웹서버가 stateless인것은 맞지만, stateless라고 반드시 http를 써야하는것은 아니다.**

**C++로도 stateless서버를 만들수 있다. 그리고 바이너리로도 진행이 가능하다.**

**하지만 비효율적이므로 웹서버로 개발하고, 이는 text기반이 될수밖에 없다. 때문에 JSON을 사용하게 되는것.**

그러다보니까 JSON을 쓰는것이다.

**JSON은 느리다?**

처음접했을때 걱정을 많이 하는 부분이다.

**http라는 웹프로토콜도 tcp기반이고, 일반적으로 쓰는 웹사이트는 기본이 2~30KB 정도로**

**큰데 이것을 서버에서 계속 전달한다. 이것을 텍스트로 검사해서 파싱하여 사용하므로, 용량이 너무 커서 느리지 않을까 하는 점이다.**

(css코드, htmel, 자바스크립트... 들어가야 할 것이 매우많음)

하지만 C#이란 언어도 생각보다 빠르고 모바일기기가 생각보다 빠르다.

**게임하는데 지장없는 수준으로 성능은 나오기 때문에 느려서 쓰지못한다는 것은 통용되지않음.**

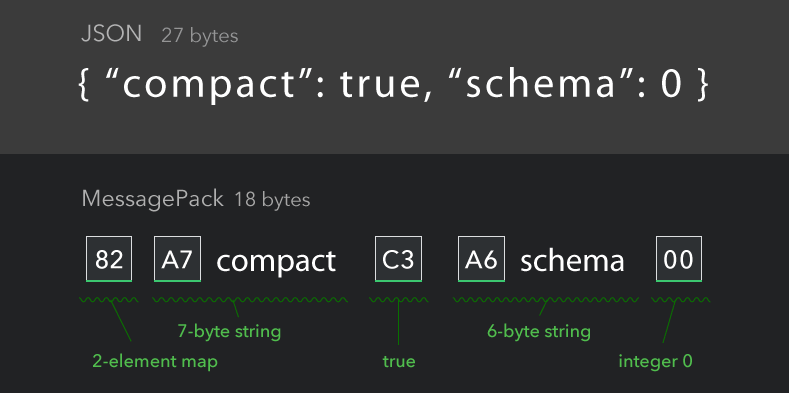
**MessagePack**

JSON의 단점을 보완하기 위해 만들어진 MessagePack.

**바이너리 패킷을 사용하려는 상황에서 key-Value형태를 사용하기에는 단점이 너무 크다.**

**(예전에 바이너리로 byte를 나눠 프로토콜 설계해서 쓸때는, 관리도 힘들고 Key-Value형태가 아니라서 쓰기도 애매했다.)**

이런 단점들을 보완하기위해 JSON이 나온것인데, JSON에도 단점이 존재하기에 이를 또 보완하기위해 MessagePack이라는 중간단계가 나오게된다.



**Message Pack은 JSON과 바이너리의 중간단계로서,**

**바이너리로 표현해서 좋을것들은 바이너리로가고, string으로 표현해야 할 키값같은 것은 string으로 간다.**

(MessagePack은 아직까지 대중화되지는 않았으며, 앞으로 얼마나 대중화될지는 미지수이다.)

**RapidJson**

UTF-16에 대해 내부에 있다..라는점을 찾아볼것. (없음)

우리는 UTF-16인코딩 방식을 사용하고 있는데, **RapidJson을 통해서 JSON코드로 인코딩을 한**(JSON 기준에서 인코딩을 뜻함. 우리의 오브젝트를 JSON으로 만드는것이므로, JSON으로 인코딩 한다는 표현을 사용한다.) **JSON string은 UTF-8이된다.**

**결론적으로 우리는 UTF-16을 넣었는데 UTF-8을 반환한 모양이 된다.**

**따라서 우리가 UTF-16으로 변경을 하는 과정이 들어가야 한다.**

(RapidJson 내에서 이와같이 변경하는 과정이 있는지 찾아볼 것)

**(UTF16 -> UTF8 저장 / UTF8 -> UTF16로드)**

**RapidJson C++에서 사용법**

CPP객체(어떤 변수들..)를 JSON형태로 만든다.

#include "pch.h"

#include <iostream>

#include "Rapidjson/document.h"

#include "rapidjson/writer.h"

#include "rapidjson/stringbuffer.h"

using namespace rapidjson;

StringBuffer StringJSON;

//버퍼 객체생성. (실질적으로 JSON이 만들어지는 공간.)

Writer<StringBuffer, UTF16<>> writer(StringJSON);

//우리가 어떤 데이터를 Writer에 넣으면 StringBuffer에다가 JSON텍스트로 변환시키면서 쌓아 나갈것

//UTF16으로 지정하지않으면 Default인 UTF-8로 진행이 될것.

writer.StartObject();

//계정

writer.String(L"Account");

/\*

Key값으로 String을 넣은것.

Value를 넣어야 하는 위치였으면 Value가 된다.

만약 순서를 맞추지않아 짝을 이루지 못한다면 내용이 꼬일것.

\*/

writer.StartArray();

{ //반복문 돌면서 Account정보 저장

writer.StartObject();

writer.String(L"AccountNo"); //Key

writer.Uint64(1111); //Value

writer.String(L"NickName"); //Key

writer.String(L"닉네임"); //Value

writer.EndObject();

}

writer.EndArray();

//친구 목록

writer.String(L"Friend");

writer.StartArray();

{ //반복문 돌면서 친구 정보 저장

writer.StartObject();

writer.String(L"FriendNo");

writer.Uint64(1);

writer.String(L"FromAccountNo");

writer.Uint64(1);

writer.String(L"ToAccountNo");

writer.Uint64(1);

writer.EndObject();

}

writer.EndArray();

//친구요청

writer.String(L"FriendRequest");

writer.StartArray();

{ //반복문 돌면서 저장

writer.StartObject();

writer.String(L"RequestNo");

writer.Uint64(1);

writer.String(L"FromAccountNo");

writer.Uint64(1);

writer.String(L"ToAccountNo");

writer.Uint64(1);

writer.EndObject();

}

writer.EndArray();

writer.EndObject();

//이 문자열을 저장하거나 패킷을 쏜다.

const char\* pJson = StringJSON.GetString();

//String객체의 GetString을 호출하면 포인터를 반환한다.

//이 포인터에 JSONString이 들어가있는것.

//그럼 우리는 파일에 저장을하든, 네트워크 패킷으로 쏘던 써먹을수 있게 되는것.

요즘나오는 스크립트 언어(C#, PHP, JAVA)는 클래스를 인코딩함수에 넣으면 내부에서 멤버 변수를 모두 알고 있기 때문에 런타임시 누가누군지 알수있고, 알아서 자동으로 세팅해준다.

**const char\* p Json = json.encode(Player);**

**내부에서 모든 멤버를 알아서 다 뽑아내기 때문에 단 한줄로 세팅이 끝난다.**

**하지만 우리는 컴파일 언어기 때문에 다른 방법이없다. 변수이름 자체가 존재하지않는다.**

그래서 우리가 디코딩할때는 이런방법을 택하는것.

**목표 JSON 스트링**

**{ "Account":[ {"AccountNO":1111,"Nickname":"테스트계정1"},**

**{"AccountNO":2,"Nickname":"테스트계정2"},**

**“Friend”:[]**

**“FriendRequest”:**

**[**

**{“RequestNo”:1,”FromAccountNo”:1,”ToAccountNo”:2,”Time”:1437344611},**

**{“RequestNo”:2,”FromAccountNo”:1,”ToAccountNo”:3,”Time”:1437344611}**

**]**

**}**

**JSON스트링을 C++데이터로 변환하기**

이 JSON에 있는 Text에 있는 데이터를 서버가 켜질 때 로딩해서 C++에 있는 데이터로 얻어낸다.

일단 UTF-8로 저장이 되어있었기때문에 UTF-16으로 바꿔준다. (아래함수사용)

#include <iostream>

#include <windows.h>

#include "Rapidjson/document.h"

#include "rapidjson/writer.h"

#include "rapidjson/stringbuffer.h"

#define NICK\_MAX\_LEN 256

using namespace rapidjson;

StringBuffer StringJSON;

//버퍼 객체생성. (실질적으로 JSON이 만들어지는 공간.)

Writer<StringBuffer, UTF16<>>writer(StringJSON);

//우리가 어떤 데이터를 Writer에 넣으면 StringBuffer에다가 JSON텍스트로 변환시키면서 쌓아 나갈것

//UTF16으로 지정하지않으면 Default인 UTF-8로 진행이 될것.

const char\* pJson;

int Size;

bool UTF8toUTF16(const char\* Text, WCHAR\* Buff, int BuffLen)

{

int Re = MultiByteToWideChar(CP\_UTF8, 0, Text, strlen(Text), Buff, BuffLen);

if (Re < BuffLen)

Buff[Re] = (WCHAR)L"\0";

return true;

}

void LoadData(void)

{

//char\* pJson;

/\*파일을 열거나 패킷으로 받은 JSON문자열.

파일의 경우 UTF-8 BOM없음으로 저장되어야 읽어짐.\*/

Document Doc;

Doc.Parse(pJson); // JSON문자열(pJson)을 전달하면 기본적인 파싱을 해줄것.

UINT64 AccountNo;

WCHAR NickName[NICK\_MAX\_LEN];

rapidjson::Value::ConstMemberIterator iter = Doc.FindMember("Account");

//위와 같은 예외 처리를 하는이유 (ㄱ)에 별도설명

if (iter == Doc.MemberEnd())

{

DWORD err = 0;

//return;

}

Value& AccountArray = Doc["Account"]; //Account키를 찾는다.

wprintf(L"\n\nAccountNo / NickName List\n");

for (SizeType i = 0; i < AccountArray.Size(); ++i)

{

Value& AccountObject = AccountArray[i];

AccountNo = AccountObject["AccountNo"].GetUint64();

UTF8toUTF16(AccountObject["NickName"].GetString(), NickName, NICK\_MAX\_LEN);

//읽혀진 데이터 사용

wprintf(L"AccounNo : %lld \t NickName : %s\n", AccountNo, NickName);

}

UINT64 FriendNo;

UINT64 FromAccountNo;

UINT64 ToAccountNo;

Value& FriendArray = Doc["Friend"]; //Account키를 찾는다.

wprintf(L"\n\nFriendList\n");

for (SizeType i = 0; i < FriendArray.Size(); ++i)

{

Value& FriendObject = FriendArray[i];

FriendNo = FriendObject["FriendNo"].GetUint64();

FromAccountNo = FriendObject["FromAccountNo"].GetUint64();

ToAccountNo = FriendObject["ToAccountNo"].GetUint64();

//읽혀진 데이터 사용

wprintf(L"FriendNo : %lld \t FromAccountNo : %lld \t ToAccountNo : %lld\n",

FriendNo, FromAccountNo, ToAccountNo);

}

wprintf(L"\n\nFriendRequest\n");

Value& FriendRequestArray = Doc["FriendRequest"]; //Account키를 찾는다.

for (SizeType i = 0; i < FriendRequestArray.Size(); ++i)

{

Value& FriendObject = FriendRequestArray[i];

FriendNo = FriendObject["RequestNo"].GetUint64();

FromAccountNo = FriendObject["FromAccountNo"].GetUint64();

ToAccountNo = FriendObject["ToAccountNo"].GetUint64();

//읽혀진 데이터 사용

wprintf(L"FriendNo : %lld \t FromAccountNo : %lld \t ToAccountNo : %lld\n",

FriendNo, FromAccountNo, ToAccountNo);

}

}

int main(void)

{

writer.StartObject();

//계정

writer.String(L"Account");

/\*

Key값으로 String을 넣은것.

Value를 넣어야 하는 위치였으면 Value가 된다.

만약 순서를 맞추지않아 짝을 이루지 못한다면 내용이 꼬일것.

\*/

//반복문 돌면서 Account정보 저장

writer.StartArray();

{

writer.StartObject();

writer.String(L"AccountNo"); //Key

writer.Uint64(1111); //Value

writer.String(L"NickName"); //Key

writer.String(L"NICK1111"); //Value

writer.EndObject();

}

writer.EndArray();

//친구 목록

writer.String(L"Friend");

//반복문 돌면서 친구 정보 저장

writer.StartArray();

{

writer.StartObject();

writer.String(L"FriendNo");

writer.Uint64(1111);

writer.String(L"FromAccountNo");

writer.Uint64(2222);

writer.String(L"ToAccountNo");

writer.Uint64(1111);

writer.EndObject();

}

writer.EndArray();

//친구요청

writer.String(L"FriendRequest");

writer.StartArray();

{ //반복문 돌면서 저장

writer.StartObject();

writer.String(L"RequestNo");

writer.Uint64(1111);

writer.String(L"FromAccountNo");

writer.Uint64(2222);

writer.String(L"ToAccountNo");

writer.Uint64(1111);

writer.EndObject();

}

writer.EndArray();

writer.EndObject();

//이 문자열을 저장하거나 패킷을 쏜다.

pJson = StringJSON.GetString();

Size = StringJSON.GetSize();

//String객체의 GetString을 호출하면 포인터를 반환한다.

//이 포인터에 JSONString이 들어가있는것.

//그럼 우리는 파일에 저장을하든, 네트워크 패킷으로 쏘던 써먹을수 있게 되는것.

LoadData();

return 0;

}

대괄호에 대한 오버로딩이 되어있기 때문에 원하는 키값을 넣으면 데이터를 뽑아준다.

여기서 뽑아주는 데이터는 내부에서는 알고있지만 우리는 어떤 타입인지 알수없으므로 만능타입인 Value를 얻어 우리 용도에 맞게 사용한다.

만약 Account에 대한 Value값이 String이었다면 GetString(), Getint하면 int.

위에서는 Array역할을 하기로 했기 때문에 배열[]로서 접근한다.

**(ㄱ)**

**타입을 다르게 사용했을 경우**

**AccountValue를 array로 선언했고 AccountArray에 Value로 들어가는데 배열이 아니었다면 예외를 던지게 된다.**

**예외는 C++의 throw느낌으로 마구잡이로 던져지기 때문에 여기 account Key가 없다면 안에서 그냥 throw던지고 Crash가 난다.** **(=서버다운)**

하지만 값을 뽑을때마다 try-Catch를 다 받아낼수도 없고, Try-Catch문을 크게 묶어버리면 어디서 오류가 났는지 알 수 없으므로 아무런 의미가 없음

rapidjson::Value::ConstMemberIterator iter = Doc.FindMember("Account");

//멤버 찾을 수 없음

if (iter == Doc.MemberEnd())

{

DWORD err = 0;

//return;

}

따라서 위와같이 Find를 사용해 값이 존재하는지 확인한다.

Iter를 뱉기 때문에 반환값이 End값이라면 찾지 못했다는 의미이므로 우리의 예외처리 구문으로 넘어가야 할 것.

))

**BOM(Byte Order Mark) Code**

**BOM code는 어떤 텍스트 파일을 저장이 됐을때 해당 텍스트파일의 유니코드 인코딩을 명확하게 하기위해서 파일 가장앞에 들어가는 코드이다.**

대부분이 UTF-8로가고, 윈도우만이 UTF-16을 사용한다.

**(거의대부분의 C++오픈소스들은 UTF-16을 쓰지않는다. 다 UTF-8이 기반이다.)**

fopen\_s(&fp, "FileName.txt", "w,ccs=UTF-8");



위와같이 BOM코드를 설정하지않으면 기본은 UTF-16으로 가게된다.

설정 후 파일을 오픈하면 최초 파일포지션은 BOM코드 뒤를 가리킨다.

**처음부터 설정을 넣고 read함수를 사용한다면 문제가 없을 것이나 바이너리로 오픈하면 BOM코드를 읽을것이기 때문에 위험하다.**

만약 BOM코드가 포함된채로 rapidjson document에 넣으면 이를 식별할 수 없으므로 즉시 예외가 던져져 crash가 나게된다.

**따라서 테스트할때는 반드시 파일에서 UTF의 BOM코드가 없이 저장을 해야한다.**

**UTF8toUTF16() 함수**

bool UTF8toUTF16(const char\* Text, WCHAR\* Buff, int BuffLen)

{

int Re = MultiByteToWideChar(CP\_UTF8, 0, Text, strlen(Text), Buff, BuffLen);

if (Re < BuffLen)

Buff[Re] = (WCHAR)L"\0";

return true;

}

UTF8로 얻어진 데이터를 UTF16으로 바꾸고있다.

우리로직에서 직접 UTF8toUTF16()함수를 구현해 UTF8을 UTF16으로 변환시킨 것.

**어차피 웹에서는 UTF-16을 사용하지못하기 때문에 (HTTP는 다 UTF-8임) 다른퍼블리셔랑 연동되도 다 UTF-8이다.**

그래서 이런식으로 하나하나 바꿔주는 작업이 들어간다. 불편해보이지만 어쩔수 없음.

앞으로 제이슨은 많이 사용이 되기 때문에 어떤 라이브러리가 됐든 사용할수있어야 한다.

**실제로 현업에서는 어떻게 요청이 오냐에 따라 다를 것.**

**JSON 스트링을 C++ 에서 읽어들이기.**

**(C++(UTF-8) -> JSON(UTF-8))**

{ "Account":

[

{"AccountNo":1,"Nickname":"테스트계정1"},

{"AccountNo":2,"Nickname":"테스트계정2"}

],

"Friend":[],

"FriendRequest":

[

{"RequestNo":1,"FromAccountNo":1,"ToAccountNo":2,"Time":1437344611},

{"RequestNo":2,"FromAccountNo":1,"ToAccountNo":3,"Time":1437344611}

]

}

이렇게 된 JSON 을 읽어들이는 예제 입니다.

Account 는 배열이며 배열의 요소에는 { AccountNo / Nickname } 이 묶이는 객체들 입니다.

각 데이터의 의미는 없습니다. 단지 변수,배열,객체를 표현하는 json 문법에 집중 하시면 됩니다.

**애초에 UTF-8로 저장된파일을 UTF-8로 읽어들이는 것으로, BOM코드변경이 필요없음.**

**만약 파일이 UTF-16으로 저장되어 UTF-16으로 읽어와야한다면?**

**1. Value 와 Document 의 인코딩을 UTF16 으로 변경해야한다.**

**typedef GenericValue<UTF16<> > Value;**

**typedef GenericDocument<UTF16<> > Document;**

**이렇게 일부 typedef 의 인코딩을 변경시키면 UTF16 문자열 사용이 가능 합니다.**

#include "rapidjson/document.h"

#include "rapidjson/writer.h"

#include "rapidjson/stringbuffer.h"

using namespace rapidjson;

void JsonLoad(void)

{

char \*pJson = 파일 또는 네트워크로 받은 JSON 문자열

지금은 UTF - 8 을 전제로 함

Document Doc;

Doc.Parse(pJson); // Document 로 Json 텍스트 파싱.

UINT64 AccountNo;

const char \*pNickname;

Value &AccountArray = Doc["Account"]; // 첫번에 Account 객체 접근

// rapidjson::Value 로 받아서 사용

for (SizeType i = 0; i < AccountArray.Size(); i++)

{

Value &AccountObject = AccountArray[i];

AccountNo = AccountObject["AccountNo"].GetUint64();

pNickname = AccountObject["Nickname"].GetString();

// 읽혀진 데이터 사용

}

Value &FriendArray = Doc["Friend"];

for (SizeType i = 0; i < FriendArray.Size(); i++)

{

Value &FriendObject = FriendArray[i];

FriendNo = FriendObject["FriendNo"].GetUint64();

FromAccountNo = FriendObject["FromAccountNo"].GetUint64();

ToAccountNo = FriendObject["ToAccountNo"].GetUint64();

// 읽혀진 데이터 사용

}

이와 같은 형태로 Key 문자열을 [“ “] 로 접근하여 손쉽게 사용이 가능 합니다.

**C++ 데이터를 JSON 으로 변환하기**

**(UTF16 -> UTF8 저장)**

**UTF16으로 선언해 8로 저장하고있다. (그냥처음부터 8로 선언해서 하면안됨?)**

**그런데 JSON파일을 UTF-16으로 저장해야 한다면,**

**1. StringBuffer 의 인코딩 타입을 UTF16 으로 지정 합니다.**

**typedef GenericStringBuffer<UTF16<> > StringBuffer;**

**2. Writer 의 Input 인코딩과 Output 인코딩을 모두 UTF16 으로 지정 합니다.**

**Writer<StringBuffer, UTF16<>,UTF16<>> writer(StringJSON);**

StringBuffer StringJSON;

Writer<StringBuffer, UTF16<>> writer(StringJSON);

writer.StartObject(); // {

writer.String(L"Account"); // { “Account”

writer.StartArray(); // { “Account”:[

{ // 반복문 돌면서 Account 정보 저장

writer.StartObject(); // { “Account”:[{

writer.String(L"AccountNo"); // { “Account”:[{“AccountNo”:

writer.Uint64(1111); // { “Account”:[{“AccountNo”:1111

writer.String(L"Nickname"); // { “Account”:[{“AccountNo”:1111, “Nickname”:

writer.String(L”닉네임”); // { “Account”:[{“AccountNo”:1111, “Nickname”: “닉네임”

writer.EndObject(); // { “Account”:[{“AccountNo”:1111, “Nickname”: “닉네임”},

}

writer.EndArray(); // } 추가

writer.String(L"Friend"); // ……

writer.StartArray();

{ // 반복문 돌면서 친구정보 저장

writer.StartObject();

writer.String(L"FriendNo");

writer.Uint64(pFriend->No);

writer.String(L"FromAccountNo");

writer.Uint64(pFriend->FromAccountNo);

writer.String(L"ToAccountNo");

writer.Uint64(pFriend->ToAccountNo);

writer.EndObject();

}

writer.EndArray();

writer.String(L"FriendRequest");

writer.StartArray();

{ // 반복문 돌면서 저장

writer.StartObject();

writer.String(L"RequestNo");

writer.Uint64(pFriend->No);

writer.String(L"FromAccountNo");

writer.Uint64(pFriend->FromAccountNo);

writer.String(L"ToAccountNo");

writer.Uint64(pFriend->ToAccountNo);

writer.EndObject();

}

writer.EndArray();

Writer.EndObject();

const char \*pJson = StringJSON.GetString(); // 이 문자열을 저장하거나 패킷을 쏨

**UTF-8, UTF-16 위코드 코드상에서 증명해보기. 못했음..**

**친구관리 서버만들기**

**목적1. 우리의 Select모델이 조금더 안전하게 잘 만들어져 있는가 테스트.**

**목적2. 데이터를 관리하는 연습을 해보자.**

(친구요청 - 친구요청 받은것), (친구관계)

**둘다 양쪽에 데이터가 들어가야한다. 각기 친구의 정보를 자료구조로 들고 있는 형태는 객체지향 설계로, 프로그래밍적 접근이다. (객체 회원에 데이터를 몰아넣는 형태)**

**DB를 설계하거나 데이터끼리 관점에서는 데이터끼리 모아놓아야 한다. (ex액셀)**

**(데이터는 친구의 목록, 요청보낸목록(=요청 받은 목록))**

친구목록을 액셀로 관리할때, 제1정규화의 오류를 범해서는 안된다.



이는 결국 하나의 컬럼에 하나의 데이터가 들어가야 한다는 의미가되며, 중복된 데이터는 허용하지않는다.

**데이터 베이스가 이러한 정규화 규칙을 세운 이유는, 하나의 독립된 데이터가 아니기 때문에 데이터로서의 가치가 떨어지기 때문이다.(추가작업 필요)**

**최종적으로 필요한 것 : 회원목록, 친구목록, 요청보낸 목록(=요청 받은 목록)**

회원목록에는 오직 회원정보(Account번호와 ID)만 존재하며, 다른쪽에는 친구관계 테이블이 존재한다.

**친구관계 테이블**

회원이라는 데이터가있는 시트가 있다고 가정한다.

|  |  |
| --- | --- |
| ID | 친구 |
| 1 | 2,3,4 |

**위와같은 방식이 제1정규화의 오류**

|  |  |
| --- | --- |
| ID | 친구 |
| 1 | 2 |
| 1 | 3 |

수정된 결과. **하지만 이렇게 갈 경우 2번의 입장에서도 1번이 친구이고, 3번의입장에서도 1번이 친구**이다.

**따라서 ID/친구로 나뉘는게 아니라 누가 누구한테 친구인지로 바꾸고, 아래 두가지 방법중 선택.**

* **1이랑 2가 친구일때 2와 1을 또 넣는것.**

**(같은 정보의 데이터가 두번 들어가게되는것)**

|  |  |
| --- | --- |
| A | B |
| 1 | 2 |
| 2 | 1 |

* **검색을 양쪽다 하는것**

**(1번과 친구를 찾고싶다면 A에서도 1을검색, B에서도 1을 검색)**

|  |  |
| --- | --- |
| A | B |
| 1 | 2 |
| 3 | 1 |

**객체기반 vs 데이터기반**

**지금 위와같은 데이터지향 구조가 객체지향에 비해 장점이 있는지 크게 체감되지 않을 수있다.**

**객체설계 기반으로 가게되면 회원에 대한 친구 요청정보와 친구정보가 내부에 다 들어있기 때문에, 특정 회원만 찾아낸다면 내부에 모든 정보가 존재한다.**

**하지만 데이터설계 기반인 경우 친구목록이라는 데이터에 전체 회원에 전체 친구목록이 모두 들어있는것으로, 특정회원의 친구를 얻고 싶다면 모든회원을 검색해야 한다는 얘기가 된다.**

**이는 결국 객체설계 기반에 비해 성능면에서 떨어지나, 데이터설계 기반시 안정성이 올라간다.**

(DB설계에서는 상태이상이라는 표현을 쓰는데 이는 값을 변경시키거나 갱신하는과정에서 이상이 생길만한 과정을 차단시킨다는 것이다.)

**객체지향의 설계로 컬럼하나에 여러개의 데이터가 들어간다는 형식으로 설계가 되면 약간의 실수로 인해서 데이터가 어긋나는 경우가 생길 수 있고, 애초에 이런 위험성을 차단시키는 것이 DB의 정규화 규칙이다.**

(만약 DB를 사용하는 것이 아니라 내코드로 데이터를 관리하는 서버나 매니저를 만든다면 어떤 형태로 만들어도 상관없을 것이나, 객체를 관리하는 방식은 위험하기 때문에 권장하지 않는다.)

**친구회원에 대한 정보들만 관리하는 컨테이너,**

**친구관계만 표현하는 컨테이너,**

**친구의 요청을 보내고 받은 컨테이너.**

위와 같이 관리하는것이 훨씬더 유연하고 안정성이 높으며 관리하기 용이하다.

(검색에는 좀더 빈번함이 있어 성능이 떨어질 수 있음)

((

**DB와 NOSQL**

**DB는 RDBMS(관계형 데이터베이스 관리시스템(relational database management system)으로,**

**서로 테이블과 테이블이 관계로 엮여 서로의 역할과 기능이 정해져있음을 뜻한다.**

**이같은 틀을 깨고(DB의 단점을 커버하기위해) 나온것이 NOSQL이다.**

NOSQL는 JSON으로 받기 때문에 컬럼의 개념이 없고 데이터를 넣다가 형태를 바꿔도 상관없이 단순히 집어 넣기만 한다.

**이는 기능이 없는대신 자유도가 높아진다고 해석할수있다.**

**데이터 끼리의 연관성이 없는 블로그나 SNS에서 활용도가 아주 높으나, 게임은(일반적인 stateful) 데이터끼리의 연관성이 아주 높기 때문에 대부분 NOSQL을 하지않는다.**

(모바일 웹서버일 경우에는 NOSQL을 사용)

DB에는 분산이나 확장이 자동화되있고 데이터 규격도 따로 없으므로 쓰는것에 대한 규칙이 매우 자유롭다.

**싱글게임에 가까운 비동기 온라인게임(stateless서버)는 싱글게임에 가까운 형태**나, **stateful서버라고해도 MMORPG형식이 아닌 배틀그라운드나 아니면 포트나이트 같은 경우 무한히 확장(스케일아웃) 되고 유저간의 상호작용이 다양하지 않은 형태이기에** NOSQL을 쓰기도 한다.

**위와같이 유저간의 상호작용이 얼마없고 유저간의 데이터관계가 복잡하게 얽히지 않은 경우 (stateless 웹으로하는 비동기 온라인 게임)는 사용하기도 하지만,**

**현재에 이르러서는 NOSQL이 적합한 부분들이 정해졌고 게임서버 DB에 NOSQL을 적용하는 것은 권장하지 않는다.**

))

**데이터 기반에서의 검색 방식**

* **AB-BA방식으로 같은 데이터를 두번저장 (데이터양 증가, 데이터오류확률 증가)**
* **From- to의 방향을 없애고 AB를 하나만 저장 (검색 두번필요)**

서버를 끌때 JSON으로 저장한다. (지금 DB도 없고 JSON도연습할겸)

JSON으로 저장했다가 프로그램이 켜질때 JSON데이터를 로딩해서 예전과 똑같은상황 만들어지게끔 한다.

**서버에서 사용되는 자료구조형태**

**자료구조중 map이나 list로 관리를 해야할텐데 사실 list는 쓸 이유가 없고, 지금은 정렬된 순회도 필요없으므로 unordered\_map을 권장한다.**

서버에서 대부분은 unordered\_map(hash)이 기본이 된다.

만약에 MO류로 방개념이 들어간다면 list를 사용해도 된다.

**DB에서는 우리가 원하는대로 From칼럼과 To 칼럼에 Index를 저장할수 있고 어느쪽으로 넣던 상수값의 검색속도가 나오기 때문에 문제가 없으나,**

(삽입/삭제할때 이미 index를 다 만들어둔다)

현재 예제를 진행하는 우리입장에서는 애매할 수 있다.

**A와 B가 친구인데 AB로만 넣겠다고 했을때 (From - To구분짓지 않는 방식) 키가 하나가 되기 때문에 Value를 기준으로 검색할 방법이 없다.**

따라서 하나의 컨테이너가 더 추가되어야 한다.

(Key값이 From인 컨테이너와 Key값이 To인 컨테이너, 메인 친구관리를 전담하는 데이터, 회원목록데이터.)

**- A 와 B 가 친구라면(메인관리)  
             FriendMap - Key:No / Value:{No, A - B}**

**여기서 NO는 친구관리에 대한 고유값이다.**

**(A와 B가 친구라면 1번, A와 C가 친구라면 2번...) 과같은 형태로 데이터에 고유번호 부여.**

**FriendMap에서 그냥 데이터를 가져도 되겠지만, 친구관계에 대한 키값만 넣어보자는 뜻이다.**

**그리고 실제 그키에 대한 친구정보는 따로 관리되는 유일한 오리지날 데이터.**

**이렇게 한번더 분리하는 이유는, FriendMap이 포인터를 들고있다는 것이 안정성 면에서 좋은 설계가 아니기 때문**이다. (여기저기서 다 포인터를 들고있으면 누가진짜인지 알 수 없음)

**때문에 실질적인 친구정보 객체는 하나만 들고 나머지는 찾아갈수있는 키값만 들고있는것이다.**

**FriendMap\_FromIndex Key:FromAccountNo / Value:No  
            FriendMap\_ToIndex Key:ToAccountNo / Value:No**

Index가 중복될수있으므로 multimap을 사용해야 할것.

DB의 경우 조건걸고 쿼리를 날리면 알아서 해준다.

따라서 DB설계가 충분하다면 DB로 가는게 낫다. 결국 DB의 목적이 데이터관리와 빠른검색이기 때문에.

왠만해서는 빠르지만 너무 많은 자료일경우 느려질수있는데, 제일 좋은 것은 DB차원에서 성능을 높이는 것이다.

DB차원에서 분산을 쉽게할수있는 구조적인 방법이 존재하기 때문에 실제로 친구관리 서버를 따로 만들일은 없을 것이나 연습겸 진행한다.

**Redis**

DB에서 검색하기는 너무 느린데 stateful서버로 CPP로 만들기는 너무귀찮은 경우 Redis를 사용한다.

**레디스**(**Redis**)는 메모리데이터 베이스로, NOSQL의 형태로 보는데 Key-Value형태이다.

맵을 하나두고 어떤키에 Value를 쿼리로 보내면 맵에 넣은것처럼 메모리에 넣어준다.

그리고 키값을 통해 Value를 다 뽑아준다.

**기본적으로 Key-Value형태를 관리해주는 구조이기 때문에 간단한 랭킹 서버/닉네임서버를 레디스로 해결한다.**

**우리는 그런느낌의 서버를 수동으로 만들어 보자는 느낌.**

이런것조차 만들기 귀찮으니 이러한 솔루션이 나오는것..

웹개발자 입장에서는 스크립트를 가져다쓰기 때문에 솔루션이 없으면 해결이 되지않는다. 하지만 CPP서버 개발자들의 경우는 보통 이러한 것들을 게임에 특화되게 직접만들었다.

이러한 솔루션이 점점 대중화되다 보니 stateful서버, Cpp서버인 컴파일 언어로도 넘어온 것이다.

echo스트레스 테스트를 해볼건데 이걸로 select모델에 대한 마지막 검증까지하고 그다음에 이제 TCP FIGHT MO서버를 만드는 방식으로 갈것..

==================================================================================

친구시스템 종류  
  
팔로워방식 (트위터)  
  
신청즉시 팔로워가 됨  
상대방의 정보를 볼 뿐 쌍방향 컨텐츠는 없음.  
  
  
친구방식 (페이스북)  
  
친구요청 > 수락 or 거부  
  
요청과 수락이라는 과정이 있으며  
친구관계에선 서로 컨텐츠를 공유하며 플레이  
  
  
  
초대시스템  
  
카카오톡처럼 특정 플렛폼에 붙을 경우  
해당 플렛폼의 다른 유저들을 해당 게임으로 초대 함  
  
단순히 메시지를 보내는 초대라면 아주 간단하지만  
일정시간 내에 초대 횟수 제한, 그리고 초대에 대한 보상처리가 대부분 들어감.  
  
  
  
# 친구관계 처리를 위한 메시지(패킷) 종류  
  
  
FriendList - 친구목록  
  
FriendList\_Request - 친구목록\_요청보낸거  
  
FriendList\_Reply - 친구목록\_요청받은거  
  
FriendRemove - 친구관계 끊기

FriendRequest - 친구 요청하기  
  
FriendCancel - 친구 요청 보낸거 취소  
  
FriendDeny - 친구 요청 거부  
  
FriendAgree - 친구 요청 수락  
  
  
# 친구관계 테이블(저장소)  
  
Account  
  
AccountNo (64bit int)  
ID (최대 19자의 문자열)  
  
  
  
Friend  
  
          FromAccountNo  
          ToAccountNo  
  
          Time  
  
            - A 과 B 가 친구라면  
             FriendMap - Key:No / Value:{No, A - B}  
             FriendMap\_FromIndex Key:FromAccountNo / Value:No  
             FriendMap\_ToIndex Key:ToAccountNo / Value:No  
  
  
  
FriendRequest  
  
            FromAccountNo  
            ToAccountNo  
            Time  
  
----------------------------------------------------------------------  
  
만들 프로그램  
  
1. 회원가입 - 닉네임 문자열 받음, AccountNo 서버에서 생성.  
  
일단 메모리에만 저장 되도록 함.  
  
테스트를 위해서 10명 내외의 회원을 코드상에서 하드 코딩으로 기입  
  
이는 다음 시간에 JSON 으로 저장되도록 바꿀 것임  
  
  
2. 친구관련 기능 구현 (친구목록, 친구요청, 친구수락,  
  
위의 친구관련 테이블을 Map 또는 List 로 관리

==========================================================================

**<6.5>**

진행하는 예제에서 로그인이라는 개념은 없다.

[내가 누구다] 라는 것을 프로토콜 상대에서 생략한 상태이다.

**실제로 우리가 기능을 분산시켜서 이러한 형태의 서버를 만든다고 한다면 누가 누구한테 누구의목록을 이라는 항목들이 항상 들어가야 한다.**

//--------------------------------------------------

// 친구관리 프로토콜

//

//

//--------------------------------------------------

#define dfNETWORK\_PORT 25000

#define dfNICK\_MAX\_LEN 20

#define dfPACKET\_CODE 0x89

//------------------------------------------------------

// 패킷헤더

//

// | PacketCode | MsgType | PayloadSize | \* Payload \* |

// 1Byte 2Byte 2Byte Size Byte

//

//------------------------------------------------------

**서버간의 통신에서는 서버를 믿고 쓸 것이기 때문에 최대한 간략하게 가게된다. 때문에 체크섬은 제외한다. (PacketCode는 원래 빼야함)**

이후에도 클라이언트를 연결시킬 라이브러리와 서버간의 통신 라이브러리 만들텐데 서버간의 라이브러리는 믿고간다. 외부에서 조작 할수있다는 가정을 고려하지않음.

#pragma pack(push, 1)

struct st\_PACKET\_HEADER

{

BYTE byCode;

WORD wMsgType;

WORD wPayloadSize;

};

#pragma pack(pop)

//------------------------------------------------------------

// 회원가입 요청

//

// {

// WCHAR[dfNICK\_MAX\_LEN] 닉네임

// }

//------------------------------------------------------------

#define df\_REQ\_ACCOUNT\_ADD 1

//------------------------------------------------------------

// 회원가입 결과

//

// {

// UINT64 AccountNo

// }

//------------------------------------------------------------

#define df\_RES\_ACCOUNT\_ADD 2

// ## 컨텐츠 서버와의 통신하는 서버 이므로 본래 로그인 이라는 개념은 필요가 없습니다.

// ## 그런데 로그인 이라는 기능이 있는 이유는 테스트 클라이언트에서 매번 요청자 AccountNo 를 입력하기가

// ## 번거로운 관계로 연결된 세션에 대하여 '로그인' 이라는 과정을 통해서 요청자의 AccountNo 를 서버측 세션에 보관 해두기 위해서 입니다.

// ## 그래서 로그인 후에는 친구요청,수락,거부 등의 모든 행동 메시지에 요청자 AccountNo 가 생략되고 있습니다.

//

// ## 실제 서비스 목적의 서버로서 개발을 한다면 회원 로그인 이라는 과정은 없어져야 하며,

// ## 친구 목록 Req / 친구요청 Req / 수락 Req 등 서버에게 요청하는 보든 과정에 요청자 AccountNo 가 포함 되어야만 합니다.

//------------------------------------------------------------

// 회원로그인

//

// {

// UINT64 AccountNo

// }

//------------------------------------------------------------

#define df\_REQ\_LOGIN 3

//------------------------------------------------------------

// 회원로그인 결과

//

// {

// UINT64 AccountNo // 0 이면 실패

// WCHAR[dfNICK\_MAX\_LEN] NickName

// }

//------------------------------------------------------------

#define df\_RES\_LOGIN 4

//------------------------------------------------------------

// 회원리스트 요청

//

// {

// 없음.

// }

//------------------------------------------------------------

#define df\_REQ\_ACCOUNT\_LIST 10

//------------------------------------------------------------

// 회원리스트 결과

//

// {

// UINT Count // 회원 수

// {

// UINT64 AccountNo

// WCHAR[dfNICK\_MAX\_LEN] NickName

// }

// }

//------------------------------------------------------------

#define df\_RES\_ACCOUNT\_LIST 11

//실제 상용화할 서버를 만든다면 회원리스트를 결과를 요청하거나 받는 이러한 메시지들을 만들지않음. 몇만명일줄알고? 애초에 없음

//------------------------------------------------------------

// 친구목록 요청

//

// {

// 없음

// }

//------------------------------------------------------------

#define df\_REQ\_FRIEND\_LIST 12

//------------------------------------------------------------

// 친구목록 결과

//

// {

// UINT FriendCount

// {

// UINT64 FriendAccountNo

// WCHAR[dfNICK\_MAX\_LEN] NickName

// }

// }

//------------------------------------------------------------

#define df\_RES\_FRIEND\_LIST 13

//------------------------------------------------------------

// 친구요청 보낸 목록 요청

//

// {

// 없음

// }

//------------------------------------------------------------

#define df\_REQ\_FRIEND\_REQUEST\_LIST 14

//------------------------------------------------------------

// 친구목록 결과

//

// {

// UINT FriendCount

// {

// UINT64 FriendAccountNo

// WCHAR[dfNICK\_MAX\_LEN] NickName

// }

// }

//------------------------------------------------------------

#define df\_RES\_FRIEND\_REQUEST\_LIST 15

//------------------------------------------------------------

// 친구요청 받은거 목록 요청

//

// {

// 없음

// }

//------------------------------------------------------------

#define df\_REQ\_FRIEND\_REPLY\_LIST 16

//------------------------------------------------------------

// 친구목록 결과

//

// {

// UINT FriendCount

// {

// UINT64 FriendAccountNo

// WCHAR[dfNICK\_MAX\_LEN] NickName

// }

// }

//------------------------------------------------------------

#define df\_RES\_FRIEND\_REPLY\_LIST 17

//------------------------------------------------------------

// 친구관계 끊기

//

// {

// UINT64 FriendAccountNo

// }

//------------------------------------------------------------

#define df\_REQ\_FRIEND\_REMOVE 20

//------------------------------------------------------------

// 친구관계 끊기 결과

//

// {

// UINT64 FriendAccountNo

//

// BYTE Result

// }

//------------------------------------------------------------

#define df\_RES\_FRIEND\_REMOVE 21

#define df\_RESULT\_FRIEND\_REMOVE\_OK 1

#define df\_RESULT\_FRIEND\_REMOVE\_NOTFRIEND 2

#define df\_RESULT\_FRIEND\_REMOVE\_FAIL 3

//------------------------------------------------------------

// 친구요청

//

// {

// UINT64 FriendAccountNo

// }

//------------------------------------------------------------

#define df\_REQ\_FRIEND\_REQUEST 22

//------------------------------------------------------------

// 친구요청 결과

//

// {

// UINT64 FriendAccountNo

//

// BYTE Result

// }

//------------------------------------------------------------

#define df\_RES\_FRIEND\_REQUEST 23

#define df\_RESULT\_FRIEND\_REQUEST\_OK 1

#define df\_RESULT\_FRIEND\_REQUEST\_NOTFOUND 2

#define df\_RESULT\_FRIEND\_REQUEST\_AREADY 3

//------------------------------------------------------------

// 친구요청 취소

//

// {

// UINT64 FriendAccountNo

// }

//------------------------------------------------------------

#define df\_REQ\_FRIEND\_CANCEL 24

//------------------------------------------------------------

// 친구요청취소 결과

//

// {

// UINT64 FriendAccountNo

//

// BYTE Result

// }

//------------------------------------------------------------

#define df\_RES\_FRIEND\_CANCEL 25

#define df\_RESULT\_FRIEND\_CANCEL\_OK 1

#define df\_RESULT\_FRIEND\_CANCEL\_NOTFRIEND 2

#define df\_RESULT\_FRIEND\_CANCEL\_FAIL 3

//------------------------------------------------------------

// 친구요청 거부

//

// {

// UINT64 FriendAccountNo

// }

//------------------------------------------------------------

#define df\_REQ\_FRIEND\_DENY 26

//------------------------------------------------------------

// 친구요청 거부 결과

//

// {

// UINT64 FriendAccountNo

//

// BYTE Result

// }

//------------------------------------------------------------

#define df\_RES\_FRIEND\_DENY 27

#define df\_RESULT\_FRIEND\_DENY\_OK 1

#define df\_RESULT\_FRIEND\_DENY\_NOTFRIEND 2

#define df\_RESULT\_FRIEND\_DENY\_FAIL 3

//------------------------------------------------------------

// 친구요청 수락

//

// {

// UINT64 FriendAccountNo

// }

//------------------------------------------------------------

#define df\_REQ\_FRIEND\_AGREE 28

//------------------------------------------------------------

// 친구요청 수락 결과

//

// {

// UINT64 FriendAccountNo

//

// BYTE Result

// }

//------------------------------------------------------------

#define df\_RES\_FRIEND\_AGREE 29

#define df\_RESULT\_FRIEND\_AGREE\_OK 1

#define df\_RESULT\_FRIEND\_AGREE\_NOTFRIEND 2

#define df\_RESULT\_FRIEND\_AGREE\_FAIL 3

//------------------------------------------------------------

// 스트레스 테스트용 에코

//

// {

// WORD Size

// Size 문자열 (WCHAR 유니코드)

// }

//------------------------------------------------------------

#define df\_REQ\_STRESS\_ECHO 100

//------------------------------------------------------------

// 스트레스 테스트용 에코응답

//

// {

// WORD Size

// Size 문자열 (WCHAR 유니코드)

// }

//------------------------------------------------------------

#define df\_RES\_STRESS\_ECHO 101

서버를 만들고 스트레스 테스트를 할 때, 목표로 하는 기준이 있을 것이다.

(ex 전세계면 1,2억, 국내면 최소 3천만명 등)

우리는 단순하게 네트워크 proc를 테스트하는 것이 아니기 때문에 (친구요청/수락/거절/끊기)를 수행하는 봇을 만들어서 테스트를 해야하지만 봇을 만들기가 쉽지 않다.

**우리의 목적은 Select코드가 안전하게 송수신이 잘되는가 확인하는것에 있으므로, 에코테스트로 진행한다.**

**의미없는 Echo Test방식**

어떤클라이언트가 어떤 사이즈의 문자열을 줄것이고, (길이제한은 512로 설정되었을 것)이를 서버가 받으면 그대로 돌려보낸다.

**그대로 돌려보낼때는 새로운 직렬화버퍼에 넣어서(마샬링하여) 보내야한다.**

((

마샬링도 하지않고 그대로 돌려보낸다면 의미없다.

예전에 CGC엔진 에코테스트도 같은 맥락이다.

**에코/스트레스 테스트를 할 때 받은 버퍼를 Copy를 완전히 0으로 만들어서 Send까지 보내는 방식은** (우리가 만든 네트워크 라이브러리에 직렬화버퍼도 메모리풀의 참조 카운터 방식으로 전환하고 링버퍼도 모두 없애서 그대로 네트워크 송신버퍼까지) **성능이 매우 좋게 나올것이다.**

**하지만 실제 게임서버에서는 아무도 이런방식으로 사용하지 않기 때문에 이와 같은 방식은 아무의미없는 보여주기식 테스트이다.**

))

**우리는 항상 Echo테스트를 할 때 Loacal로 뽑아내고 새로운 직렬화버퍼를 뽑아서 다시 보내는 것을 원칙으로 한다.**

**Echo Test로 테스트 할 수 있는것**

일반적인 Echo테스트의 경우 더미클라이언트가 서버에 무작위로 연결하여 에코를 쏟아부을 것이다.

이때 우리가 테스트할 수 있는 것은,

**1. 송수신이 잘되는가 (그대로 데이터가 나에게 돌아왔는지 확인)**

**돌아온 데이터가 잘못됐거나 보냈는데 안오거나 다른클라한테로 갔다면 오류가 존재하는 것.**

(이때 보낸데이터가 다른 클라한테 간 것은 잘못된 데이터가 온것과 똑같이 처리할수 있을 것)

**2. 성능은 어떤가**

**더미클라이언트가 대량으로 서버에 연결해서 쏟아부을 때, 서버가 똑같이 응답하므로 이걸가지고 가장 기본적인 성능 테스트를 실시한다.**

**네트워크 라이브러리**

지난시간에 설명한 Redits 같은 걸로 대체를 할수도 있겠다.

우리 여기에 최종적인 목표는 게임서버, 채팅서버를 만들어서 연동하는게 목표인데, 일단은 네트워크 라이브러리를 만들어야 한다.

**네트워크 라이브러리가 완성되었다면 서버는 단지 찍어내면 될뿐, 큰 문제가 아니다.**

(MMORPG의 서버라면 컨텐츠가 워낙많아서 오래걸리는것으로, 보통은 길면 3일에 하나, 짧으면 하루에 하나씩 찍어내야 어느정도 따라잡는다.)

어떠한 기능이 필요할떄 솔루션이 있나 찾아서 사용법을 익히고, 설치하고, 테스트하고, 라이브러리 클라이언트쪽을 받아 연동하고..

하는 시간에 만드는게 훨씬더 빠르고 효율적일 수 있다.

(회사에 들어갔을 때 결국은 사수스타일에 맞게 결정나게 될 것)

**테스트의 최종목표**

서버를 만들어서 테스트할떄, 단순히 기능을 테스트하는 것은 포트폴리오로서도, 테스트로서도 큰 가치가 없다.

**우리가 봇을 유저처럼 만들어서 테스트하기에는 어려우므로, 일단 하나의 테스트를 통해 예측된 지표를 만들어 봐야한다.**

**목표로하는 기준치(인원)로 먼저 메모리부터 확인한다.**

(몇 명 넣었을 때 어느정도의 메모리가 사용되는지)

우리는 현재 싱글스레드 기반이므로 한명의 요청의 성능이 결국은 TPS(Transaction per second : 초당 트렌젝션 속도)가 되고, **일단 네트워크 레이턴시는 제쳐두고 네트워크 서버측에서만 테스트했을 때 어느정도의 시간(TPS)이 걸리는지 평균값을 체크한다.**

(친구요청을 했을 때, 거절을 했을 때, 수락을 했을 때에 걸리는 평균값)

(TPS가 1u/s일경우 1초에 백만건, 1m/s일경우 천만건.)

**다음으로 실제 라이브서비스에 지금만든 서버를 적용할수있는지 (우리가 구한 TPS 평균치를 가지고) 체크해야한다.**

어차피 실제로 적용할때까지는 정확한 수치를 알아낼 수 없으므로, 현재 라이브서비스 데이터를 기반으로 예측해본다.

**지금은 친구서버이므로, 동접이 n명일 때 친구관련 요청을 초당 얼마나 하는지 구해보고 어느정도 TPS평균치로 커버가 가능한지 생각해본다.**

실제로 적용했을 때는 예측대로 맞아떨어지지 않을 수도 있지만, 일단 그럴싸해야함.

이정도의 자료만 나와준다면 포트폴리오에 한페이지 들어갈수 있을 것이다.

(ex 이렇게 설계를했고 이렇게 데이터를 관리했고 이렇게 헀는데 이렇게 성능테스트해봤더니 동접 100만명의 유저를 커버할 서버가 되었다. 추가로 자료구조 A를 썼는데 이렇고 B로 썼을때는 이랬다.)

지금은 싱글스레드의 Select 기반이지만, 멀티스레드의 경우 동기화 이슈가 걸릴 것이고 동기화의 유무에 따라 세분화시켜서 따로 뽑아야 할것.

**데이터관리**

DB설계와 최대한 흡사한 모양새로 일단 간다면,

**먼저 친구관계가 성립된 관계 고유번호를 부여하여 저장한다.**

(CPP의 프로그래머 입장이라면 포인터로 번호느낌이 되어도 상관없을 것)

결국은 회원자체, 친구정보, 친구요청데이터에 대한 자료구조 세덩어리가 나오게 될 것.

**Client Map**

지금 예제에서는 유니캐스트 밖에 없어서 유저번호를 검색할 일이 없기 때문(??)에 키를 소켓으로 잡았다.

(예전에 진행했던 ‘내가 속한 방에 뿌린다’와 같은 개념자체가 없기 때문)

**Account Map**

회원의 고유번호가 키가 되고, 회원이 Value가 될 것.

추가로, 친구에 대한 정보를 관리하는 맵이 나온다.

**AB-BA방식을 사용한다면 AccountNo(고유번호)가 Key가 될것이다.**

**이게아닌 AB(Non From-TO)관계를 사용한다면 이 자료구조 내부에는 회원정보는 없을 것이다.**

**그리고 이를 검색하기위한 Index역할을 하는 자료구조가 또 만들어 져야할 것**

**이것은 삽입삭제/검색의 빈도에 따라선택을 해야한다.**

특정 아이템 고유키를 가지고 Index를 걸었다고 가정한다. 이때, 아이템 종류에 따라 검색을 해야하거나, 아이템 요소안의 능력치를 기준으로 검색을 해야할 일이 생겼을 때는 그 검색을 빠르게 하기위한 별도의 보조 Index역할이 되는 해시테이블이 나와야 할것이다.

그런데 이러한 보조 Index역할이 되는 자료구조(ex unordered\_map)이 4개가 될경우 데이터하나를 삽입하는데 실제 메인컨테이너까지 포함해서 5개의 자료구조에 추가를 해줘야 하며, 삭제할때도 마찬가지일 것이다.

**이때 검색의 횟수보다 삽입삭제가 더 빈번하여 부담이 크다면 이는 잘못된 설계.**

**Request Map**

(생략..)

여기도이제 한가지방법은 리퀘스트맵에는 메인컨테이너 니가 실제 리퀘스트에 대한 포인터와 리퀘스트에 대한 고유번호가 여기서 관리되고 있고 나머지는 인덱스 역할로 안전하게 간다고 한다면 이것일 것. From기준. 키가 FromAccount. Value는 이 리퀘스트에 대한 고유번호.

근데 이렇게 되면 리퀘스트 고유번호를 얻어서 또 검색해야되니까 여기에 대한 방법은 아예 포인터를 넣는 방법이 될수도 있기도 하다.

근데 이렇게되면 안전하게 써야된다는 뜻이고.

**추가로 로직을 보면서 설명..**

(생략…)

데이터 로딩할때는 AddCount. 신규 만들때는 CreateAccount.

고유번호가 부여되있느냐 안되있느냐에 차이기 때문에 나뉘었으나, 실제로 이런 서버를 만든다면 DB에서 모두 긁어올 것이다.

CreateAccount라는 개념도 존재하지 않을겁니다 사실은.

그리고 친구 정보를 삽입하는거. 지금은 두개의 Account가 들어왔을 때 이친구를 친구로서 넣는 것이기 때문에 제가 선택한 것은 봤던 것처럼 친구정보를 하나를 셋팅을 동적할당 하고요

여기다 정보넣고 friendmap에 insert시켰고요.

데이터 로딩했을때와 안했을때로 자꾸 분기를 나누고있는데 이거는 여러분들 상황에 맞게가면 될 것.

저는지금 AB-BA로 저장되는 형태라서.

두개를 교차시켜서 양쪽에다 넣는 구조이다. AB-BA를넣고 인덱스가 없는 구조이기 때문에.

근데 지금 얘 같은 경우는 데이터를 저장할 때 AB-BA가 그대로 저장되있다 보니까 로딩할때는 저런과정없이 들어온 그대로 셋팅할려고 따로 분리시켜 놓은 것이다.

이거자체에 대한 논의거리 딱히없죠?네

**재연결 테스트**

아쉽게도 이 예제에서는 재연결에 대해서는 만들어 놓지 않았다. 별로 큰 특이점은 없다.

여유가 된다면 해볼 것.

**스트레스 테스트와 더미,봇테스트**

**스트레스 테스트는 단순히 부담을 주는것이며 더미테스트, 봇테스트라고 한다면 컨텐츠까지 들어가는 상황이다.**

**더미, 봇테스트의 목적은 실제 유저들이 몇 명이 들어올수있는지 가상으로 테스트하는것이고,**

**스트레스 테스트는 그냥 때려부어서 처리속도나 간단한 네트워크 상에서 버그를 테스트한다.** (두가지를 따로 분류 하지않는 경우도 있는데 여기에서는 나눠서 본다.)

스트레스 프로그램이나 더미프로그램은 버그가 0이어야하기 때문에 개발이 오래걸릴 것.

**테스트가 되려면 왠만큼 인프라가 받쳐줘야 한다.**

(더미테스트의 경우 5000명을 한다고 했을 때 한쓰레드당 50명씩 100개의 쓰레드를 만드는 구조이기 때문에, 버벅일 수밖에 구조이다.

또한 한대에다가 돌린다면 해당 컴퓨터외에도 이더넷도 부담될것이고, 스위치가 뻗을 수도있다.

예전에는 더미키면 L2가 죽어서 마비가 되는 상황도 있었음.)

**로컬에서 테스트하는 것은 아무런 의미가 없다.**

로컬에서 테스트할때랑 네트워크테스트할때랑 다르며, 네트워크 테스트시에 어느정도 장치를 거쳐서 테스트하는지에 따라서도 완전히 달라질 것이다.

**어쩄든 우리의 상황은 한프로세스에서 몇천명이상이 커버가 가능하게 해야한다.**

((

테스트할때, 프라우드넷 서버엔진으로 서버를 만들었다면 더미도 프라우드넷 클라로 가야 통신이 될것이다. 프라우드넷 클라이언트는 CPU사용률이 엄청나게 높은데다가, 한대밖에 제공되지않기 때문에 결국 하나의 프로세스에 프라우드넷 라이브러리 Instance를 100개를 만들어야 하는 환경이 되는데 이를 컴퓨터가 버텨내지를 못한다.

))

**스트레스 테스트라면 특정 메시지를 대량 전송하는 형태로 만든다.**

**우리는 에코로 메시지를 만드는 것.**

**테스트하면서 관련된 정보와 로그를 확인한다.**

**서버와 서버를 테스트하기위한 더미클라를 만들 때**

**이둘을 동시에 진행할경우 문제를 찾기 매우 어려워지므로, 어느정도 검증된 서버나 더미를 통해 나머지를 테스트해나간다.**

때문에 일단 제공된 더미로 테스트를 진행하고, 이후 어느정도 서버가 테스트됐다 싶으면 서버를 기반으로 스트레스 테스트 더미를 만들어 본다.(재연결까지 권장. 재연결은 단순히 랜덤하게 연결/끊기 를 반복한다.)

**Disconnect에 대한 기준**

이번 예제 이후에 제공되는 더미에는 재연결기능이 들어가 있으나 해당예제에는 넣지않음.

현황 : Cononect 시도함수, Connect 실패함수, 또는 Disconnect.

애초에 지금 더미 스트레스 프로그램에 끊는 코드가 없으나, 끊고 재연결하는 코드를 만들었다고 가정할경우 서버입장에서 끊어진 것을 감지하여 해당 세션을 정리하고, 새로들어온 세션을 테스트하고 정리하는 작업이 반복될 것.

**이경우 서버측에서 세션이 문제가 생겨서 끊어졌는지, 알아서 끊었는지 판단할 기준이 없다.**

**이는 더미측에서 검증을 하는것으로, 더미가 스스로 끊지않았는데 임의로 끊은 것을 카운팅 해주는 형태로 가야한다.**

그걸 카운팅하면 그걸 한 개라도 나온다면 잘못된 애를 끊는 것이다.

**패킷 송수신에 대한 레이턴시**

더미에서 패킷이 서버를 찍고 오는 시간을 체크하는 형태로 레이턴시를 측정한다.

**우리는 기본적으로 Select모델이기 때문에 레이턴시가 길게 나올 것이다. 대강 3,40m/s정도 나올것인데, 로컬에서하면 프레임으로 돌기 때문에 어쩔수없는 상황이다.**

**이후 IOCP로 만들면 네트워크 분산서버를 붙여도 제대로 만든다면 2~3m/s선이 되며, 이것 이하로도 나올 수 있다.**

예전에도 언급했듯 국내라면 인터넷상의 레이턴시는 거의 존재하지않으므로, (국내에서는 보통 핑에대한 레이턴시는 4m/s, 5m/s수준.) **3,40수준의 레이턴시가 나온다면 이것은 우리가만든 서버/클라의 문제일 것.**

**TPS 측정**

**초당 몇만건이 나오기 때문에 평균치를 보게되는데, 이때 최대값과 최소값을 항상 보관한다.** (최대값과 최소값이 평균과 오차가 심할경우 평균이 매우 달라지기 때문에, 잘못 판단할 수 있다.)

**TPS(Transaction Per Second)는 초당 처리건수를 말한다. 이때 Packet이라는 용어는 L3에서 사용하는 용어이므로, L3장비는 PPS라는 표현을 사용한다.**

일반적으로 L3장비를 본다면 보통 PPS는 60MPPS정도쯤(초당 6천만건) 나오게된다.

L7계층의 용어는 Message이므로 사실 MPS라고 표현해야 맞겠지만 이러한 표현은 잘 쓰지않는다. (TPS또는 PPS라고표현)

**또한 우리는 패킷 송수신 전송량도 뽑아내야 한다.**

**회선은 10Mb, 100Mb, 1Gb등이 존재하는데 이 계약은 월단위로 몇천만원 단위이므로 반드시 전송량을 바탕으로 보고한다.**

**이때 프레임은 장비까지 가는것이기 때문에 상관없으나, IP헤더와 TCP헤더는 무조건 붙여야 한다.**

**이러한 전송량을 계산할때는 Send를 Count하여 \*40하는 방식은 애매한데, Send를 호출한다고 실제로 Send를 하는것이 아니기 때문**이다. (최소한 이수치보다 적게나올것)

하지만 개발자 입장에서 이를 알수있는 방법은 없으나, 최대치를 잡아야 하는 입장으로서는 최선의 선택이 될것.

물론 이와같은 상황도 구체적으로 표기가 되어야 할것이다.

(ex SendCount로 계산했을때 n가 나오고, 이는 최대치예상으로...etc)

**하지만 위와같은 상황은 모두 코드상에서 말한 최대치를 얘기하는것이고, 확실하게 하려면 이더넷 차원에서 수치를 뽑아내야한다.**

이후 모니터링 하는 파트에서 이더넷자체를 모니터링 하는 방식으로 간다.

단, 이더넷차원에서 뽑아낸 수치는 최대치가 아니기 때문에 실제로 더 높은 수치가 나올 수도 있다.

이러한 사항들은 라이브 서비스 들어가기전에 필수적으로 체크해야할 항목들

**특정 TPS의 최대치가 비정상적으로 높을때**

간헐적으로 핑이튈때는 대대적인 원인파악을 해야할 것인데 보통 로컬이라면 그럴일이 잘 없다.

네트워크 인프라상에서 테스트할 때 우리코드가 아닌 외부(L2, L3)에서 영향을 받을때가 있다.

이때는 먼저 로직상에서 먼저 체크한다음(Recv받아서 처리하고 Send하는것 까지 프로파일링) 클라-서버 양쪽다 문제가 없다면 장비를 하나씩 체크한다.

((

마치 데스크탑에 문제가 생겼을때 센터에서 하는 방법과 비슷하다.

부품들을 하나씩 바꿔보면서 (파워, 메모리, 비디오카드...etc) 정상적으로 진행되는지 체크한다. 이후 불량인 부품이 나오게되면 또 세부적으로 체크를 할 것.

))

**코드설명**

제일 후반부에서 스트레스테스트 하는 코드를 보면서 설명한다.

**Wait echo count**

몇 명의 에코가 기다리고있는지 표기.

에코가 다녀와야 다음에코가 가는것이므로 기다리는 수치이다.

0이 나올수록 좋은 것.

**MAX Latency, 연결시도, 연결성공, 연결실패...**

**연결 실패 : 백로그큐가 다 차서 fail하는 경우를 뜻함**

**사실 위와같은 연결 실패는 클라입장에서는 서버가 좀 느린 정도에 체감하고 그치므로, 사실 큰 의미는 없다.**

Accept하는 초당 TPS가 초당 2천건이 나온다면.. (1초에 2천명이상이 들어올 일이 없음)

Connect Fail을 카운팅하긴하지만 문제삼지는 않는다.

**단,위와같은 항목들은 모두 더미쪽에서 체크하는것이지 서버쪽에서는 불가능하다.**

**DisConnect From Server**

서버가 임의로 나를 끊은 것으로, 이는 매우 중요하다.

**클라쪽에서 내가 끊은것과 (내가 끊지않았는데)끊긴 것을 계속 체크할 것이다.**

Recv, Send에서 에러가 났다고하여 무조건 끊는것도 아니고, 내가먼저 끊기를 시도했다면 에러가 나도 상관이 없을 것이다.

**TimeOut처리**

TimeOut처리 역시 필수적으로 해야 할것.

**FPS의 봇클라이언트와 MMORPG의 봇클라이언트**

**MMORPG에서 플레이어와 같은 봇만들기는 불가능에 가깝다.**

오토로 퀘스트를 진행하거나, 단순히 전투만 한다거나 하는 등의 간단한 봇은 만들 수 있겠지만 필드의 몹을잡고, 거래를하고 아이템 조합을 하는등의 다양한 행위를 할수 있기때문.

**FPS일 경우에는 봇을 플레이어와 거의 흡사하게 만들 수 있다.**

((

지금은 섭종한 듀랑고 같은 경우도 오픈전에 더미/봇으로 테스트를 완료했지만 모닥불컨텐츠에서 문제가 터졌다.

모닥불을 피우면 주변 클라가 힐링되는 컨텐츠가 있는데 그 모닥불을 테스트하지 못한 것이다. 구체적으로는 유저들이 모닥불을 많이 사용하다 보니 내부 네트워크 자체에서 마비가 온것으로,

셀단위로 서버 분산처리를 해놨기 때문에 내부 네트워크 자체에서 마비가 온것이다.

결국은 당장 해결할 수 없는 부분이어서 긴급점검으로 모닥불컨텐츠를 없애버렸다.

))

**더미/봇클라이언트를 얼마나 잘 만드느냐가 굉장히 중요하기 때문에 차근차근 만들어 두는것을 적극 권장한다.**

**MMORPG의 더미봇**

MMORPG에서의 더미봇은 아직까지 뚜렷한 해결책이 없다.

**라이브서비스가 되는 상황에서, 새롭게 들어가는 컨텐츠보다는 새롭게 들어가는 컨텐츠로 인해 다른곳에서 문제가 터지는 경우가 대다수이다.**

때문에 이를먼저 검증해야 하는데 (결국은 메모리 포인터같은 곳에서 문제가 터지게 되있음) 플레이어와 같은 더미를 만들지 못하므로 테스트서버를 운영하면서 검증을 거치게된다.

실제 라이브서비스하는것에 대한 메시지를 매크로처럼 받아서 해당 라이브서비스를 그대로 재현할 수 있다면 환상적일 것이지만 아직까지는 방법을 찾지 못함.

(구체적으로는, 어느시점 부터 DB를 풀백업받고 캡쳐, 클라들이 쏘는 메시지들을 그대로 다 받아서 그대로 리플레이 하듯이 재현을 하면 그것을 3~4일치라도 받아 놓고 돌려서 테스트하는 형태.)

**Replay**

라이브서비스를 재현하는것을 구상하기위해서는 먼저 Replay의 원리부터 살펴봐야 한다.

**리플레이의 경우 모든 프레임이 카운팅이 되야하고 몇번재 프레임에 키가 들어갔다가 그대로 똑같이 아귀가 맞아 떨어져야한다.** 이거자체도 쉽지않다.

오버워치는 중간중간 키프레임에만 도달시키는 키프레임 방식을 사용했다.

때문에 실제 플레이와 리플레이가 완벽하게 똑같지 않고 애니메이션이 약간씩다른 형태로 나타난다.

예전에 언급했듯이, 콘솔쪽(액박,플스)에서는 로직자체를 무조건 프레임 고정으로 간다.

지금시대에서도 전체 프레임을 동기화시켜서 멀티스레딩에서 A로직에서 렌더프레임찍고 다음코드로 넘어가는 형태를 취한다.

이와같이 프레임이 동기화되는 콘솔환경에서는 리플레이 시스템이 가능해질것.

하지만 실제 서버의 경우 캡쳐된 내용들을 그대로 넣는 동안 프레임을 서버쪽에 완전히 똑같이 재현을 시켜야 하는데 아직까지는 구상단계. 너무 난해하다.

# 스트레스 테스트 프로그램 ?  
  
- 서버에 대량의 접속 및 메시지(패킷)을 보내서 처리속도 및 버그를 확인하는 프로그램  
  
  
  
  
  
# 필수기능  
  
**- 100명 ~ 수천명의 접속 가능 (5000명이 적절)**  
- 특정 메시지(패킷)의 대량 전송  
  
- 관련 정보 로그 및 확인  
  
  
  
  
# 모니터링 정보  
  
- 접속 수 (Connect 시도 수, Connect 실패 수, 연결끊김 수 )  
  
접속 상태는 중간에 끊길 수 있으므로 수시로 반영 되어야 함  
  
  
- 패킷 송신 -> 수신의 레이턴시 (평균, 최대(옵션), 최소(옵션), 초당 최대(로그저장))  
  
- 패킷 송수신 횟수 (초당)  
  
- 패킷 송수신 전송량 바이트 (초당)

==============================