**LanClient**

LanServer <-> LanClient

Lan통신망안에서 서버끼리의 통신.

**CLanClient**

- bool Connect 바인딩 IP, 서버IP / 워커스레드 수 / Nagle옵션

- bool Disconnect()

- bool SendPacket(Packet \*)

virtual void OnEnterJoinServer() = 0; < 서버와의 연결 성공 후

virtual void OnLeaveServer() = 0; < 서버와의 연결이 끊어졌을 때

virtual void OnRecv(Packet \*) = 0; < 하나의 패킷 수신 완료 후

virtual void OnSend(int sendsize) = 0; < 패킷 송신 완료 후

// virtual void OnWorkerThreadBegin() = 0;

// virtual void OnWorkerThreadEnd() = 0;

virtual void OnError(int errorcode, wchar \*) = 0;

**LanClient는 완성된 LanServer를 가져다 그대로 만든다.**

**대신 Client이므로 Session이 하나만 있으면 될것이다. (SessionID의 개념도 없을 것)**

**Connect의 block**

Client에서만 호출되는 Connect함수의 경우는 지금까지는 block함수인 Connect()를 사용했다.

ConnectEx를 써서 OverlappedIO로 비동기 구현이 가능하나, 이렇게까지는 하지않음

**우리는 Nonblock소켓으로 제어한다**

**block으로 가도 문제는 없지만 서버 어플리케이션 자체가 멈춰서는 안됨(Lan은 서버간통신)**

**따라서 Non-block소켓으로 진행하고,(항시 실패) Select를 통해 연결 성공여부를 판단한다.**

어차피 Select를 쓰고있으므로 Nonblock소켓 Connect를 왜쓰는지 의아할수있는데,

**(어차피 Nonblock소켓을 쓰고있는데 왜 Select까지 쓰냐? 가 문맥적으로 맞지않나)**

**Nonblock소켓에서의 Connect는 time-out제어가 전혀 되지않기 때문이다.**

**우리는 Connect를 비동기로 가자는 것이 아니라 block의 시간을 Select로 제어하고자 하는 것.**

(300m/s정도면 충분하며, 이 시간 안에 판정이 떨어지지않는 경우 끊긴것으로 봐도 무방하다.)

**결국 Select의 TimeOut을 걸고 Nonblock소켓으로 Connect하여 진행하게 된다.**

성공/실패 유무를 빠르게 판단하여 Connect함수 안에서 빠지는 형태.

LanClient가 잘 돌아가는지는 우리가 만든 LanServer를 통해 검증해야 한다.

만약 LanServer가 문제가 있는 경우 문제파악이 굉장히 힘들어 질 것.

에코서버이고 Session이 하나밖에 존재하지 않기 때문에 동기화이슈가 날만한 곳도 없다.

**문제는 연결과 재연결이 반복될 때 나타나게 될것이다. (더미클라의 Reconnect항목과 같음)**

클라 개수도 큰 문제가 없을 것이며, 하나만 가지고 하더라도 Connect/DisConnect.

**LanServer와 NetServer**

**LanServer와 NetServer는 거의 모든코드가 동일한 형태이다.**

단순히 복사해서 NetServer로 이름만 바꾸면 NetServer가 된다.

**단, 프로토콜 암호화가 들어가야 하기 때문에 프로토콜 헤더쪽이 달라질 것이다.**

**그리고 세션의용량(개수) 가 훨씬 많아질 것이다.**

**SessionIndex로 ID검색하기**

**<SessionID검색 한방에 하기. 생략>**

**NetServer**

NetServer용 자료.

**이에 맞게 인코딩/디코딩용 함수를 구현하면 된다.**

**그래서 고정키 하나와 랜덤키 하나를 가집니다.**

**- 1 byte 고정 Key**  (상수값)

**- 1 byte 랜덤 Key**  (직렬화버퍼가 만들어질때마다 랜덤값 생성)

**키는 2 개를 사용한다.**

**고정키는 상호간에 정해진 상수값 상태이며, 랜덤키는 패킷 헤더에 첨부된다.**

**RandKey**

**Code(1byte) - Len(2byte) - [RandKey(1byte)] - CheckSum(1byte) - Payload(Len byte)**

**(Total : 5byte)**

**RandKey의 용도는 매번 다른 형태가 되게끔 원하기 때문이다.**

암호화를 XOR로 진행하고 (대부분의 암호화는 XOR로 진행된다)

**비트단위가 아닌 byte단위로 나눠서 헤더를 사용한다면, 프로토콜 자체 패턴이 노출되게 된다.**

(LanServer(서버통신)는 지금 만들었던 길이 2byte짜리로 그대로 가게될것이며, NetServer(클라통신)은 이 프로토콜을 기반으로 가게된다.)

**@ 암호화 대상은 CheckSum + Payload 입니다. Len 과 RandKey 는 암호화 하지 않고 그대로 노출 합니다.**

**RandKey를 암호화대상에 넣지 않은 이유는?**

**XOR의 특징 - 교환법칙**

Data ^ A ^ B ^ C = RES

**Data를 위와같이 암호화했다고 한다면, 이를 복호화하기 위해서는 ABC를 XOR한 하나의값만 찾아내면된다.**

따라서 3번이나 XOR한 것이 의미없는 행위가 되는 것.

K = 고정키

R = 랜덤키

**PayLoad모양**

**RES = PayLoad ^ K ^ R**

위와 같은 형태로 했다면 푸는사람 입장에서는, XOR한 순서대로 풀 이유가 없다.

**총 패킷모양**

**Code(1byte) - Len(2byte) - [RandKey(1byte)] - CheckSum(1byte) - Payload(Len byte)**

현재 RandKey는 RandKey ^ K(고정키)한 값이다.

R ^ K.

PayLoad는 PayLoad ^ K ^ R.

**그렇다면 헤더에있는 RandKey를 PayLoad로 XOR한다면 이는 한번에 복호화가 성공하게된다.**

헤더에있는걸 그냥 띵걸면 풀리게된다.

**XOR은 어설프게 하는경우 안하느니만 못하기 때문에 주의해야한다.**

**@ CheckSum**

**(모든 데이터는 unsigned char 로 처리 합니다.)**

**Payload 부분을 1byte 씩 모두 더해서 % 256 한 unsigned char 값**

**결론적으로 체크섬은 PayLoad를 대상으로하고, 체크섬과 PayLoad를 암호화한다.**

**체크섬의 용도**

**체크섬이 없고 암호화만 진행된 데이터인 경우, 복호화 하였을 때 이것이 내가보낸 데이터인지 위변조된 데이터인지 알 방법이 없다.**

**따라서 체크섬을 통해 우리가 보낸 데이터가 맞는지 확인하는 것.**

**체크섬-암호화를 진행하는 목적**

우리는 메시지를 매우 복잡하게 하여 아무도 뚫을수 없게 만들고자 하는 것이 아니라, 최대한 가볍게 데이터 패턴을 감추고자 하는것이다.

**패킷 암호화 조건**

**1. 키의 용량이 적어야 한다. (속도저하, 가벼운 암호화 목적)**

**2. 패킷 생성시 마다 키가 변경되게 하고 싶다 - 같은 패킷이라도 항상 다른 결과가 나오도록**

**3. 데이터에 패 턴이 있더라도 암호화 결과에는 패턴이 나타나지 않아야 한다.**

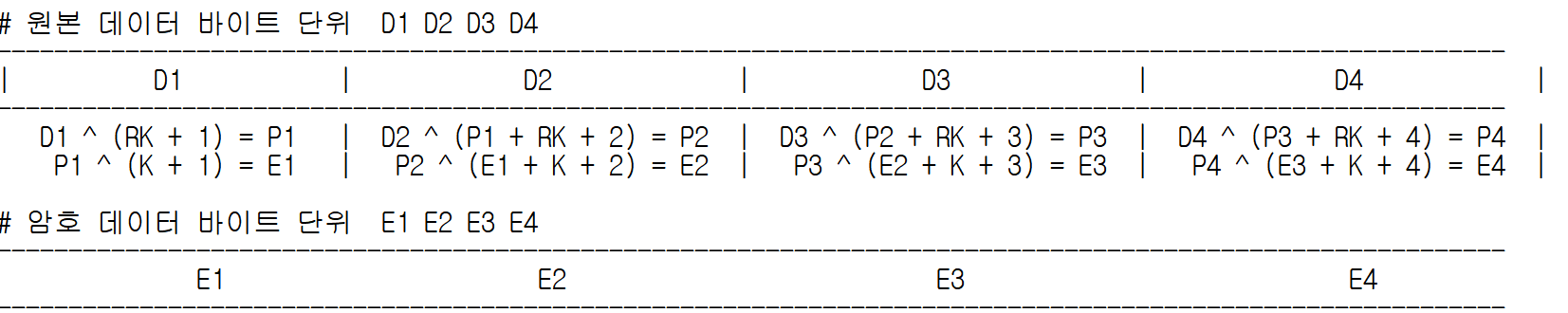
**4. 키를 잘못 입력하면 잘못된 결과가 나와야 한다.**

위 네개를 기준으로 본다.

상호 고정키 1Byte - K (클라 - 서버 쌍방의 상수값)

공개 랜덤키 1Byte - RK (클라 - 서버 송수신 패킷 헤더에 포함)

# 원본 데이터 byte 단위 D1 D2 D3 D4



1. 키의 용량이 적어야한다 - 무작위 대입법(브루트포스) 에 약하지만 이는 무시 함

2. 패킷 생성시마다 랜덤키 1개를 생성하여 이를 그대로 패킷에 포함하여 전달 할 것임. 이는 패킷마다 모양을 다르게 하기 위함

3. 0 0 0 0 0 같은 데이터가 반복 되더라도 이 패턴이 나타나지 않도록 하기 위해 byte 단위로 +2,+3,+4 를 해주고 있음

4. 각 위치마다의 ^ 대상에 RKey, Key 를 반영하여 잘못된 키가 들어올때 잘못된 결과를 유도함.

-- sample data --

위 방법대로 암호화된 샘플 데이터, 이와 같은 결과가 나오는지 그리고 복호화가 잘 되는지 확인 합니다.

사이즈 : 55byte

데이터 (텍스트) : aaaaaaaaaabbbbbbbbbbcccccccccc1234567890abcdefghijklmn (널문자포함 55byte)

데이터 (16진수) : 61 61 61 61 61 61 61 61 61 61 62 62 62 62 62 62 62 62 62 62 63 63 63 63 63 63 63 63 63 63 31 32 33 34 35 36 37 38 39 30 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 00

고정키 : 0xa9

랜덤키 : 0x31

암호화 (16진수) : f9 43 95 8c 5f f3 f7 44 b1 87 46 23 ad b5 1e 01 c1 a3 1e 3f b4 80 18 1b b2 ac 36 0b 8c 9c 4a 5e 84 84 7a 0e 74 84 72 0c 16 a8 82 68 c6 ac 72 74 86 20 32 50 86 04 2d

위와같이 인코딩/디코딩 함수를 만들어, 직렬화버퍼(CMsg 클래스)에 내장시킬 것이다.

일단은 LanServer테스트하시고, LanClient 적당히 테스트.

그다음 NetServer를 만들기위해 직렬화버퍼를 변경시킨다.

**앞 데이터를 기준으로 뒤에 위치한 데이터 암호화에 영향을 주는 방법은 실지 암호화에서도 많이 사용된다. (일반적으로는 블록을 훼손시키면서 넘어가는 방식)**

**이때 R키나 K키가 중간중간 들어가지않고 맨앞(1byte)데이터만 암호화한다음 이 데이터를 기준으로 뒤에 데이터를 암호화하게된다면,**

**키가 틀린경우 첫번째(1byte)데이터만 틀리고 나머지는 모두 풀리는 꼴이된다.**

**<9.16>**

**SendPacket - 헤더 세팅**

완성된 LanServer에 암호화기능을 넣고 NetServer로 바꾼다.

이때, SendPacket부분이 달라진다.

**SendPacket(SessionID, CMsg)**

(현재 SenPacket은 직렬화버퍼 포인터 자체를 던지고 있음.)

**이제 직렬화버퍼에는 NetServer만의 헤더가 생긴다.**

LanServer같은 경우 길이(2byte)만을 나타내는 헤더로 넣었다.

**이 헤더는 CLanServer클래스(데이터 송수신 담당부분)의 헤더로, ClanServer의 컨텐츠 쪽에서는 존재를 알지못한다.**

(에코서버, 에코더미에는 2byte를 제외하고 전달**.**)**(?길이를 포함한 2byte까지 전달해야지;)**

SendPacket한다면, SendPacket()내부에서 헤더를 세팅해야한다.

(이 헤더는 네트워크 라이브러리의 코드)

**따라서 컨텐츠쪽에서 헤더가 필요하다면 컨텐츠에서 넣어야 한다.**

프로토콜 구조에서 헤더에 가장 기본적으로 들어가는 것은 길이이다.

**네트워크 라이브러리 입장**

**--------------------------------------------------------------------------------**

**Header : [ LEN(2byte) + TYPE(2byte) ]**

**PayLoad : [ Data ]**

**--------------------------------------------------------------------------------**

이때 컨텐츠헤더는 LanServer입장에서는 PayLoad이다.

**헤더 세팅**

**NetServer라면, 지난번에 설명한 암호화헤더(5byte)가 추가적으로 들어가야 한다.**

**따라서 SendPacket안쪽에서 이 작업이 이뤄져야한다. (인코딩 + 헤더세팅)**

원장님 질문) 지금 직렬화버퍼에 헤더공간을 확보했는가? X

이제 직렬화버퍼 앞공간 5byte를 떼놓을 것이다.

**이제 도입될 직렬화버퍼 방식은 우리 네트워크라이브러리용을 위한 개선방식이다.**

(직렬화버퍼를 계속 범용적으로 사용하고싶다면, 현재 직렬화버퍼를 변경하지않고 상속받아 사용하면 된다. 하지만 우리는 네트워크 라이브러리를 개발하는 입장으로서 굳이 상속할 이유는 전혀없음)

**이제 데이터를 Enqueue/Dequeue하는것은 앞 5byte헤더공간을 제외한 직렬화버퍼 내부 버퍼로, 헤더 5byte는 private으로 막는다.**

**헤더공간은 LanServer, NetServer안쪽에서만 저 헤더공간을 접근해서 세팅이 가능하게끔한다.**

**Setheader()**

이제 헤더를 건드리는 공간의 함수가 나와야한다.

**이때 LanServer의 헤더는 2byte이고, NetServer의 헤더는 5byte이다.**

만약 LanServer의 헤더가 들어가야하는경우 5byte중 뒤 2byte를 사용해야 한다.

헤더부터 하나의 메시지가 되게끔 하기위함이다.

**따라서 Setheader함수역시 2byte버전, 5byte버전 두개로 나뉘게된다.**

**실제 이 메시지 전체에 대한 포인터를 얻는 것 역시 두개로 나뉘게된다.**

|  |  |
| --- | --- |
| Header(5byte) | PayLoad |

(이제 실제 Send시에는 헤더를 제외한 PayLoad버퍼부터 들어가게된다.)

해당 방식이 마음에 들지 않을 경우

직렬화버퍼 내부에 몇byte헤더를 쓰는지 셋팅해놓고 내부에서 자동으로 분기를 타는 방식도 가능하지만 이렇게까지 할 필요는 없다.

결국 멤버변수를 두개 (실제 버퍼의 시작포인터, public접근 가능한 버퍼 포인터)를 두고 이 둘을 활용해서만 사용한다.

그리고 시작점을 얻어주는 GetPtr…()등의 함수가 나오게 될것이다. 당연히 위와 같은 함수들은 private으로 잠궈야할 것.

**중간정리**

**헤더길이(Len)만이 들어간 LanServer에서는 인코딩 개념이 들어가지도 않아도 상관없다.**

NetServer쪽에서는 체크섬과 암호화의 개념이 들어간다.

**인코딩은 헤더세팅 + 암호화 + 체크섬확보**

**디코딩은 호출하여 복호화한뒤 체크섬확인**

이때문제는, 패킷포인터 하나로 여러세션에게 보낼 때 코드상에서 여러 번 인코딩하는 꼴이된다.

지금 에코에서는 하나의 직렬화버퍼 패킷을 한세션한테만 보내지만 실제 게임에서는 아니다.

**따라서 인코딩은 최초 한번만 호출이 되게끔 Flag를 둬야되는데..**

**이는 인코드 함수 자체만보면 멀티스레드 환경에서 안전하지않다.**

이는 싱글톤에서도 항상 문제가 되는 부분이다.

((

**싱글톤을 만들 때 언급한 세가지 방식**

**1. static멤버를 인스턴스로 둔다. (문제 X)**

**2. static멤버를 포인터변수로 둔다. (문제 O)**

**3. GetInstance()를 지역 static함수로 둔다. (문제 O)**

if(pointer == nullptr)

{  
 pointer = new Obj;

}

위와 같은 싱글톤 함수를 사용한다고 했을 때, A스레드가 new를 진행중일 때 B스레드가 같은 분기를 탄다고한다면 2번생성되는 꼴이된다.

따라서 1번이 아닌 2,3의 상황에서 해결책은?

**명시적으로 초기화시점을 수동으로 두는 것이 가장 익숙한 방법.**

**(멀티스레드 환경에서 동시 호출시 문제가 생기는 것이기 때문에 프로세스가 생성되는 main문에서 명시적으로 생성되도록 호출해 주는 방법.)**

**CallOnce**

C++11 에서는 나온 문법으로, 한번만 호출되는 시점에서 CallOnce를 호출하도록 한다.

C++에서는 싱글톤을 사용할 때 CallOnce를 권장한다.

CallOnce는 내부적으로 Mutex가 들어간다.(Mutex를 인클루드해야한다)

결국 기존의 개념을 랩핑한 것.

**Flag를 두고 접근을 한번만 하도록 제한하는 방식**

그렇다면 flag를 하나두고 flag의 여부를 가지고 최초에 한번 수행이 되게끔 한다면?

이때 문제는,

**A스레드가 TRUE로 바꾸고 객체를 가지고 나가려는데,**

**B스레드가 들어와서 flag를 확인하고 TRUE라면? B스레드입장에서는 할 수 있는이게 없다.**

**결국은 누군가가 들어가서 진입해서 뭔가 하는 행동을 시작한다고한다면 해당 작업이 끝나기전까지 블락걸려야한다. 다른 방법이 없음.**

결국 이 방식으로는 해결할 수없음.

))

**다시 인코딩 함수**

결국 하나의 패킷을 여러 세션한테 보낼 때, flag를 두고 한번만 인코딩을 하겠다고한다면 이는 멀티스레드에서 안전하지않은 코드가 된다.

결국 Lock을 사용해야하고, C++11의 CallOnce를 사용해야 한다고 하더라도 내부에서 SRWLock인 SRWLockExclusive가 사용된다.

락을 거는행위자체가 문제가 되는 것이 아니다.

경합이 없다면 단순히 인터락비트체크를 하고 빠질것이지만, 경합이 있는 경우에 SpinLock이 돌고 운이없다면 Block까지 될수있는 상황.

그렇다면 이 부분을 어떻게 해결해야 할 것인가?

**직렬화버퍼의 thread - Safe?**

멀티스레드에서 하나의 직렬화버퍼를 대상으로 사용해서는 안된다.

애초에 우리는 직렬화버퍼를 thread-Safe하게 사용하는 것을 고려하지않았기때문.

**애초에 직렬화버퍼를 여러 스레드에 전달하여 동시에 SendPacket하는 설계가 매우 난해하기때문에 고려하지않는다.**

**(하나의 직렬화버퍼를 멀티스레드환경에서 동시에 사용한다는 것은 말이되지않음)**

결국 하나의 Msg를 여러 Session한테 보내는경우 반복문을 돌리면서 보낼것이다.

**직렬화버퍼의 인코딩 시점**

SendQ넣기 직전 SendPakcet()안에서 인코딩한다.

(SendQ에 넣고나서 인코딩을 한다면 꼬일수있음)

**SendPacket의 불필요한 호출?**

어차피 하나의 Msg가 만들어지고 한번의 SendPacket이 호출되어야 하는데, 반복문을 돌기떄문에 SendPakcet이 여러 번되는 부분이 불필요해 보일수있다.

**성능 위주로만 생각한다면 사용자측에서 인코딩까지하여 SendPacket()하는 것이 나을 수있음.**

**단, 인코딩이라는 과정 자체가 컨텐츠쪽에서 하도록 유도하는 것이 설계상으로 좋지않을 뿐더러, 우리는 네트워크 라이브러리를 만들고자 하는 목적(용이성, 편의성)이기 때문에 안쪽으로 넣는 방법을 택한다.**

((

**우리는 직렬화버퍼를 참조카운트 방식을 사용하여 여러군데서 재사용하게끔 만들었다.**

**이 부분도 여기 있어야 할 것**

))

**최종적인 형태**

이제 NetServer에서는 이부분이 바뀌게된다. 세션의 개수가 1~3만개가 될것이고,

SendPacket에는 인코딩함수가,

수신부쪽에서는 디코딩함수가 들어가야한다.

Recv완료통지가 왔을 때 메시지에 대한 우리의 헤더길이가 있다.

이 길이는 암호화 대상이 아니므로,

**1. 완성된 메시지인지 길이로 판단**

**2. 뽑아서 직렬화에 넣은다음 디코딩을 호출시켜 복호화한다**

**3. 복호화된 데이터가 우리것이 맞는지 체크섬을 통해 확인**

**4. 문제없는 경우 최종적으로 OnRecv로 넘긴다.**

이때 체크섬이 다르다거나, 복호화에 실패했다면 해당 세션은 로그를 남기고 끊어버린다.

**(근데 헤더가있어도상관은 없어요.. 있어도 접근할 방법이 없기 때문에.**

**PayLoad만 있어도 상관이없슴..)**

여기까지 완료가 됐다면 채팅서버를 만든다.

채팅서버에서는 하나의 메시지를 여러명한테 보내는 등의 상황이 나온다.

**랜서버 랜클라 <-> 넷서버 넷클라 차이**

**CMsg.h**

#include "time.h" // rand()를 구하기위한 인클루드

#pragma pack(push, 1)

struct NET\_HEADER

{

BYTE Code;

SHORT Len;

BYTE RandKey;

BYTE CheckSum;

};

#pragma pack(pop)

넷헤더 전용 구조체 추가.

**#define MSG\_CODE 119**

**#define MSG\_KEY 50**

사용할 1Byte코드와, 고정키를 define.

**#define HEADER\_SIZE**

2Byte에서, NET\_HEADER 구조체의 사이즈로 바뀐다.

**Setheader함수의 변경**

단순히 Len사이즈만 헤더로 셋팅했다면,

이제는 규칙에따라 NET\_HEADER를 채워넣어야 한다.

추가적으로 데이터를 보내는 순서는,

**[ PayLoad Copy -> Setheader -> Encoding ]**

**RecvComplete 변경**

완료통지 이후 호출되는 RecvComplete함수도 변경된다.

**먼저 NET\_SERVER 구조체로 길이와 1Byte 코드를 판단한다.**

**이후 뽑아낸 데이터를 Decoding하여 OnRecv로 전달.**

**SendPacket 변경**

**Setheader 이후 Encoding하는 코드가 추가된다.**

**로그인 서버**

우리가 만든 에코서버는 실제 로그인 서버에 적합하다.

**로그인서버는 다른세션과 연관성이 전혀없기 때문에 동기화가 없다.**

단순히 ID/PAS 체크후 끝.

따라서 워커스레드는 해당 세션만을 물고 처리해도 아무문제가 없음.

((

서버프로그래머는 서버를 자유자재로 만들어 낼수있어야 한다.

기획자가 필요하다고 하는데 DB에서 하지못한다면 새로운서버가 나와야한다.

웹개발자는 솔루션/오픈소스부터 찾지만, stateful개발자는 직접 만들어야한다.

))

Redis(메모리 DB. 메모리에 올려놓고 빠르게 결과보관, 검색해서 뱉어주는 것. 전원이 내려가면 저장되지않고 사라진다.), 물리저장이 되는 MySQL. 관계가 있는 데이터 베이스.

우리는 성능을 높이고자 캐시역할로 이러한 관계형 데이터베이스를 가져다 쓴다.

우리가 예전에 만든 친구관리 서버는 단순히 메모리에 올려놓고 메모리에 누군가 친구요청을 한다면 맵으로 관리하면서 만들었음.

**랭킹 서버**

유저데이터가 1억명인데 랭킹이 필요한다고 가정하자.

유저수가 어마어마하기 때문에 이는 DB에서 불가능하다.

따라서 단독서버가 필요하게 된다.

**랭킹서버는 유저 1억명을 모두 올려놓고, 단순히 랭킹만을 관리하게 될것이다.**

(Redis에 올려놓고 할 수도있겠지만, 레디스의 목적자체가 아닐뿐더러 매우 느림)

**게임서버는 랭킹순위가 필요할때마다 랭킹서버에 요청하고, 랭킹서버는 데이터를 뱉어준다.**

**랭킹서버 설계**

랭킹서버 입장에서는 우리의 게임서버가 클라이언트가 되는 것이다.

가장 단순한 방법은 하나의 자료구조에 모두 넣고 sort하는 방법.

랭킹은 계속해서 변환되므로 트리를 사용할수도 없다.

이 랭킹서버입장에서는 우리의 서버가 클라이언트가 되는것임.

게임서버(클라)가 랭킹서버(서버)에 요청할수 있는 것

**1. 랭킹 데이터 요청**

**2. 랭킹데이터 갱신**

어쨌든 자료구조를 가지고 관리를 해야한다.

게임서버에서는 수시로 스코어변경이 있을것이고, 새로운 데이터(사용자)가 추가될 것이다.

이러한 경우 어떤 설계가 적합할 것인가?

List로 들고 sort를 하는 것이 빠를지, Map으로 들고 삽입/삭제를 반복하는 것이 나을지는 직접 성능 테스트를 해봐야 할수있을 것.

**스레드 설계시 염두해야 할 것**

**스레드를 설계할 때 가장 처음 고려해야할 상황은 공유될 자원이 뭔지를 기준으로 판단해야한다.**

이미 앞전에 IOCP를 설명하면서 나올수있는 스레드 설계는 모두 언급되었다.

**1. Workerthread가 직접적으로 처리를 하는 것**

**2. 스레드를 따로 만들어두고 작업을 요청.**

**(동시처리가 되지않으므로 동기화문제 X, 성능은 기대하기 힘듬)**

**3. 기능별로 스레드를 분리**

**다시 랭킹서버**

워커스레드가 직접 처리한다는 구조로 본다면,

결국 스코어 변경과 유저추가가 될때마다 sort작업이 되어야 한다.

**어떤 구조든 단독접근만이 가능한 상황이므로, 정렬스레드를 따로 두는 것은 애매하다.**

**(만약 요청빈도가 많은 경우 WorkerThread에 직접접근인 경우 shared, 변경이 있을때는 Exclusive로 가는것도 생각해 볼 수 있겠다.)**

**Sorting은 꽤 긴 시간을 요하는 작업이고, Sorting하는 동안에는 어떠한 다른 작업도 할 수 없는 상황이다. (데이터(유저)추가, 데이터 읽어오기 등의 작업.)**

**1. sorting빈도 수를 낮추기위해 별도의 기준으로 sorting**

**이때 스코어 데이터가 완벽하게 실시간이 아니어도 된다면,**

**정렬스레드를 따로두고 싶다면 sorting자체에 대한 빈도수를 낮출수 있겠다.**

sorting자체에 대한 빈도수를 낮추기위해 실시간으로 sorting하지않고 별도로 기준을 세운다.

(ex 일정 시간마다 sorting하거나, n개이상이 쌓이는 경우 sorting하거나.)

이경우 정렬스레드가 따로 나올수 있게되었다.

**2. 스레드마다 사본을 활용하는 방식**

추가적으로 지금처럼 느슨한 동기화가 가능하다면,

**원본데이터가 아닌 사본을따로 만들어 활용하는 방법도 있겠다.**

멀티스레드 환경에서 분산시킬 때 스레드가 각자 자신만의 데이터를 가지고 있는다면 동기화 이슈를 없앨 수 있다.

(단, 데이터자체에 대한 동기화이슈는 남아있으므로 느슨한 동기화가 전제가 되어야 함)

**데이터 요청이 올경우 사본을 뱉어주고, 사본은 일정 주기마다 원본데이터로 업데이트된다.**

**결론**

결론적으로 어떤 방법이 더 좋다 나쁘다 단정지어 얘기할 수 없다.

sorting으로 Lock을 걸면서 접근을 제한시키는 시간과,

한쪽이 sorting될때마다 읽기전용 데이터로 카피하는 행위중

어떤 것이 오버헤드가 더 클것인지는 테스트를 통해 알아봐야 할 부분이기 때문이다.

(1억명인 경우 1명당 4byte라고 가정할경우 4억바이트로 4GB.이 데이터를 copy..)

또한 sorting같은 경우도 처음부터 끝까지 완전히 뒤집어 엎는게 아니기 때문에 오히려 오버헤드가 적을 수있음.

**테라의 논타겟팅**

느슨한 동기화가 전제가 될 때 사본데이터가 활용되는 예시로 테라의 논타겟팅이 있다.

**논타겟팅에서 충돌처리를 한다고 한다면, 공격 궤적에 따라 오브젝트를 모두 뒤져 충돌했는지 로직을 타야한다.**

이때 충돌 여부를 판단하는 작업에 락을 걸어버리면, 다른애들은 움직일수 없는 상황이된다.

**여러스레드에서 하나의 데이터를 대상으로 동시에 충돌처리하는 것은 말이 되지않기 때문에,**

**각자의 오브젝트 데이터를 따로 들고있고, 각각의 스레드는 자신의 데이터를 기준으로 충돌처리한다.**

이 경우 아무리 실시간으로 반영한다고해도 완벽하게 동기화가 맞을 수없게 된다.

**따라서 이 결과는 1차 필터링으로 사용되며, 최종결과물은 해당 후보군을 대상으로 판단한다.**

관점에따라 논타겟팅을 스레드가 타겟팅으로 바꾸고 있는 것.

**렌더 - 로직 파트분리**

멀티스레드 환경에서 가장 많이 사용하는 것이 로직스레드와 렌더스레드를 분리하는 것이다.

로직스레드에서는 오브젝트들의 좌표가 바뀌고, 렌더스레드는 좌표를 읽어야한다.

**이때 오브젝트에 락을 걸어버린다면 아무 의미없으므로, 할 수 있는 방법은 사본을 활용하는 것이다.**

로직을 돌고나면 렌더쪽에서 복사한다음 렌더한다.

단, 전제는 데이터를 카피하는 행위보다 렌더링이 훨씬 길다라는 전제가 깔려야 성능의 향상이 있을 것이다.

물론 렌더는 약간의 과거데이터가 될 것.

**채팅서버가 할일**

누군가 채팅을 보내면 본인주 9개 Sector에 메시지를 보낸다.

따라서 섹터의 개념이 들어가고, 플레이어들이 각 섹터에 들어있을 것이다.

**플레이어는 섹터간 이동할수 있고, 메시지를 보내는경우 9개의 섹터에 메시지를 뿌린다.**

**즉 섹터는 플레이어라는 데이터를 대상으로 변경, 삽입, 삭제, 순회의 로직이 필요한 상태.**

**채팅서버 설계**

**이때 워커스레드에서 모두 처리를 하되 섹터단위로 락을 걸고 간다면 데드락이 걸리게 된다.**

락이 많은 것 자체가 문제가 되지는 않는다.

(동기화 이슈가 발생하지않는다면 인터락변수를 한번체크하는것과 같기 때문)

A섹터 -> B섹터 이동시 A,B섹터가 락이걸리고 그전에 누군가 쓰고있었다면 중첩락..

이 경우 꼬일가능성이 거의 100%에 가깝다.

**그렇다면 맵단위로 락을 건다면?**

맵역시도 섹터간의 이동이 나올수밖에 없다. 역시 데드락의 위험성이 존재한다.

**단 우리입장에서는 하나의 맵만이 존재한다.**

**따라서 이렇게 락걸고 WorkerThread에서 치고들어올바에는 동기화 걱정이없는 스레드구조가 차라리 나을 수있음.**

**우리가 사용할 방식**

우리는 여러개의 WorkerThread가 하나의 updateThread로 Jop을 던지는 싱글스레드로 진행한다.

**필드가 하나이고, 동시에 처리하자고 한다면 동기화를 해결할 방법이 없기 때문.**

**이런 싱글스레드 구조여도 큐자체에 대한 동기화문제, 결국 Jop을 던지고, 해당Jop을 받아 처리하기 때문에 이 오버헤드가 생길수밖에 없다.**

**따라서 워커스레드 여러 개가 하나의 공유자원을 대상으로 Lock을걸고 각기 처리하는 것이 나을 수도있다.**

성능이 좋아서 전자를 택하는 것이 아니라, 다양한것을 연습해보기위함.

보통은 WorkerThread에서 직접 치고들어가는 방식이 가장 낫고, 이는 고전적인 방식이다.

(단, 이름있는 회사들은 모두 멀티스레드 방식을 택함)

대신 동기화를 얼마나 줄이는 것이 관건이다.

이제부터 직렬화버퍼/메모리풀 등 우리가 만든것에 대한 문제가 많이 나올 것이다.

**1.13**

**네트워크 라이브러리에서 전달되는 핸들러**

**OnClientJoin -> AcceptThread 내부에서 호출**

**OnRecv -> WorkerThread 내부에서 호출**

**OnClientLeave -> 어떤스레드에서든 호출이 가능하다.**

**OnClientLeave는 I/O Count가 0이될떄 ReleaseSession의 호출이 끝나고 호출된다.**

OnClientLeave처럼 어떤스레드에서 호출이 가능한 상태가 좋은 상황은 아님.

**현재 설계의 OnClientLeave에 내장된 문제**

(이후 바로 시작할 채팅서버는 심플하게 갈것이므로 이러한 문제가 나지않음)

**컨텐츠쪽의 스레드조차 여러개인경우, SendPacket이 여러번 동시에 호출될 수 있다.**

**따라서 동시에 여러개의 OnClientLeave가 호출될 수 있다는 말이된다.**

**이 경우 Session락이 재귀형태가 되므로 CriticalSection을 써야한다.**

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

**예시)**

컨텐츠 파트에서 Player를 다루는 자료구조가 있을것이고, 병렬처리를 위해 Player마다 동기화객체가 있을 것이다.

OnClientJoin으로 Player가 접속이 완료되었다.

Player를 Player자료구조에 삽입해야하므로 동기화를 건다.

**이때 삽입과 동시에 컨텐츠 코드가 들어가는 경우 (새로운 사용자접속시 해당 사용자 주변에 뭔가를 뿌려야 하는 등의 상황) 에는 SendPacket이 호출되어야 하며, 이말인 즉슨 도중에OnClientLeave까지 호출될수 있는 상황이된다.**

결국 재귀락이 걸릴수있다는 얘기.

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

OnClientJoin은 여기저기서 호출이 되므로 이러한 문제가 된다. 내부에서만 호출이된다고 한다면 덜할것인데, 네트워크 라이브러리 바깥범주인 유저쪽에서도 호출이 될수있다는 가정이 깔리기때문이다.

**스레드설계시 주의할점**

스레드 설계를 할때는 스레드의 방향성, 함수에대 한 스레드의 호출규칙 등을 명시하는것이 좋다.

위와같이 OnClientJoin이 어느곳에서나 호출할 수 있는 상황때문에 문제가 나는 상황은, 네트워크 라이브러리 내부에서만 호출이 된다고만 하더라도 훨씬 덜할 것이나 네트워크 라이브러리 바깥범주인 컨텐츠쪽에서도 호출이 될수있기 때문에 반드시 염두해야한다.

**문제 해결**

**이 네트워크 라이브러리를 사용할때 주의할점이라고 명시하여 CS를 사용하도록 해도 상관없음.**

현재 우리는 IOCount == 0 -> ReleaseSession -> OnClientJoin 순으로 호출된다.

따라서 Release를 네트워크 라이브러리 안쪽으로 넣어서 IOCount가 0이되는 순간이 다른쪽에서발생되지 않도록 하는 방법과,

0을 만드는 순간과 Release를 호출하는 순간을 분리하여 0을 만드는 순간 네트워크 라이브러리 쪽에서 어떻게든 알려주도록 하는 방법이 있을것.

가장 기본적으로 생각할수 있는 방법으로는 네트워크 라이브러리쪽에서 faling하는 것이 될수있겠다.

**네트워크 라이브러리 추가기능 - TimeOut**

네트워크 라이브러리내에 TimeOut기능은 일정시간 통신이 없는 세션을 알려주는 기능을 말한다.

**물론 TimeOut된 Session이 나왔다면 핸들링하여 컨텐츠쪽에서 끊는것을 결정하도록 한다.**

**TimeOut기능 구현**

수신이 있을때 시간을 저장하고, 다음번 Recv에서 해당시간을 차감하여 구한다?

이는 송신이 아예 없는 경우에는 식별할수가 없음.

결국은 타임아웃 스레드를 놓고 계속해서 세션들을 체크하는방법.

물론 체크하는대상을 억지로 WorekrThread로 억지로 할수있게끔 가능하기도하다.(처리를)

우리 네트워크 라이브러리 구조는 비동기(Overlapped I/O) 설계로, 반응이 있을때만 깨어나서 할일을 하도록 되어있다.

**따라서 반응이 없는 것들을 찾아내려면 별도의 스레드가 나오든 따로 순회를 하든 다른방법으로 찾아내야 한다.**

**채팅서버 전제**

우리 채팅서버는 MMO라는 전제로 간다.

실제 MMO라면 여러 Map이 나오겠지만 우리는 하나의 필드(Map)으로 간다.

**클라이언트는 게임서버와 채팅서버 각각에 접속해있는 상태이며, 게임서버와 채팅서버는 아무런 연관이없다.**

채팅은 결국 섹터의 개념이 나와야한다.

**채팅서버 섹터는 굳이 게임서버의 섹터와 같은 개념으로 갈 필요는 없으므로 좀 더 범위가 커도 상관없다. (실제로도 더 큰게 어색하지않음)**

**또는 채팅서버의 섹터를 새로 만드는것이 아니라 게임서버의 기존섹터를 n\*n형태로 늘려서 사용해도 상관없다.**

**채팅서버 문제?**

1-2-3 세개의 섹터가 연달아 있다고 가정하자.

이때 1섹터의 플레이어와 2섹터의 플레이어가 대화를 할때, 3섹터에 있는 사람은 2섹터에 있는 사람혼자 떠드는것으로 보일수있음.

이를 문제삼지 않는다.

**채팅서버의 설계 방식들**

**보통 필드에서 하는 채팅은 대부분 하지않기 때문에 채팅서버 자체를 분리하는 개념이 점점 없어지는 추세이다.**

**귓속말 / 길드채팅 / 파티채팅 등은 컨텐츠를 알아야 하므로 게임서버 쪽으로 빠진다.**

**더군다나 게임서버1 – 채팅서버1 로 간다면 그만큼 채팅서버가 늘어나고, 물리서버가 많아지는 경우 관리도 힘들어진다.**

**따라서 대부분은 통합된 하나의 채팅서버를 두도록 설계한다.**

(모바일게임, PC MMORPG등 모두 포함하는 얘기)

**그래도 우리는 (게임서버1 – 채팅서버1) 가 세트가 이뤄지도록 구현한다.**

이 경우플레이어는 게임서버와 채팅서버에 접속하여 게임을 플레이 하는것.

**이같은 설계는 채팅서버가 할일이 거의 없다.**

(이같은 채팅까지 채팅서버에 넣어야 한다면 매우복잡해짐)

동접이 5만명씩 나온다고 하더라도 한대의 채팅서버가 여러개의 게임서버(서버군마다 나옴)를 커버하는 설계로 간다. (더 많아지는 경우 세트로 갈것..)

**게임서버 - 채팅서버 관계 설계**

채팅서버에서도 결국에 하나의 플레이어 객체로 관리를 해야한다.

하지만 플레이어의 섹터좌표만 알면 된다.(월드좌표 X,Y를 모두 알 필요는 없음.)

**클라이언트에서는 직접 본인섹터를 계산하고, 섹터간 변화가 있을때 채팅서버에 섹터좌표를 보낼 것이다.**

(사실 클라는 섹터크기와 섹터좌표를 전혀 몰라도 상관없다.)

이같은 경우 완벽하게 실시간 동기화가 되지않고, 클라쪽 데이터를 가지고 로직을 타기때문에 조작이 가능해진다.

(실시간 동기화는 별다른 방법이 없고, 채팅은 조작해도 아무의미가 없으므로 이대로 진행.)

이같은 설계가 마음에 들지않는다면 결국 게임서버가 끼어들어 채팅서버와 데이터를 주고받아야 하고, 이같은 설계는 차라리 채팅서버를 없애고 게임서버로 통합시켜 버리는것이 나음.

(채팅이란 결국 플레이어들에게 돌려주는 형태이기 때문)

**서버 설계 추세**

**요즘 CPU코어는 너무많기 때문에 서버분산을 하지않고, 하나의 프로세스안에서 멀티코어를 잘 활용하자는 추세이다.**

**서버를 분산하게되면 개발/관리가 더 힘들어지고, 버그도 많이나올뿐더러 동기화를 맞추고, 여러과정속에서 오버헤드가 더 많이 발생되기 떄문.**

우리는 다양한 서버를 개발하기위해 일단 게임서버 -채팅서버 세트로 가는 설계로 간다.

**채팅서버 구현**

먼저 섹터가 구현되어야한다.

**섹터는 TCP FIghter랑 똑같이 구현한다.(2차원 배열)**

**섹터는 플레이어를 담아야 하므로 섹터배열의 인자는 STL의 자료구조가 될것이며,**

**섹터 내부에는 플레이어포인터, 또는 세션아이디가 들어갈 것.**

(채팅서버에서는 섹터배열을 제외하고는 어떠한 배열도 사용X)

**그리고 플레이어들을 관리할 컨테이너는 정렬이 필요없으므로 unordered\_map으로 간다.**

(키를 회원 고유번호인 ACountNo로 잡든, SessionID로 잡든 알아서 결정.)

**채팅서버를 분리하지않는 경우 (다른 설계)**

게임서버 – 채팅서버를 별도로 분리하지않고 길드채팅 / 파티채팅과 같이 게임서버쪽으로 포함시키는 경우를 말한다.

채팅서버를 따로 분리하지않고, NetServer에서 다 처리한다면

**OnClientJoin쪽에서 플레이어를 생성.**

**OnRecv에서는 PlayerMap에서 플레이어를찾아 플레이어 주변에 채팅을 보낸다.**

**Leave에서는 PlayerMap에 접근하여 플레이어를 제거.**

**1. 싱글스레드 구조**

**결국 어떤게 호출되던간에 Sector배열과 PlayerMap은 각기 락이 걸려야하므로, 분리를 할 필요가 없다.**

**로그인을 제외한 모든컨텐츠는 Sector에 영향을 줄수밖에 없음**

**별도의 스레드를 두고 이쪽에서만 일괄관리하도록 모든 핸들러가 이 스레드에게 잡을 던지는 싱글스레드 구조가 될것이다.**

그리고 이 UpdateThread 하나에서만 플레이어와 섹터가 접근되게끔 할것이다.

**즉, IO처리를 하는 다수의 WorkerThread와 하나의 UpdateThread로 가는 형태이다.**

이는 가장 고전적인 한국IOCP의 게임서버 설계형태이기도 하다.

(IOCP입장에서는 비효율적인 형태. 그래도 Select, APC, Event 방법보다는 낫다.)

그래도 모니터링서버 등으로 여러스레드를 기능별로 빼기때문에, 코어를 다 활용하지 못할정도의 비효율적인 구조는 아니다.

우리는 이느낌으로 NetServer를 만들것이다.

((

**스레드 구조 복습**

**1. I/O를 활용하는 WorkerThread 여러개, 하나의 UpdateThread를 가진 싱글스레드 구조.**

**2. WorkerThread에서 직접적으로 치고 들어가버리는 병렬처리 구조**

**3. WorkerThread는 I/O의 일을 하도록 두고, UpdateThread를 여러개 늘리는 구조.**

이것들을 벗어난 획기적인 방법은 아직까지 없음.

))

이와 같이 여러 개의 WorkerThread와 하나의 UpdateThread를 병렬처리를 하게끔 설계를 한다면?

**싱글스레드 구조의 병렬처리를 위한 설계 1**

**A. SRWLock으로 Read/Write Lock을 분리**

**B. 섹터배열, PlayerMap이외에도 섹터하나하나 마다 동기화객체(CS)를 두고 가는방법**

**하지만 2번의 경우, 데드락에 걸릴 가능성이 굉장히 높으므로 사용할 수 없음.**

데드락을 막기위해 락을 거는 방향의 순서를 만드는 방법도 존재하지만, 이렇게 갈바에는 차라리 전체를 락걸고 가는것이 낫다.

**싱글스레드 구조의 병렬처리를 위한 설계 2**

NetServer는 멀티코어 환경에서 멀티스레드를 활용하도록 만들었다.

이와같이 컨텐츠 쪽에서도 멀티스레드로 병렬처리가 되게끔 하고싶다면..

**1. 만약 섹터가 매우 많다면, 맵단위로 락 걸기**

**사용자들이 서로간 영향을 주지않도록 분리되어있다면 맵단위로 동기화객체를 두고,**

**해당 맵에 있는 애들끼리 경합이 일어날때만 Lock을거는 형태를 고민해볼수 있음.**

이 경우에도 워커스레드 10개에 하나의 같은 맵에 존재하는 유저들의 메시지만이 몰려왔다면 역시나 Lock을 걸고 기다리겠지만, 이와같은 가능성은 흔치않음.

**수백개의 맵이 있다면 메시지가 분산되어 들어오기때문에 병렬처리를 기대해 볼수있을 것**

**의문 - 맵은 여러개로 나눴지만, PlayerMap은 하나인데?**

read/write Lock을 사용한다면 충분히 커버할수있다.

물론 굉장히 로직이 복잡해질 것이다.

**다시 채팅서버 설계 (싱글스레드)**

우리는 여러개의 WorkerThread와, 하나의 UpdateThread를 가진 싱글스레드 구조로 가기로 했다.

**결국 여러개의 WorkerThread가 하나의 Queue를 통해 잡을 Enqueue하고,**

**하나의 UpdateThread가 Dequeue하여 뽑아낸다.**

이 Queue는 LockFreeQ를 적극적으로 활용.

이 핸들러를 처리하고, DisConnect, SendPacket을 호출하는대상은 UpdateThread.

**Jop 설계**

당연히 template타입에 들어가야 하므로 객체가 될것이다.

현재 OnClientJoin, OnRecv, OnClientLeave 의 핸들러가 모두 UpdateThread에서 처리되어야 하는 설계이다.

그렇다면 이 객체안에 뭐가 들어가야 하는가?

결국 잡이라는것이 종류가 나뉘어야한다.

**1. Type**

**Join, Msg, Leave... (**어떤 핸들러를 통해 왔는지 = 무슨일을 수행해야 하는지)

**2. 수행해야할 대상**

SessionID

만약 컨텐츠쪽에서 ClientJoin핸들러를 받을 때, 해당 클라의 IP 등의 정보를 알고싶다면

**Jop큐 구조체 인자로 필요한 정보(IP등)을 구조체에 추가한다.**

단점 : 해당 정보를 필요로하지않는 타입인 경우 메모리 낭비

하지만 이정도의 메모리차지는 유의미한 차이는 없기떄문에, 개발의 용이성을 위해 넘어가도 상관없음. 찝찝하다면 다른 방법을 강구해도 상관없다.

이렇게 만들어진 구조체또는 클래스가 JopQ의 템플릿타입이 될것이다.

**Jop생성**

Jop을 생성하는 실질적인 주체는 AcceptThread, WorkerThread

사용하는쪽은 UpdateThread가 될것이다.

가장 기본적인 방법은 동적할당이며 성능과 디버깅을 위해 JopPool을 사용한다.

우리가만든 TlsFreeList를 사용할 것.

**이렇게 나온 FreeList역시 얼마나 사용되고있는지 모니터링이 가능하게끔 만들어야 할 것.**

**왜 싱글스레드 하나로 채팅서버를 만들었냐고 묻는다면?**

이후에 나중에 멀티스레드 채팅서버로 하나 더 만들어야 할 것.

이렇게 최대한 심플하게 만든다음에, 나중에 멀티스레딩으로 바꿔서 해볼 것이다.

이는 성능 때문에 싱글스레드로 가는 것이 아님. 고전적인 서버형태를 따라보는것이다.

기존의 NetServer를 가지고 채팅서버를 만든다.

**채팅서버를 멀티스레드로 간다고 한다면?**

**1. IOCP에 있는 WorkerThread들이 직접적으로 컨텐츠를 처리하도록 간다.**

**이때는 UpdateThread에서 마음대로 Jop을 뽑아 처리한다면 순서가 엉망이 된다.**

**멀티스레드 방식으로 효과를 보기위해서는 무조건 컨텐츠가 분리되어있어야 하는 전제가 되어야한다**. (또는 겉으로는 통합되어있는 것 같지만 내부에서 어떻게든 분리)

**2. IOCP WorkerThread들이 여러 개의 UpdateThread로 처리**

이때 JopQ를 하나로 간다고한다면 의미가 없으므로 UpdateThread마다 Q가 나와야한다.

채팅서버 같은 경우에는 1번이 낫다. (할수있다면 1번방식이 대부분 맞다고 생각함)

**채팅서버 검증**

채팅서버가 제대로 받아서 주변에 뿌리는 것은 약간 검증이 어렵다.

따라서 Recv / Send의 비율로 확인한다.

서버입장에서는 Send가 훨씬 많을 것이고.

클라입장에서는 Recv가 훨씬 많을 것이다.

만약 섹터하나당 한명씩 들어간다면 Send가 8~9배정도가 나오게된다.

한섹터에 세명정도씩 들어가면 27~30배정도가 나올 것.

우리는 성능과 메모리, 오류를 위주로 확인한다.

과제 : 채팅서버의 간단한 틀을 잡아와볼 것.

**1.15**

**IOCP - RunningThread조절**

CreateioCompletionPort()의 마지막인자 (스레드 조절)를 정할 때, MonitorThread와 UpdateThread도 감안해야 할까?

단순히 1초에 한번씩 출력(IO작업)만 하는 MonitorThread때문에 WorkerThread가 하나돌지 못하는 것은 아깝다.

**RunningThread개수를 상수화된 수치로 알 수 없으므로 성능테스트를 통해 잡아 볼 것.**

**실제 우리서버 외에도 다른프로세스에서 스레드가 돌고있기 때문에 일반적으로는 코어에서 두세개정도는 제외한다.**

**UpdateThread가 깨어나는 방식**

우리 채팅서버는 핸들러 함수에서 Jop을 만들어 JopQ에 Enqueue하고, UpdateThread는 JopQ에 무언가 들어왔을 때 깨어나서 일을 해야한다.

따라서 Event방식, APC Q, IOCP등 여러가지 방법을 사용하여 UpdateThread를 깨울수있다.

**IOCP를 큐로 활용**

IOCP를 사용하는 경우 Enqueue가 PostQueue CompletionStatus가 될것이고, GQCS로 깨어날 것이다. (Running스레드 인자는 의미없음. 어차피 UpdateThread가 1개이기 때문)

**장점 : Event를 사용하는 방식보다 훨씬 깔끔하고 신뢰할 수 있음**

**단점 : 내부 큐에대한 제어가 전혀되지않으며 NonPagedPool메모리를 사용하는 단점 존재.**

**큐 내부 모니터링이 필요한 상황**

예시로, 평소에는 큐에 10개 정도가 쌓여있다가 갑자기 5,6만개로 올라간다면?

UpdateThread로직에서 개선을 해보고, 결국은 동접을 낮춰야한다.

상승곡선을 그린다면 뭔가 문제가 있는 것.

이러한 상황들을 모니터링하지않고 간다면 굉장히 위험하다.

( 별도언급 : 채팅서버의 최대 가용능력을 검증하고자한다면 테스트밖에 답이없음)

**우리가 사용할 Q**

우리는 LockFreeQ로 갈것이지만, 실무에서 사용하고자 한다면 아무도 믿어주지않을 것.

해당구조에서 Q가필요하고 내가 만든 구조를 활용하기 곤란하다면 IOCP가 매우적합하다.

**UpdateThread의 Jop분기문**

UpdateThread가 깨어나서 타입별로 분기문을 타게될것이다.

**swith(Type)**

**{**

**case Join :**

**break;**

**…**

**default:**

**break;  
}**

**default로 들어온 경우**

현재 게임서버로부터 핸들링된 Jop을 Type별로 처리하는 과정이다.

유저랑은 아무런 관련이 없으므로 유저를 킥하면 안될 것.

**이는 서버문제이므로 강력한 로그 후 Crash를 내는 것이 낫다.**

만약 라이브서비스라면 고민을 해봐야할 것..

**분기문 개선 - 다형성**

함수포인터, switch-case문은 구시대적인 방식.

대부분은 다형성을 이용하는 추세이다.

**다형성으로 가고자 한다면, 부모 Jop Object를 상속받아 타입별 오브젝트가 나오게된다.**

그리고 누군지 모르는상태로 각자가 각자타입에 맞게 로직을 도는 것

물론 코드는 더 깔끔해지겠지만 switch-case문에 비해 더 느려질 것이다.

**하지만 추세가 C++문법을 사용하는 추세이기 때문에 두가지 다 만드는 것을 권장한다.**

(실무에서는 코드의 분위기와 사수의 성향에 따라갈 것)

**다형성을 적용할 경우 문제**

다형성을 적용한다면 제일처음 JopQ의 타입이 최상위 Obj클래스 타입으로 바뀌게 된다.

**결국 메모리풀 역시 Jop별로 나와야하므로 Jop별로 메모리풀이 등장하는 모양이된다**

**또 반환대상이 달라지므로 UpdateThread에서는 Jop을 반환할 때 Free를 일괄적으로 하지못함.**

**제일 큰 문제는 핸들링되는 타입이 추가될때마다 타입별 메모리풀이 추가되어야 한다.**

**범용적인 메모리풀(malloc pool) 설계**

우리가 만든 메모리풀은 엄밀히 따지면 오브젝트 풀이다.

**malloc과 같이 범용적인 메모리풀은 사이즈별로 할당이 필요하므로, MS의 LFH(LowFragment heap : 저단편화 힙)처럼 다양한 버킷단위로 관리해야 한다.**

**규격별로 메모리풀을 만들어 각자가 관리하며, 요청하는 사이즈에 가장가까운 규격덩어리를 뽑아줘야한다.**

**(단편화 = 여러 조각으로 나뉘는 현상)**

재사용 시 빈공간을 어떻게 빠르게 찾는 것이 관건이므로, 포인터를 보관해야 한다.

우리가만든 오브젝트 풀은 스택구조로 되어있지만 범용적인 메모리풀은 어떤 사이즈를 요청할지 알수없다.

힙 역시 사이즈들을 리스트로 연결하면서 사용하므로 이를 해결하지못한다면 기존힙이랑 다를게 없어진다.

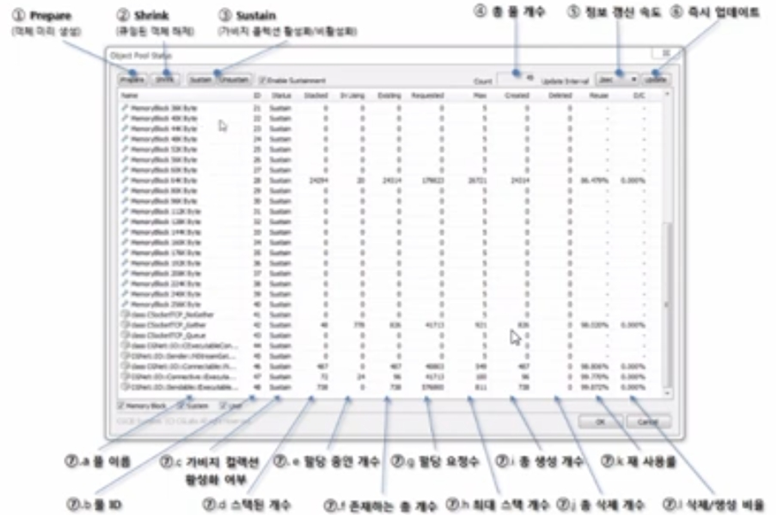
**우리가 사용하는 heap**

실제 힙에서도 일정 사이즈를 해제하고 다시금 malloc하면 해당 포인터가 그대로 들어오며 각기 다른사이즈를 계속 할당한다고 하더라도 위에서 아래로 일정하게 커지는 것을 확인할 수 있다.

**힙은 규격(버킷)을 메모리주소를 정해서 가는 것이 아니라 순차적으로 내려가면서 할당하기 때문.**

**범용적인 메모리풀(malloc pool) 구현**

이 범용적인 메모리풀 내부에는, 사이즈별 메모리풀이 별도로 존재해야 한다.



(첨부된사진는 CGC엔진의 메모리풀)

단편화를 없애고 재사용이 가능하게끔 하기위해서는 규격을 맞춰야 한다.

실지 MS의 LFH같은 경우도 버킷이 10개이상 존재하여 규격에 맞춰 뱉어주고 있음.

(규격을 상회하는 MB단위라면 VirtualAlloc을 개별적으로가서 페이지를 따로 떼어 할당한다음, 해제 요청시 바로 free한다)

이후 같은 규격을 반환후 재할당 했을 때 같은 공간의 포인터를 빼주게된다.

**우리가 이런식으로 간다면 힙이랑 다를바가 없기 때문에, (캐시로 인한 성능저하, 메모리 단편화 등 의 이유) 일반적으로 규격별로 풀을만들어 내장하는 방식을 사용한다.**

따라서 60byte를 할당요청한다면 가장 가까운 크기인 64byte규격을 할당한다.

**(모든 크기를 풀로 내장하는 것은 한계가 있기 때문)**

**단, 이 경우에는 생성자 및 가상함수테이블을 염두해야 할것이다.**

(Malloc과 같이 단순히 크기만을 할당할것인지, 객체의 생성자 호출을 위해 placement new를 호출할 것인지.)

**사용자쪽에서 Free로 해제를 요청한다면, 우리쪽에서는 어떤 버킷풀에서 나간것인지 식별할수 있있어 할 것.**

결국 CGC엔진의 메모리풀이나, 구글에서 사용되는 jmalloc, tcmalloc 등은 이러한 방식을 채택한 개량 malloc이다.

이러한 범용적인 메모리풀까지 만들어본다면 포트폴리오로서 활용가치가 높을 것.

**규격별로 메모리풀을 가지는 방식의 의문점**

메모리 낭비가 많아지지않을까?

단순히 메모리를 많이 써도된다로 결론을 내서는 안된다.

실제로 그만큼의 메모리를 사용하고있는지를 확인해야 할 것.

이 범용적인 메모리풀을 사용한다면 지금 Object타입별로 만들어진 메모리풀이 모두 포함되는 것으로, 별다른 메모리 낭비는 없어보인다.

**다시 UpdateThread의 분기문**

핸들러마다 클래스를 만들어 다형성으로 가는 형태는 커맨더패턴으로 보는게 맞다.

( 번외로, 옵저버 패턴은 옵저버라는 클래스에 가상함수를 만들어놓고 호출하면 그게 옵저버 패턴. 결국 기본원리는 다 거기서 거기며 어떻게 활용하느냐에 따라 다름.)

커맨드 패턴은 뭔가 명령을 수행하는 것이 목적인것이고,

옵저버 패턴은 다른애한테 알려주는게 목적.

어쨌든 대부분은 다형성을 활용하기 때문에 이러한 방식으로도 만들어 두는 것을 권장한다.

**성능과 최신기술**

물론 성능과 직관성을 보자면 switch - case()문이 훨씬 더 좋아보일 수도있음.

성능만을 본다면 NetServer클래스 형태도 버려야하고, 직렬화버퍼도 버려야 한다.

직렬화버퍼와 구조체를 예시로 든다면,

**RecvQ. Deque(Buffer, MsgSize);**

**PacketProc(buffer, Type);**

**{**

**Switch(Type)**

**{**

**Case Move:**

**이상황에서 구조체인경우 단순히 형변환해서 buffer.type에 접근한다.  
 }  
}**

**이때 직렬화버퍼를 사용한다면, 버퍼를 직렬화버퍼로 Copy한다음, 다시 직렬화버퍼로부터 별도의 타입자료형 변수로부터 Copy가 일어난다.**

대부분의 회사에서는 자체적으로 무언가를 만들어 쓰는 경우는 없음.

터지는 경우가 감당이 되지않기 때문이다.

보통은 프라우드넷이나 CGC엔진 등 돈을주고 쓰는 경우는 해당 업체에서 책임지고 고쳐줄것이기 때문에 있는 것을 활용한다.

또는 큰회사에서는 별도의 연구팀이 있어서 해당 연구팀이 개발한 것들을 사용한다.(ex NHN)

실제 현업에서도 STL과 new/delete등을 일반적으로 사용하는 추세이다.

**다시 UpdateTherad의 분기문 2..**

**다형성을 사용한다고해도 타입별로 오브젝트 풀을 만드는 것이 문제로 남는다.**

**타입별로 가상함수 테이블과 역할이 다를것이지만 가장 큰 이유는 크기가 다르기 때문인데,** **이는 단순히 가장 큰 타입의 메모리로 메모리풀을 사용한다면 해결된다.**

물론 가상함수 테이블이 필요하므로, alloc을 받아서 placement new는 개별적으로 가야 한다.

(생성자가 필요없다면 단순히 대입만 해주면 끝나기 때문에 placement new필요없이 대입만 하면 될것이지만, 엄밀히 따지는 경우)

**따라서 구조체든 클래스든 구조를 잡고 제일큰 크기가 200byte인 경우에 이것을 template에 넣어놓고, Alloc받아 placement new를 한다음 해당 Jop으로 사용하는 것.**

보기 좋은 선택은 아니지만, 우리환경에서 메모리풀을 통합시키기 위해서는 이러한 방법밖에 없음.

풀을 달리가도 상관은 없지만 일단은 방법은 숙지해 놓을 것.

**서버 설계**

우리가 지금 설계하는 채팅서버는 하나의 서버군내에서 게임서버와 세트를 이루는 채팅서버를 만들고자 함이다.

**그리고 로그인서버가 별도로 나올 것이다.**

**클라이언트는 로그인 서버를 통해 로그인을 하고, 다음으로 채팅서버와 게임서버로 각자의 커넥션을 가지고 두개로 연결한다.**

**로그인 서버**

**로그인서버는 통합회원정보 DB랑 연동하여 ID/PS와 접속할 서버를 확인하여 토큰을 발급한다.**

**그리고 이 값을 어떻게든 게임서버에게 넘겨줘야 하는 상황이다.**

이후 클라이언트는 토큰을 가지고 채팅서버에서는 부담없는 가벼운 인증만을 거친후 접속한다.

결국 로그인서버에서 1차적인 검증을 통한다음, 게임서버와 채팅서버에 접속하는 것.

여기서 토큰이란 유추할 수 없는 1회용 키를 뜻한다. (보통64byte ~ 128byte크기)

**로그인 서버가 있는 이유**

결국 클라이언트는 로그인서버에서 회원정보를 검증받아 토큰을 발급받고, 게임서버와 채팅서버에서 또한번의 검증을 거쳐야 한다. 왜 굳이 이런 확인을 하는가?

**우리 NetServer는 어떤곳에서든지 Connect할수있으므로, 클라이언트는 로그인서버를 거치고왔다는 것을 입증해야한다.**

로그인서버 웹서버가 담당하는 부분은 두가지이다.

**1. DB연동 (회원정보DB) 의 부하를 줄인다.**

**2. ID/PS 에 대한 노출문제.**

**단, ID/PAS에 대한 암호화는 웹서버가 하는역할이다.**

NetServer - NetClient간 연결에서 프로토콜은 암호화를 해도되고 안해도 딱히 상관없다고 언급해왔다.

어차피 stateful에서 메모리에 모든 데이터를 가지고있고, 게임을 플레이할 때 오가는 메시지들은 공개되어도 별다른 문제가 없기 때문이다.

**단, ID/PAS는 개인정보이므로 게임서버 프로토콜로 노출시키는 것은 절대로 안된다.**

**만일 비대칭키 RSA를 구현할 정도의 수준이라면 해도되지만, 이정도면 리눅스나 안드로이드를 개발한것과 같은 수준이 된다.** (가히 세계최고수준)

**더군다나 만들었다고 해도 어마어마하게 느리기 때문에, 모든통신에 사용하는 것은 불가능하다.**

(키교환 알고리즘에 대해서는 이후 다시 언급한다.)

**번외) 암호화에 대한 얘기**

**대칭키 암호화란, 암호화하는 키와 복호화하는 키가 똑 같은 암호화를 뜻한다.**

**반대로 비대칭키 암호화란 인코딩 키와 디코딩 키가 서로 다른 암호화를 뜻한다.**

(비대칭키 암호화는 특정 인코딩한 키로 암호화를 했으면 특정 디코딩키로만 풀수가있음)

NetServer-NetClient간의 간단한 암호화는 키가 같으므로 대칭키 암호화이다.

**비대칭키 - RSA암호화**

**RSA암호화는 어마어마하게 큰 수를 대상으로 소인수분해를 통해 브루트포스(무작위 대입법)으로 는 풀기 불가능한 시간이 나오게끔 만드는 암호화다.**

(128암호화, 256암호화, 512암호화 1024암호화 같은 것들은 숫자가 비트크기를 뜻한다. 64비트 표현범위도 어마어마한테 1024의 경우는 상상도 할수없음)

이를 구현하고자한다면 저 크기를 감당할만한 로직부터 만들어야 한다.

더군다나 수학적으로 접근하는 암호화이기 때문에 엄청나게 느리다.

**대칭키 - AES암호화**

**대칭키 암호화중에서도 표준인 AES암호화 역시 국방부에서 사용할 정도로 믿을만 하지만, 결국엔 키가 외부로 노출된다는 위험성이 존재한다.**

**따라서 노출조차 되지않는 비대칭키가 가장 좋지만, 굉장히 느리기 때문에 대부분은 대칭키와 비대칭키를 섞어서 사용한다.**

(https역시 키가 노출되는 것을 막기위해 키에대한 공유만 비대칭키 RSA암호화를 통하고, 다음으로 대칭키 AES암호화로 전환한다)

**게임에서의 암호화 (로그인 인증)**

**어쨌든 게임에서도 클라이언트로부터 로그인이라는 인증을 거쳐야 하기 때문에, 요즘은 로그인서버를 웹으로 사용하는 경우가 많다.** (굳이 로그인서버가 직접짠 CPP서버로 존재할 이유는없으므로)

이 경우는 클라이언트에서는 https로 안전하게 통신 후 토큰을 발급받아 게임서버 등으로 진입을 하게된다.

**해당 토큰은 우리의 프로토콜을 통해 전송되므로 노출되겠지만, 인증토큰은 1회용이므로 노출되어도 상관없음.**

아파치나 웹서버 같은 경우도 오픈소스를 사용하기 때문에 우리도 오픈소스를 사용해도 상관없으나, 이러한 오픈소스를 가져다 쓰는것조차 어렵기 때문에 시간낭비 할 필요가 없음.

결국 우리는 로그인 인증쪽에서 웹서버의 https를 사용하여 토큰을 감춘다.

이렇게 암호화가 필요한 서버를 우리가 직접만드는 것은 매우 위험하다.

**다시 로그인 서버**

결국 우리는 웹 로그인서버를 통해 토큰을 얻고, 우리 게임서버로 인증토큰을 들고오고, 게임서버에서는 인증토큰을 확인한다.

따라서 앞으로 드릴 프로토콜 안에도 SessionKey라는 항목이 존재한다. (테스트환경에서 필요함)

또한 인증토큰을 거쳤다는 전제로 가기 때문에 무조건적으로 허용하고 진행한다.

**로그인서버는 데이터를 유지할 필요없이 ID/PAS만을 확인하면 되기 때문에 stateless로 간다.**

**따라서 ScaleOut이 가능한 형태로, 여러대로 늘리는것도 가능하다.**

**서버의 토큰확인**

클라이언트는 먼저 로그인서버와 연결하여 토큰을 발급받은 이후, 서버를 선택하여 선택한 서버에 연결한다.

그렇다면 서버는 클라이언트가 들고온 토큰을 어떻게 확인하는가?

**서버군이 수백개가 생길수 있고, 로그인서버역시 scaleOut이므로 여러대가 나올수있는 상황이다.**

**방법1.**

**로그인서버가 모든 게임서버에게 해당토큰을 뿌리는 것.**

단점 : 부담이 매우 클것이다.

**방법2.**

**유저 로그인이후, 어떤 서버를 선택했는지 까지를 로그인서버에게 보내고 로그인서버는 해당 게임서버에 전달한다.**

**단점 : 로그인서버와 클라이언트간의 연결을 계속 물고 있어야 한다.**

어차피 서버선택하는 시간이 오래걸리지않으므로 크게 문제되는 부분은 아니다.

**방법3.**

**유저 로그인이후, 어떤 서버를 선택했는지 까지를 로그인서버에게 보내고, 유저가 게임서버에 연결시 게임서버가 로그인서버에 연결하여 토큰을 확인한다.**

**단점 : 로그인서버가 많아지는경우 이 역시 부담이 커지게되고, 로그인서버에서 언제까지 토큰(데이터)를 들고있어야 하는지에 대한 문제가 있다.**

**방법4.**

**별도의 토큰서버를 둔다.**

**결국 다수의 로그인서버와 다수의 게임서버가 존재하기 때문에 부담이 일어나므로, 이를 체크하는 별도의 게임서버를 두는 것이 가장 일반적이고 좋은 해결책이다.**

**토큰DB vs 토큰 서버**

**이 서버가 DB가된다면 게임서버가 DB와 연동해야하므로 부담이 생길것이며, DB자체도 매우 커지기 때문에 부하가 클것이다.**

(DB를 사용하는 방법도 많이 사용하는 방법)

**결국은 우리가직접 서버를 만들거나, 메모리DB인 레디스를 사용하는 것이 가장 좋다**

레디스는 메모리DB이기 영구 보관이 아닌 휘발성 성격을 띄고있기 때문에 오히려 좋다.

인증이 끝난 토큰을 따로 보관해야할 이유가 없기때문.

로그인서버**(토큰서버 말하는거같음)**는 Account번호와 거기에 해당하는 토큰만이 필요하므로, 레디스를 사용하지않는다면 해쉬테이블이나 STL의 unordered\_map을 두고 push/pop만을 해주면 끝난다.

**직접만든 토큰서버 vs 레디스사용**

**레디스는 모든 언어를 대상으로 모든 프로토콜 라이브러리가 만들어져있기 때문에 웹에서 사용하기 굉장히 편리하다.**

(대부분 로그인서버를 웹으로 간다)

더군다나 레디스자체가 텍스트를 기반으로 가기때문에 연동도 쉽다.

**그런데 토큰서버를 레디스가 아닌 우리가 직접 개발한다면, 웹이라는 환경에서 실제 TCP서버에 연결할 방법도 고민이 되어야한다.**

물론 웹에서도 소켓/포트를 열고 연결하는 것이 불가능한 것은 아니지만, 레디스가 굉장히 용이함

**Redis**

**레디스는 관계지향형DB가 아닌 Key-Value형태이기 때문에 우리입장으로 본다면 맵이랑 비슷하다.**

단 일반적인 해쉬와는 달리 레디스는 Json을 받아들이기 때문에 더 다양한 기능이 존재한다.

(JSON은 오브젝트로, 어떤키에 해당하는 오브젝트의 기능이 된다.)

**단 이렇게 범용적으로 활용되고 다양한 기능을 제공하고, 프로토콜 자체를 스트림으로 분석까지 하기 때문에 위와같이 로그인서버가 하는일 정도로만 사용한다면 우리가 만든 서버와 비교한다면 성능이 떨어질 수밖에없다.**

대신 게임서버를 만들어 사용할 경우 문제가 터졌을 때 독박을 쓰기 때문에..

그리고 추세가 레디스로 가는 추세.

요새는 네트워크 라이브러리조차 없는 경우가많음. 되도록이면 있는 소스를 가져다 쓰는 추세.

**실제 물리서버의 위치**

**위와 같은 설계는 토큰서버나 로그인서버가 모두 머신으로 따로 존재한다고 가정하고 설명한 것.**

해당 컴퓨터는 내부에 MySQL, 또는 레디스 같은 것들을 설치하고 갈것이며, 하나의 머신당 하나의 각각 역할을 하는 프로세스가 메인으로 있다는 느낌으로 설명한 것.

실제로 한 컴퓨터에 서버프로세스와 DB프로세스를 같이 올리는 경우는 잘 없음.

단, 사용자가 없다면 굳이 돈낭비를 할필요는 없으므로 같은곳에 둘수도 있겠다.

**우리가 구현할 채팅서버**

우리는 위와 같은 설계대로 가지 못할 확률이 크다.

테스트를 위한 환경에 맞춰서 가고 있기 때문에, 우리가 만든 걸 기준으로 생각한다면 실제 라이브서비스 환경에서 대응을 하지못할 수 있음.

따라서 위와 같은 설계들을 반드시 숙지해 둘 것.

**중간 정리**

**1. 클라이언트는 로그인서버와 커넥트하여 ID/PAS를 인증받아 토큰을 발급받는다.**

**2. 로그인서버는 토큰을 생성하여 토큰서버(또는 레디스)와, 클라이언트한테 발급한다.**

**3. 클라이언트는 발급받은 토큰을 통해 게임서버에 접속한다.**

**4. 게임서버는 토큰서버에 가서 토큰이 유효한지 확인한다.**

(게임서버가 토큰서버에 직접접근하여 가져올수도 있고, 요청할 수도 있다.)

토큰서버는 결과만을 전달한다.

**1.18**

**메모리풀 가비지컬렉터**

**일정시간 동안 할당한 메모리공간을 사용하지않는다면, 자동적으로 해제해주는 기능을 말한다.**

만약 가비지컬렉터까지 검증이 가능하도록 구현한다면 판매가 가능할정도의 퀄리티가 된다.

**단, 어차피 사용할것인데 굳이 해제를 할 필요가있는지는 의문.**

(가비지컬렉터에 관해서는 논문이 쓰여질정도로 수준이 높고 의견이 갈림)

가비지컬렉터 스레드를 단순히 FREE할때 깨운다면, Free를 하지않는다면 오래도록 쓰이지않는 메모리도 해제되지 않는다.

(C#에서도 가비지컬렉터가 돌떄마다 시스템자체가 버벅이는 현상이 있음.)

**타임아웃 기능**

타임아웃 기능을 넣는다고 가정할때,

**네트워크 라이브러리단에서 체크 vs 컨텐츠쪽에서의 체크**

현재 NetServer에는 주기적으로 polling하는 스레드가 없으므로, 만약 사용자쪽에서 하트비트 기능을 사용하지않는다면 낭비라고 생각할 수도있다.

하지만 게임이외의 대부분이 필요하기때문에 넣어도 무방하다.

단, 현재 우리는 SessionArray가 필요할때, 즉 반응이 왔을때 깨어난다.

**이때 하트비트 확인이 들어온다면 해당 Session에 대해 IO가 있을지 없을지 모르기때문에 이러한 부분을 잘 테스트하여 검증한뒤 넣어야 할 것.**

**채팅서버 프로토콜**

TCP Fight에서는 CS와 SC가 프로토콜의 방향을 뜻했지만, 채팅서버 프로토콜에서는 Req, Res로 방향을 의미하게된다.

서버-서버간의 통신은 SS로 표기한다.

**단, 게임이라는 것은 클라가 별다른 요청이 없더라도 메시지를 보내는 경우가 많기 때문에, Request와 Response에 대한 용어에 대해 조금 안맞는다고 느낄 수 도있다.**

실제로 프로토콜 이름을 잡을때는 자유롭게 해볼 것.

**당연히 모든 프로토콜에는 어떤 프로토콜인지 Type이 들어간다.**

//#ifndef \_\_GODDAMNBUG\_ONLINE\_PROTOCOL\_\_

//#define \_\_GODDAMNBUG\_ONLINE\_PROTOCOL\_\_

enum en\_PACKET\_TYPE

{

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

//

// Client & Server Protocol

//

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

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

// Chatting Server

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

en\_PACKET\_CS\_CHAT\_SERVER = 0,

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

// 채팅서버 로그인 요청

//

// {

// WORD Type

//

// INT64 AccountNo

// WCHAR ID[20] // null 포함

// WCHAR Nickname[20] // null 포함

// char SessionKey[64]; // 인증토큰

// }

//

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

en\_PACKET\_CS\_CHAT\_REQ\_LOGIN,

이 프로토콜은 클라이언트가 로그인서버로부터 안전하게 인증을 받은다음 들어올 것이다.

**AccountNo**

클라이언트는 본인을 알릴수있도록 Account번호를 들고올것이다.

**ID**

ID는 없어도 상관없으며, 지워도 무방하다.

Account번호가 없더라도 ID만 들고온다면 대응되는 Account를 찾는것도 가능하다.

현재 채팅서버 설계는 DB와의 연결을 0으로 만드는것이 목적이므로, 필요한 정보를 클라가 직접 들고오는 상황이다.

따라서 클라쪽에서 닉네임 조작이 가능한 상황이다. 일단은 허용한다.

**이전에 설명한대로 설계하여 토큰서버가 존재한다면, 닉네임역시도 공유하여 확인하는 절차를 거칠수도 있겠다.**

**NickName**

채팅서버의 경우 섹터단위로 범위도 전송되고 하다보니까 누군가로부터 채팅이 왔지만, 화면에 보이지않을 수있으므로 닉네임을 전송한다.

(그리고 게임서버와 채팅서버가 완전히 분리되어있기 때문에 동기화가 약간 다를 수있음.)

만약 클라이언트가 채팅서버로부터 채팅메시지를 받았는데, 해당 주체인 Account번호가 내 클라이언트에 없는 경우 메시지를 출력하지 않겠다고 하는 등의 설계라면,

**클라이언트가 닉네임표현을 한다면 굳이 서버가 가지고 갈 필요는 없다.**

**NULL문자를 포함하는 이유**

보통 가변 문자열을 표현할때 길이-문자열로 표현할 것이지만, 길이가 정해진 경우에는 굳이 길이를 포함시킬 필요가 없기때문에 NULL로 간다.

**인증 토큰**

현재 우리서버는 인증자체를 어딘가로부터 받았다라는 전제로 가기때문에, 일단은 모두 허용.

**참고로 세션 인증토큰은 호환성을 위해 바이너리가 아닌 아스키코드(텍스트)로 표현한다.**

**로그인 요청 프로토콜**

게임 서버측에서는 이 로그인요청 프로토콜을 받으면 확인하는 작업을 거쳐야한다.

자기 메모리에 있는 토큰을 확인하거나, 별도의 토큰확인서버에게 확인작업을 거칠수도 있음.

이러한 검증이 끝나면 로그인 결과가 나온다.

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

// 채팅서버 로그인 응답

//

// {

// WORD Type

//

// BYTE Status // 0:실패 1:성공

// INT64 AccountNo //Debug

// }

//

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

en\_PACKET\_CS\_CHAT\_RES\_LOGIN,

**로그인 응답 프로토콜**

로그인에 대한 응답 프로토콜이다.

**여기서는 단순히 성공/실패 여부만을 전달하지만, 실제 라이브서비스라면 성공실패에 대해 세분화할 필요성이 있다.**

Account번호에 대한 토큰 없음, 토큰이 일치하지않음, 중복로그인 등등의 상황이 있을 것.

**AccountNo**

**AccountNo는 클라이언트가 로그인서버를 거치면서 가져왔기때문에, 본인은 알고있는 상태이다.**

**그런데도 AccountNo를 재차 주는 이유는, 디버깅을 위함.** (AccountNo불일치시 Crash();)

(크래시나 메모리누수의 경우는 서버를 바라보기만해도 확인되지만, 더미측에서 판단해야 하는것들이 생각보다 많다. 이러한 이유로 밑에서 나올 프로토콜에도 계속해서 AccountNo가 등장할것. )

**실제 라이브 서비스에서는 이 항목이 빠지는 것이 맞다.**

**라이브서비스에서의 컨텐츠 추가**

라이브서비스에서 새로운 컨텐츠가 추가될때마다 지금처럼 더미테스트를 하면서 장시간 테스트하는것은 불가능하다.

그나마 할수있는건 새 컨텐츠를 추가하고 클라이언트 4,5명 붙여서 정상작동 되는지 확인하는 것.

또는 테스트 서버가 존재한다면 최소한 일주일정도라도 거친 후에 본섭으로 나가게 된다.

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

// 채팅서버 섹터 이동 요청

//

// {

// WORD Type

//

// INT64 AccountNo //Debug

// WORD SectorX

// WORD SectorY

// }

//

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

en\_PACKET\_CS\_CHAT\_REQ\_SECTOR\_MOVE,

**클라이언트의 섹터이동 요청 (Req)**

클라이언트가 게임서버에게 섹터이동을 요청하는 프로토콜이다.

가장 처음 로그인하는 경우 섹터를 할당받지 않기 때문에 특정값으로 초기화 해놓으며(-1) 섹터가 정해지면서 시작된다.

출발지와 목적지까지 전송하는 것이 아니라 단순히 목적지만을 알려준다.

서버에서는 알아서 처리할 것.

**AccountNO**

위에서 설명했듯, AccountNo는 본인이 알고있지만 디버깅을 위해 재차 전송한다.

**섹터좌표 X,Y**

섹터좌표가 게임서버에 오면 이동이 이루어진다.

실제 게임서버라면 세션의 섹터좌표를 이동시키기 전에 검증을 한번 거쳐야 할것이다.

(말도안되는 요청일 경우 무시)

지금은 더미가 랜덤으로 이동하기때문에 이러한 안전장치는 넣지않는다.

물론 단순히 채팅이므로 굳이 막지않아도 상관없음.

**섹터이동의 전제조건**

로그인만 성공하고 게임을 시작하지않은 상태 (=섹터에 진입하지않은 상태)라면 섹터좌표의 프로토콜을 받아서는 안된다.

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

// 채팅서버 섹터 이동 결과

//

// {

// WORD Type

//

// INT64 AccountNo //Debug

// WORD SectorX

// WORD SectorY

// }

//

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

en\_PACKET\_CS\_CHAT\_RES\_SECTOR\_MOVE,

**서버의 섹터이동 결과 응답(RES)**

섹터 이동이 완료됐다면 클라이언트에게 결과를 전송한다.

사실 클라가 섹터이동을 요청한 프로토콜과, 서버가 이동완료했다는 프로토콜은 같음.

그래도 게임서버에서 처리할때는 실제 플레이어 객체의 섹터좌표를 참조하여 보내 확실하게 테스트를 할것.

**TCP Fight와 섹터이동처리 비교**

TCP Fight에서는 나름 복잡했지만, 채팅서버에서는 단순히 A섹터에서 뽑아 B섹터에 추가하면 끝.

비교적 간단하다.

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

// 채팅서버 채팅보내기 요청

//

// {

// WORD Type

//

// INT64 AccountNo

// WORD MessageLen // byte

// WCHAR Message[MessageLen / 2] // null 미포함

// }

//

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

en\_PACKET\_CS\_CHAT\_REQ\_MESSAGE,

**클라이언트의 채팅보내기 요청**

클라이언트가 게임서버에게 채팅을 보내도록 요청한다.

**MessageLen**

글자 길이가 아닌 byte를 의미한다.

가변길이로 가기때문에 당연히 null은 포함하지않는다.

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

// 채팅서버 채팅보내기 응답 (다른 클라가 보낸 채팅도 이걸로 받음)

//

// {

// WORD Type

//

// INT64 AccountNo

// WCHAR ID[20] // null 포함

// WCHAR Nickname[20] // null 포함

//

// WORD MessageLen

// WCHAR Message[MessageLen / 2] // null 미포함

// }

//

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

en\_PACKET\_CS\_CHAT\_RES\_MESSAGE,

**채팅서버가 클라이언트에게 채팅을 보냈다고 응답**

채팅서버를 다이얼로그로 만들었을 때는 자기자신이 보낸 채팅은 본인에게 다시 보내지않았다.

하지만 여기서는 자기자신에게 다시 보내는데, 이또한 검증을 위함.

**ID, NickName**

위에 언급했듯, A클라이언트가 B클라이언트의 채팅메시지를 받았는데, B클라에 대한 정보가 A클라한테는 없을 수도 있다.

이를 위해 넣은것. 설계에 따라 없어질 수 있음. 지금은 맞춰야 할 것.

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

// 하트비트

//

// {

// WORD Type

// }

//

//

// 클라이언트는 이를 30초마다 보내줌.

// 서버는 40초 이상동안 메시지 수신이 없는 클라이언트를 강제로 끊어줘야 함.

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

en\_PACKET\_CS\_CHAT\_REQ\_HEARTBEAT,

**더미가 게임서버에게 보내는 하트비트 프로토콜**

사실 테스트 환경에서도 이걸 안보내도 상관없음.

더미는 미친듯이 메시지를 보내기때문에 TimeOut에 걸릴일이 존재하지않음.

TimeOut건에 문제가 있었는데, 확인해서 다시 알려드림

};

//#endif

**OnClientJoin에서 할일**

할일이 있을지 없을지 생각해볼 것.

**Dummy 문제**

내 주변 8개의 섹터에게 채팅을 뿌리고, 모두 제대로갔는지 확인이 되고,

온 메시지에대해 내 섹터 내부에 있는 클라이언트 인지 확인이 된다면 가장 좋을 것이나..

매우 복잡해지므로 이렇게까지는 하지않음(너무느림)

더미는 최대 5000명이 들어갈것인데, 위와같이 디버깅이 되고자한다면 한 클라이언트가 본인포함 주변 9개 섹터에 대한 유저를 모두 관리해야한다.

그리고 나갔는지 안나갔는지 또한 모두 파악해야한다.

따라서 A클라가 B클라한테 채팅을 보냈는데 B가 받지 못하거나,

A클라가 본인 섹터 바깥에 있는 C클라에 채팅을 보냈다고 해도 알 방법이 없음.

이걸 감지하려고한다면 가능이야 하겠지만 너무 느리게됨.

그리고 프로토콜 더욱더 확장되야할것.

다른클라에는 누가나갔고 들어왔고에 대한 정보자체가 없을것이기 때문이다.

실지 현업에서도 이러한 환경이 제공된다면 좋겠지만, 시간이 없을 것이다.

라이브서비스에서 더미조차 없는 경우가 허다하다.

추가적으로 보통은 일정을 짤 때 구현까지만을 생각하지, 디버깅은 생각하지않는다.

테스트섭이라도 있다면 마음이 편하겠지만, 그게아닌경우 매우 불안할 것.

**NetServer의 마지막 검증**

이 채팅서버는 NetServer를 확인하기위한 마지막 프로젝트이다.

이안에서 최대한 문제가 나와서 문제를 잡으면 좋을것

**설정파일**

이제부터는 서버의 구색을 갖춰야 한다.

컴퓨터가 바뀌거나, IP가 바뀌었을때 간단하게 설정을 바꿀수있게끔 세팅해야한다.

실제로 퍼블리싱을 받을때, 아예 서버에 접근조차 하지못하는 경우가 생길 수 있다.

**Bind IP**

대부분은 0.0.0.0으로 갈것이지만, 만약을 위해 설정파일로 뺀다.

이더넷을 여러개 설치한다면 서버/클라가 나뉘어져서 들어올 것이기 때문.

유저들이 어디로 들어오든 상관은 없음.

단, 서버가 들어오는 별도의 환경(ex 별도의 크라우드넷)이 구축되어있다면 분리되어야 할 것.

현재 우리 서버환경 역시 마찬가지이다.

2U서버 컴퓨터에는 외부와 연결되어있는 라인과 10.대로 private네트워크로 존재한다.

**Port**

생략. 당연히 지정해야하는 부분

**WorkerThread / RunningThread**

실제 몇개의 워커스레드를 생성하고, 돌릴것인지에 대한 설정.

**MAX\_CLIENT**

동접자

**MONITOR\_NO**

모니터렁 서버 연동.

모니터링 서버를 연동하게되면 서버마다 고유번호가 부여가되서 모니터링하는 기준으로 잡아야 하기때문에 이 모니터 번호를 이용한다.

**Packet Encode Key**

PACKET\_CODE = 119

PACKET\_KEY = 50

패킷인코딩이라는 간단한 고정키를 가지고 간다.

코드라는것도 부여가 되고있음.

이러한 키를 설정파일로 빼는경우, 파싱하여 넣어줘야 한다.

이때 LanServer가 파싱의 주체가 되어서는 안된다.

**SystemLog Level**

어떤 로그로 진행할 것인지에 대한 로그레벨

**서비스에 해당하는 설정파일**

**TIMEOUT\_DISCONNECT = 30000**

타임아웃 시간을 뜻한다.

설정파일로 너무 많은 설정을 빼는것은 권장하지않는다.

**채팅서버 더미**

더미는 1U, 게임서버는 2U에 넣고 진행한다.

로그인서버는 현재 사용하지않는다.

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

//

// 더미클라이언트 Config

//

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

:LOGIN\_SERVER

{

// 현재 사용 안함

IP = "127.0.0.1"

PORT = 20002

}

**로그인서버 연동시 사용한다. (지금은 로그인서버 사용하지않음)**

:CHAT\_SERVER

{

IP = "127.0.0.1"

PORT = 12001

}

**여기에 여러분의 IP와 여러분의 포트를 적어서 연결.**

:DUMMY\_CLIENT

{

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

// 접속트래픽 설정

//

// 총 접속자 수는 THREAD\_NUMBER \* CONNECT\_CLIENT

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

THREAD\_NUMBER = 30 // 스레드 개수 최대 100

CONNECT\_CLIENT = 50 // 스레드 당 클라이언트 수 최대 50

이 더미는 Select를 기반으로한 멀티스레드 구조이다.

한스레드에서 최대 50명까지, 스레드는 최대 100개까지로 설정가능하다.

즉 30스레드 \* 클라50 인경우 총 1500의 커넥트가 발생.

채팅서버가 엉망이어서 CPU점유율을 많이 차지할경우, 더미를 여러개 키지못함.

총 15000명의 커넥트로 테스트할 것이기 때문에 세개까지 켜져야한다.

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

// 1 Loop 당, 1 Client 의 확률 조절

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

RAND\_CONNECT = 100

RAND\_DISCONNECT = 3

RAND\_CONTENTS = 80

connect가 100인경우 끊어진 클라는 이번루프에 재연결을 시도한다.

connect가 10인경우에는 10째 루프에서 재연결을 시도한다.

대부분은 100으로가지만 조금이나마 연결에대한 것을좀 낮추고싶다면 이하로 낮춰도됨.

Disconnect는 백분율로, 3인경우 3%확률로 끊는것을 말한다.

루프를 기준으로 책정한 확률이기 때문에, 3정도만 되어도 굉장히 많이 끊게된다.

Contents는 섹터이동과 채팅. 역시 백분율을 뜻한다.

섹터이동과 채팅이 1:3비율로 실행된다.

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

// 액션 딜레이 설정 (밀리세컨드)

//

// 스레드 루프마다 딜레이가 입력 된다.

//

// 그러므로 스레드가 10개이면 최대 10명은 딜레이와 상관 없이 동시에 액션을 취할 수 있다.

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

DELAY\_ACTION = 1000 // 액션 딜레이 - 실제 유저와 비슷한 시뮬레이션을 위해 조절한다.

::추가적인 딜레이 값이다.

클라이언트가 딜레이값을 기준으로 한번씩 행동한다.

DELAY\_LOGIN = 1000 // Connect 후 로그인 요청까지의 딜레이

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

// ID List - 텍스트 (유니코드) 파일로 AccountNo ID Nick 이 세로로 입력되어 있어야 함.

//

// 최대 10개 지정이 가능하며 더미클라 첫 시작시에 선택한다.

// 각 파일마다 최대 1만개의 ID 입력 가능

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

LIST\_ID\_0 = "ID0.txt"

LIST\_ID\_1 = "ID1.txt"

LIST\_ID\_2 = "ID2.txt"

LIST\_ID\_3 = "ID3.txt"

LIST\_ID\_4 = "ID4.txt"

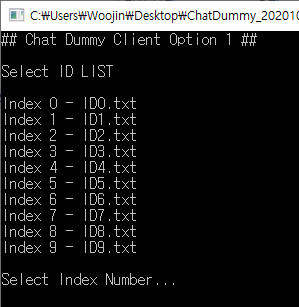
LIST\_ID\_5 = "ID5.txt"

LIST\_ID\_6 = "ID6.txt"

LIST\_ID\_7 = "ID7.txt"

LIST\_ID\_8 = "ID8.txt"

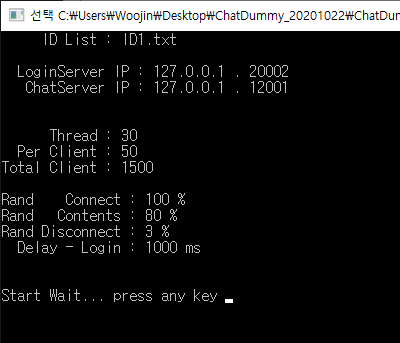
LIST\_ID\_9 = "ID9.txt"



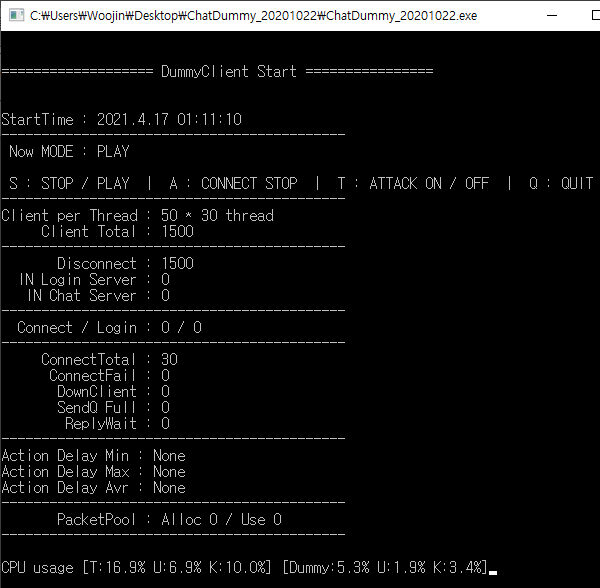
::AccountNo, ID를 텍스트에 저장해 뒀음.

실제 더미를 두 개이상 키게되면 아이디를 선택하게한다. (중복되면 안됨)

그리고 번호 지정하면



}

-

**S (Stop) STOP : 모든더미의 행동정지**

**A (Connect Stop): 연결만 중단**

만약 디스커넥트 확률이 세팅된 상황에서 A를 누를경우 connect클라가 점점 줄어들기만 할것

**T (Attack)** : 공격.

지금은 안해도된다. 문제된 클라를 서버에서는 걸러내야한다.

**Connect / Login**

Connect는 커넥트만 된 클라이언트 / 로그인은 로그인까지 성공한 클라이언트.

이는 서버에서도 카운팅되어야 하는 수치이다.

**ConnectTotal**

DownClient로 연결끊김이 감지됐을때 이 Total값을 가지고서 확인할것이기 때문에 항상 AccpetTotal값을 카운트해야할 것이다.

**ConnectFail**

접속이 몰릴때 백로그큐가 모두 다찼을때, ConnectFail이 뜰수있음.

이때는 DownClient수치가 올라갈수있음. 따라서 이ConnectTotal 수치를 비교하여 확인할 것.

**DownClient**

서버가 임의로 끊은 것이다.

ConnectTotal값과 비교하여 백로그큐 문제인지, 서버문제인지 파악할 것.

**SendQ Full**

어떤 특정세션에 Send링버퍼가 다찼을때를 얘기한다.

SendQ Full도 볼이유는없음.

어찌됐든 서버가 Recv를 느리게 하고있다는 뜻이다.

간혹 더미가 SendQ가 다차는경우 링버퍼가 다차면 큐잉하다가 포기해버리기 때문에 더미가 느려서 이 수치가 나오는 경우도 있다.(가능성은 거의없음)

내캐릭터가 아무일도 하지않더라도 섹터내에 캐릭터들이 나한테 Send한다.

단 이를 Send해주는 부분은 서버가 하는일이기 때문에, 거의 나올일이 없다.

만약 나온다면 더미쪽의 문제일 가능성이 크다.

내 서버는 문제가 없는데 메시지가 이상하게 오고 더미가 느리게 작동되고있다면 Send하는 과정에서 문제가 생겼다고 봐도된다.

**ReplayWait**

리플레이는 말그대로 기다리는 상태이다.

모든 메시지는 Req에대해 Res가 무조건 가는 구조이다. 따라서 클라는 Req를 보내고 Res를 기다리는 형태가 된다. (100% 핑퐁구조임)

따라서 클라가 Sector이동/채팅전송 등을 서버에서 Req한다음에 Res가 오기전까지는 아무행동도 하지않는다.

이 수치는 올라도 상관없다. 단 빨리빠지는게 좋을 것이며, STOP했을때 0으로 떨어져야한다.

0이아닌경우에는 서버가 날린것이기 때문이다.

**Action Delay MAX / MIN / AVR**

더미는 딜레이 값 보낼때와 받을때의 시간을 계속 체크하고있다.

단, 더미가 느려서 느리게 보낼수 있으므로 이 수치가 100% 서버의 RTT라고 볼수는없음.

**PacketPool**

클라의 패킷풀이다. 신경쓸필요 없음.

**더미 사용설명서**

#define dfPACKET\_CODE 0x77

#define dfPACKET\_KEY 0x32

지금 define해놨지만 설정파일로 빼야한다.

#define dfSECTOR\_X\_MAX 50

#define dfSECTOR\_Y\_MAX 50

섹터범위 : 0 ~ 49

더미는 랜덤하게 섹터범위 내를 돌아다닐 것이다.

ChatDummy\_2019xxxx.exe채팅더미 실행파일

ChatDummy\_Config.ini 채팅더미 설정파일

ChatList.txt 채팅더미 채팅 메시지 파일 (노래가사)

ID0.txt ~ ID9.txt 채팅더미 계정 파일 (실행하면 번호를 눌러서 지정 함)

1. ChatDummy\_Config.ini 파일을 편집하여 IP, 더미 수,패턴을 설정 함.

최대치 : 스레드 100개 / 스레드당 클라 50개 = 5,000 클라

# 먼저 RAND\_DISCONNECT 수치를 0 으로 하여 끊김없는 상태로 테스트

# 끊기 없는 테스트 완료 후, RAND\_DISCONNECT : 0 ~ 10 으로 재연결에 대한 테스트

4. 서버를 켜고 더미를 켠다.

더미 실행 후 0번 ~ 9번가지 번호키를 지정하여 ID 리스트를 선택 함

여러개의 더미를 켤 경우 ID 리스트를 모두 다른 것으로 선택 해야 한다.

S 키 - 재접속 및 컨텐츠 송신을 일시 중단 / 해제

A 키 - 재접속을 중단한다. (ConnectTotal 을 1000단위로 맞춰서 중단)

T 키 - 공격 패킷을 전송한다. (이건 제일 마지막에 테스트, 지금 할 것이 아님)

Disconnect : 연결이 끊어진 클라이언트 수

Connect / Login : 연결만 한 클라 수 / 로그인까지 성공 수

ConnectTotal : 더미가 지금까지 connect() 호출 한 횟수

ConnectFail : connect 호출시 실패한 횟수

DownClient : 서버에서 임의로 끊은 수

SendQ Full : 더미 클라이언트의 SendQ 가 꽉찬 상태

ReplyWait : 컨텐츠 패킷을 보내고 응답을 대기중인 클라 수

# 확인사항

1. 서버가 클라이언트를 임의로 끊는 상황 체크.

서버가 임의로 접속을 끊는 경우 DownClient 로 카운팅이 되고 있으나

과도한 부하가 생길 경우 이 감지에 오차가 가끔 발생.

그러므로 아래의 방법을 통해서 확실한 확인이 필요함.

- 서버에서는 AcceptTotal (서버가 켜진 후 Accept 를 호출한 총 횟수) 가 집계 되어야 함.

- 각 더미의 ConnectTotal 수치의 모든 합은 서버의 AcceptTotal 과 같아야 함.

예)

서버 AcceptTotal : 345000

더미1 ConnectTotal : 300000

더미2 ConnectTotal : 40000

더미3 ConnectTotal : 5000

더미의 ConnectTotal 총합과 AcceptTotal 이 같기 때문에 문제가 없음.

이에대한 확인은 장시간 테스트 후 모든 더미에 A 키를 눌러 1000단위로 컨넥션을 중단.

그리고 더미의 ConnectTotal 합을 구함.

더미 : ConnectTotal = connect() 횟수 - DownClient 횟수

서버 : accept() 횟수.

진짜로 서버가 임의로 끊어버린 경우는 서버와 클라이언트의 connect total 수치가 맞지 않게 됨 accept 횟수가 높게 나옴.

**더미를 복잡하게 가는 이유**

단순히 스트레스 테스트의 목적만이 아니다.

단순한 스트레스 테스트가 목적인 더미인 경우 마구잡이로 쏟아부으면 끝난다.

(기능상 오류를 잡기위함)

**OnClientJoin**

OnClientJoin에 대한 질문.

OnClientJoin에서 플레이어를 생성해야 된다고 했었다.

그런데 Accpet됐을때 여기서 바로 Player를 만드는게아니라, 로그인 인증된 유저들을 Player로 만드는게 낫지않나?

지금 주어진 기능들을 만들려고 설계를 짜다보면 답이나오기때문에 구현하면서 고민해볼 것.

파서를 도저히 못믿겠다고 생각한다면, JSON을 써도 상관없다.

**암호화에 대해 추가적인 언급**

클라가 뭔가를 조작해서 우리의 암호화를 뚫고 들어오는 행위는 절대 막을 수 없다.

(이걸 아무도 트집잡지 않음)

(우리가 한 암호화 역시 너무 단순하고 초보적인 단계)

어떤 암호화를 한다고해도 패킷안에 복호화키 까지 모두 들어가 있기 때문에 어쩔수없다.

**따라서 이 암호화를 뚫고 들어온 로직이 우리 서버에 적용유무가 중요하다.**

**1.20**

**채팅서버 딜레이**

더미를 15000명을 넣은경우, Action딜레이는 보통 200이하로 나오게 된다.

1U에서 2U로 쏘는 클라의경우 200정도, 클라와 서버를 같이 돌리는 2U는 0에 가깝게 나온다.

일단은 오류가 없는지가 우선이다.

성능은 이후의 문제.

**채팅서버 모니터링**

더미수치뿐만아니라, 서버에 다양한 항목들을 넣어 모니터링 해야한다.

**UpdateThread개수, TPS개수, 한번 깨어났을때 몇회를 처리하는지, UpdateTHread가 얼만큼 블락이 걸려서 멈추는지, 패킷 풀, 업데이트 메시지 큐 등 모두 확인할 것.**

**따라서 더미수치에만 의존할 것이아니라, 서버에 여러가지 수치를 넣어서 어디서 어떻게 느려지는지 모두 파악해야 한다.**

(당연히 이러한 수치를 체크한다면 성능이 떨어지므로 문제파악이 끝난경우는 막아야할것.)

프로파일러는 구간을 특정지어 세분화하여 100n/s단위로 쓸 때 사용하는것이므로 목적이 다름.

(SessionMiss, SessionNotFount는 로그인서버 연동시에 설명한다.)

**UpdateThread 성능비교 시**

UpdateThread에 대한 블락이나 동기화 시그널을 받는것에 대해,

Event Signal / APC Queue / IOCP / JOP Q 등 어느것이 좋을것인지 성능 테스트를 할 때는 어떤 수치를 보고 어떻게 판단하는것이 좋을 것인가?

**초당 몇회 깨어나는지**

**한번 꺠어나서 얼마의 시간동안 처리하는지**

등의 수치를 기준으로, 내부에서 어떻게 돌아가고있는지 파악해야한다.

(외부에서 메시지가 쏟아져 들어오는데 내 스레드가 늦게 깨어날 수있음.)

**전혀 예상외의 결과가 종종 나오기때문에 반드시 숙지할 것.**

**1.22 부근 채팅서버 언급**

더미는 헤더에대한 부분, 컨텐츠 메시지에 대한 부분. 최대한 조작해서 보낸다.

중복어택은 빠져있는데,

중복 로그인을 재현하고 싶다면 더미를 두개를 같은 아이디로 키면 된다.

당연히 중복로그인은 기본적으로 막혀야 한다.

**보통은 헤더 값이 이상한 경우가 대부분일 것이고, 이 부분은 네트워크 라이브러리단에서 커버되어야한다.**

그리고 컨텐츠단에서도 구별해줘야 하는 부분이 존재한다.

하나씩 확실하게 체크하여 이상한 클라는 그 즉시 끊어내야한다.

이렇게하면 생각보다 큰 문제는 없을 것

아무이상 없다고 하더라도 시간을 길게 잡고 테스트해야한다.

어택은 로직적인 부분이므로 몇시간만 검증을 거쳐도 무방하다.

그 이후로는 Disconnect쪽을 봐야할 것.

**공격/방어하는 과정에서 Disconnect가 많이 발생되고 여기서 다른세션과 동기화문제가 일어남**

AcceptTPS는 1500~2000을 권장한다. 너무 높게나와도 테스트가 안될 수 있음.

연결해서 송신하다가 끊는 등의 상황 등이 나와줘야 할 것.

보통은 멀쩡한 세션이 끊기는 등의 문제가 나오고, 원인해결을 못하고 있음.

**Cancel I/O**

Cancel I/O는 걸려있는 비동기 IO를 끊는다.

그리고 우리는 자연스럽게 IOCount가 0이 되어 Release타는 것을 유도한다.

**CancelIoEx는 현재 걸려있는 IO에 대해서만 취소한다.**

만약 수신완료통지든 WSARecv를 걸기직전에 캔슬IoEx를 한다면 WSARecv를 바로 걸어버리니까 문제가 됨. 이는 당연히 재등록하지 않게끔 막고 IOCount 0을 떨어뜨리도록 유도해야한다.

(간발의 차로 못막을수도 있음. 그럼 그다음에라도 막아서 0떨어지게 해야함)

우리가 closesocket을 한방에 하지않는 이유는 소켓자체가 재사용되기 때문이다.

(락을 건다면 이렇듯 복잡하게 갈 필요가 없음)

**캔슬IO이외에도 shutdown()을 사용해 끊기는 들어가되 리소스는 반환되지 않게끔 할수있음.**

**하지만 shutdown()의 문제는 TIME\_WAIT이 남는다는것에 있다.**

상대가 FIN을 보내지않으면 끊어낼수 없고, 수십초가 지나야 OS에서 정리한다.

문제는 이런식으로 끊어내는 대상은 이상하거나 TIME\_OUT된 클라들인데,

타임아웃에 걸려 끊으려고 했는데 끊기지 않는다?

**그렇다면 다음로직에서 계속하여 끊기를 시도하기 때문에 서버의 불필요한 부담이 된다.**

캔슬IO는 일반적으로 잘 사용하지않음.

대부분은 락을 걸고 closesocket으로 편리하게 끊어낸다.

**우리 서버의 특징**

우리는 최대한 복잡하게 고생하려고 하다보니 이 모양이 되었음.

우리 네트워크 라이브러리의 장점은 락프리.

락프리가 성능이 더 좋을것인가? 락프리가 빠른것인가?에 대한 질문이 면접에서 나올 것.

성능이 나쁘지않냐 너무 복잡하지않나 이렇게물어오면 더 다양한 경험을 위해서 했다고 하면 됨.

**채팅서버 Attack문제**

Attack부분은 알아서 고친다. (문제있는 클라는 알아서 끊는다.)

잘못된 데이터긴한데 기준이 애매한 경우는 개인적으로 질문하면 알려드림.

채팅메시지를 받을 때 지역으로 한번 보관을한다.

이때 길이가 애매하다고 하여 동적할당 할 필요가 없음.

모든 채팅은 최대길이를 정하도록 되어있다. 그냥 1000byte잡아도 상관없음

**1.27 채팅서버 언급**

**Update TPS 수치**

업데이트 메시지가 초당 15000 ~ 16000정도 나오게 할 것. (업데이트처리)

이는 더미에서 얼만큼의 채팅을 보내느냐의 문제이므로, 해당수치가 나올때까지 더미조절할 것.

더미에서 딜레이 같은 것들을 조절하면 될 것이다.

기본세팅이 Disconnect : 3 이고 초당 딜레이가 1000,1000이다

기본세팅 기준으로 맞추지말고, 서버의 업데이트TPS를 먼저 맞춘다. (이걸 15000정도)

**어쩄든 서버에서는 초당 1200 ~ 1500 ~ 2000 내외로 들락날락 하면된다. (Accpet말하는거같음)**

AcceptTPS같은 경우 ClientJoin에서 측정한다.

UpdateTPS는 ClientLeave와 OnRecv ? (X) . UpdateThread에서 카운팅 할 것.

(UpdateThread 의 처리는 Join, Leave, Chatting, Sector이동 이 있겠다)

**통신량 예측**

**현재 이 더미와 채팅서버는 주변섹터에 채팅이 제대로 뿌려지는지 전혀 확인이 되지않는다.**

**따라서 이를 단순히 통신량으로 예측하기로 했다.**

**이 통신량은 인원이 늘어나면 당연하게도 비율이 바뀌어야한다.**

(결국은 RecvTPS패킷당, SendTPS패킷당을 카운팅하여 수치를 볼 것)

**채팅서버의 부담은 대부분 Send가 될것이다. (Recv받으면 주변섹터에 모두 Send해야 하므로)**

Send수치가 잘 안나오는 경우 서버가 빨라질것인데 주변섹터에 잘 뿌리고있는지도 파악이안된다면 답이없음.

SendTPS는 SendPacket 호출회수로 본다. PDH아님

|  |  |  |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

나 포함 내 주변섹터에 한명씩만 있다면 1 : 8비율

|  |  |  |
| --- | --- | --- |
| 2 | 2 | 2 |
| 2 | 2 | 2 |
| 2 | 2 | 2 |

나 포함 내 주변 섹터에 두명씩 있다면? 2 : 32 -> 1 : 16

섹터내부에 인원이 많아질수록 비율도 같이 늘어나게된다.

**<2.3>**

**::1:14**

채팅서버가 공격없이는 잘 실행됨. 근데 어택하면 재연결을 꺼도 끊기는애가 생김.

5,10초 돌아가다가 한번 올라가는 느낌. 그렇다면 끊기는 애를 어떻게 찾을까?

다운클라가 발생했을 때 멈춰두고 어디서 문제가 생겼는지 봐야할 것이다.

그런데 이 다운클라를 어떻게 잡아낼 것인가?

**시도해본 방법**

시스템 로그를남겨 킥한 클라내역을 확인해도 찾을 수 없고,

공격대응 로직도 문제없고,

Disconnect에서 잘못된 대상을 조회하여 끊었나 봐도 문제가 없다면..

**해결책**

특정상황을 지어야 하는데 VS디버깅 도구, 나의 로그 이런것들로는 해결이 되지않는다.

attack같은 경우 잘못된 패킷을 판단한 후에 disconnect를 시도한다.

**끊어지는 것에는 다 이유가있기 때문에 세션마다 그 이유를 기록하는것도 괜찮은 방법.**

그래도 못찾는다면, (공격도 안하고 아무도 재연결안하는데 내 코드실수로 끊어진다. 그런데 어느지점인지 찾을 수없음) 더미를 활용한다.

더미가 끊는 상황에 멈춰봤자 의미가없다. 이미 끊겼고 서버는 그시점을 지났기 때문.

따라서 더미에 연결이 끊겨야 하는 세션인 경우 이것이 끊어야 한다는

(TCP재전송 역시 전송실패를 보낼 방법이 없으니 정상패킷 자체를 보냄)

**연결이 끊겨야 하는 클라인경우 그것을 식별할 수있게 메시지를 보내는것이다.**

**(이게 오지않고 끊어진다면 이상한 클라로 식별이 가능함)**

채팅메시지에는 equal(=)이 하나간다. 끊기직전에 무조건 =에 대한 회신이 와야 더미는 끊는다.

(그냥 끊는건 없음 Disconnect확률로서.)

나이제 끊을거야 하고 이거에 대한 회신이와야지만 closesocket.

**<2:00:30>**

updateTps가 15000이 나와야하는데, 이는 평균선이 아닌 최소 하한선이 15000이라는 뜻이다.

(Join, Leave등 모든 것을 포함한다. 만명으로 본다)

**Send/Recv비율**

그런데 send/recv비율은 어떻게 계산해야 할까?

섹터 2500개, 유저 15000명인 경우 나누기하여 주변섹터에 몇 명. 곱하기하여 메시지는 몇 개.

마냥 어림짐작으로 계산하는것보다는 직접 로직에서 구하는 것을 권장한다.

(주변에 메시지를 뿌리지않는 패킷감안, 한섹터에 몇 명인지, 비율은 어떻게 분포되었는지 등 코드상에서 직접 구한다음 계산을 하는 것을 권장한다.)

이렇게 했을 때 세개의 더미가 모두 200m/s아래로 딜레이값이 나와야한다.

메모리풀 등 네트워크 라이브러리의 모든 것을 손봐야한다.

인원을 맞추고자한다면 더미는 스레드 100개 50명(5000명)으로 세개키는 수밖에 없을 것.

대신 재접속 하는애와 안하는애가 나뉠것이니 어느더미에 딜레이를 더 주거나 적게 줄것인지는 알아서 조절 할 것.