Raknet 네트웍 게임엔진 한글번역 프로그래밍/Network2015.10.11 01:34

UDP패킷

패킷의 전달은 UDP와 TCP로 보내는 방식이 있다. UDP는 전송딜레이가 적은 대신 데이터의 신뢰성이 떨어지고 TCP는 그 반대다. 따라서 게임에는 UDP가 적격이다. 하지만, UDP의 문제점들이 있다.
1. 전송도중에 패킷이 소실될 가능성 있다(재전송이 요구된다)
2. 수신측에서 수신받은 패킷들의 순서가 바뀔 수 있다(수신측에서 순서의 재배치가 필요)
3. 일단 수신된 패킷의 데이터는 무결하지만, 전송중에 가로채어 수정될 수 있다(패킷데이터의 변경여부의 체크필요)
4. UDP패킷은 접속허용의 선행처리가 필요없다. 즉, 권한없는 데이터를 받을 위험이 있다(권한부여필요)
5. UDP전송은 흐름제어와 데이터집적이 없다. 그래서 수신자의 수신용량이 초과되어 더 이상의 전송이 무의미해질 수도 있다(흐름을 통제하고 데이터컨트롤이 요구)

내부처리방식

*RakNet*는 server, client, peer의 3대 주요 위상(위치와 상태)적인 시스템을 이루고, 특히 server와 client는 peer에 Receive함수안에서 다루어지는 추가적인 특성이 부가된 전문화된 인스턴스이다.

peer가 생성되면, poll상태나 업데이트상태하에 작동가능한 **업데이트쓰레드를 생성**한다. 폴상태는 Receive가 통상의 원리에 따라 호출되고 있는지를 알아보기위해 체크한다. 만약 그렇다면, 이 쓰레드는 다음 폴타임까지 슬립상태로 들어간다. 그렇지 않으면 업데이트상태로 들어간다. 만약 IO completion port(IOCP)가 사용된다면, 두 개의 추가적인 쓰레드가 IO completion port쓰레드풀에서 생성된다. 이 풀은 *RakNet*의 모든 인스턴스에서 공유된다.

파싱된 메시지를 보낼때는 압축이 사용될 수도 있고, 신뢰적인 레이어로 전송된다. 이 신뢰층은 메시지가 MTU하에서 적합하게 분리될 필요가 있는지의 여부에 따라서, InternalPacket의 오브젝트풀에서 InternalPacket오브젝트를 하나이상 생성하거나 얻어온다. 이 InternalPacket 클래스는 메시지를 기술하는 모든 다양한 파라메터를 포함한다. 그리고나서 이 오브젝트는 사용된 우선순위레벨에 따라서 몇몇의 리스트 중 하나에 저장된다.

업데이트쓰레드의 주요책임은 신뢰층의 Update함수를 호출하는 것이다. 이 Update함수는 프레임을 전송할 시간인지를 알아보기 위해 체크한다. 만약 그러하다면, 프레임은 유저메시지, 패킷응답확인 및 신뢰할 수 있는 재전송을 포함하여 생성된다. 이 프레임이 암호화되고 통계가 업데이트되고 패킷은 전송된다. 전송윈도우가 꽉차거나 또는 더 이상의 데이터가 없을 때까지 이것은 루프로 돈다.

메시지가 도착하면 암호해독이되고 유효화되며 개별적인 메시지로 분해된다. 이 윈도우크기는 업데이트된다. 만약 분리된 메시지들이 전부 도착하면 원래의 메시지로 재조립된다. 만약 이 메시지가 확인되고 중복되고 순차적이거나 정렬된 것이라면 이것은 적절히 다루어진다. 이 때 유저에게 전달될 의도가 있는 메시지는 RakPeer로 전송되며, 만약 압축되어 있다면 압축해제된다. 커넥션로스트와 같은 특정한 시스템메시지는 직접 다루어질 수도 있다. 특정 메시지와 유저메시지는 풀에서 할당된 Packet구조체로 들어갈 것이며, 뮤텍스가 설치된 큐에 저장될 것이다. Receive가 호출되면 이 큐에서 pop되고 pop된 패킷은 파싱된다. 함수호출을 대신할 수 있는 RPC패킷과 같은, 어떤 종류의 패킷은 유저에게로 반환되지 않는다. 만약 패킷이 반환되면, 때때로 패킷은 분산네트워크오브젝트나 RakVoice 또는 마스터서버와 같은, 또 다른 유저레벨시스템에 대한 것일 수도 있다. 만약 이것(RPC패킷?)을 쓴다면, 이것들은 Multiplayer Class에서 다루어질 수 있다.

주요 클래스

대부분의 클래스는 파일이름과 일치한다. 또한 각 클래스의 헤더에는 모든 함수에 세부적인 정보가 포함되어 있다.

RakPeer

*RakNet*의 코어는 RakPeer이다. 이것은 보통 작업할 수 있는 가장 하위레벨에 있다. 이것은 P2P 네트워킹시스템으로 흐름제어 및 다양한 오더링과 우선순위의 결정스키마를 가진 진보된 신뢰적인 UDP를 다룬다. 이것은 어떤 토폴로지로도 구성될 수 있고, *RakNet*의 대부분의 주요 기능을 포함하고 있다.

RakServer

RakServer는 RakPeer의 상위계층에 있고, 오직 접속요청만 허용하고 외부로의 연결은 하지 않는다. 이것은 리모트시스템의 정적데이터와 동기화된 랜덤넘버씨드를 재정의할 수 있는 추가적인 기능을 유저에게 제공한다.

RakClient

RakClient는 RakPeer의 상위계층에 있고, RakServer와 같이 작동하도록 디자인되어 있다. 이것은 오직 하나의 외부로의 연결만 허용하고 외부로부터의 연결요청을 받아들이지는 않는다. 이것은 유저에게 추가적인 정보를 제공한다. 예를 들면, 원격 클라이언트의 연결이나 연결해제 이외에 동기화된 랜덤넘버씨드와 같은 서버가 제공하는 기능도 있다.

RakVoice
RakPeer의 상위계층에 있어서 목소리의 인/디코딩의 외부라이브러리를 쓴다

BitStream
BitStream은 *RakNet*전체에 걸쳐서 사용되는 헬퍼클래스이다. 이것은 데이터를 비트수준의 스트림으로 인코딩이 가능하게 한다. 이것은 또한 데이터의 압축 및 해제를 다루므로 보다 효율적으로 데이터를 인코딩할 수 있다. *RakNet*의 대부분의 함수들은 비트스트림을 사용하며, 특히 RPC와 같은 파라메터 길이가 가변적인 함수에 쓰이고 있다.

Multiplayer.h
Multiplayer는 헬퍼클래스로써 패킷에 대한 체크와 식별자(ID)에 기초한 패킷의 파싱 및 핸들러함수(Multiplayer Class의 virtual 멤버함수임)를 호출하며, 패킷의 할당해제를 할 수 있다. 이것은 패킷을 받는 골격일 뿐이다. 그러나, 이것은 분산오브젝트패킷에 대한 분산오브젝트매니져를 호출하여, 이 패킷들을 다루는데 생기는 트러블을 막을 수 있다.

DistributedNetworkObject
DistributedNetworkObject는 네트워크상에서 자동으로 객체자신을 퍼트리거나 멤버변수를 동기화할 수 있는 능력을 가진 객체의 **기본클래스(부모클래스)**이다. 이것은 NetworkObject로부터 파생되어, 파생된 모든 인스턴스가 각자 고유의 식별자(ObjectID)를 가져서 네트워크상에서 이 인스턴스들을 참조할 수 있다. 이것은 클라이언트/서버 환경에서 작동해야하고, DistributedNetworkObjectManager의 싱글톤을 가진 클라이언트 또는 클라이언트/서버의 등록의 선행과정이 요구된다.

주요특징
이 모든 특징은 peer, server, client에 의해 제공된다. 이것들은 클래스인터페이스에서 하나 이상의 함수를 통해 구현된다. 역시 헤더파일에 자세한 내용이 있다.

IO Completion ports - 현시점에서는 쓰지 않는다
윈도우상에서는, IO completion ports라는 메카니즘이 있는데, 이것은 메모리의 내용을 네트워크카드로 직접쓰거나 네트워크카드로부터 메모리로 직접읽을 수 있다. 이것은 네트워크데이터가 대기중일 때 핸들러쓰레드를 깨운다. 이 쓰레드는 클라이언트/서버의 다수의 인스턴스가 공유하는 풀과 같은 커몬풀에 있다. 이것은 수천개의 클라이언트를 다룰 때 퍼포먼스를 향샹시킬수 있을 것으로 본다.

Remote Procedure Calls(RPC)
원격프로씨져호출은 다른 시스템상의 특정한 프로토타입을 가진 함수를 호출할 수 있게 한다. RPC 함수는 비트스트림을 가져서 어떠한 크기의 데이터타입이라도 다룰 수 있다. 또한 커스텀패킷식별자를 생성하는 것보다 RPC를 사용하는 것이 훨씬 쉽다.

Timestamping
패킷안에서 인코딩된 발송시간기록은 여러 시스템간의 각 시간이 서로 달라도, 언제 다른 컴퓨터에서 이벤트가 발생했는지를 정확히 결정하는데 도움이 된다.

Global Data Compression
*RakNet*은 클라이언트와 서버에서 송수신되는 전형적인 트래픽을 분석하고, 압축을 그것에 적용할 수 있다. 이것은 보통 약 30%이하의 대역폭의 사용만 될 수 있다. 그러나 추가적인 CPU 사용이 요구된다. 대부분의 다른 특징과 달리, 이것은 유저의 설정과정부분에서 어떤 작업이 요구된다.

Automatic Memory Synchronization
이것은 Distributed objects에 의해 쓸모없게된 낡은 특징이다. 어떻게 수신을 호출할 때마다 매번 특정한 메모리의 블록의 변경을 체크할 수 있겠나. 어떤 메모리가 변경되었다면, 그것은 그것을 동기화시켜주는 원격시스템으로 패킷을 전송하는 것이다.

세부구현
세부 구현

*RakNet*을 구현하기 위해서는 client, server 또는 peer의 인스턴스를 얻어야 한다.
헤더포함

PacketEnumerations.h	패킷의 상수이다. 이것은 *RakNet*이 쓰기로한 이미 정의된 패킷에 대한 종류를 아이디(native packet identifiers)로 부여한 것이다. 예를 들면, 디스커넥션통지같은 패킷을 들 수 있다. 만약 나만의 타입을 지정하려면 파일 끝에 정의하면 된다.
RakNetworkFactory.h	디자인 패턴의 팩토리의 구현으로써, RakPeer, RakServer,그리고 RakClient의 포인터를 얻는것으로 사용된다. 이것은 DLL사용시에 반드시 필요하다.

RakPeerInterface.h	RakPeer 클래스를 위한 인터페이스이다. 클라이언트나 서버대신 사용한다면, RakClientInterface.h나 RakSeverInterface.h를 사용하면 된다.
NetworkTypes.h	*Raknet*에서 사용되는 구조체들을 정의해 놓았다. *PlayerID* - 개별클라이언트 또는 서버 또는 peer를 구별하는 고유한ID *Packet* - API에 의해 반환되거나 데이터를 얻을 경우 및 메시지송신이 요구될 때 쓰이는 구조체. 한마디로 패킷이다

개체생성
```
RakClientInterface* client = RakNetworkFactory::GetRakClientInterface();
RakServerInterface* server = RakNetworkFactory::GetRakServerInterface();
RakPeerInterface* peer = RakNetworkFactory::GetRakPeerInterface();
```

클라이언트의 연결
```
client->Connect(serverIP, serverPort, clientPort, 0, 0);
```

serverIP 또는 host	접속할 서버의 IP주소의 문자열 ex) “192.168.1.1” 자신의 시스템에 접속하려면 “127.0.0.1” 또는 “localhost”로 대신할 수 있다. 도메인네임도 가능하다
serverPort	접속할 서버의 포트번호. 0~65535까지의 가능하지만, 보통 32000 이후의 번호를 쓸 것.
clientPort	클라이언트상에서 데이터의 송수신을 위한 포트로써, 서버포트와 같은 역할을 한다. 클라이언트포트와 서버포트는 서로 같아도 되지만 이러한 경우에는 같은 시스템에서 서버와 클라이언트를 동시에 띄울 수 없다.
0	연결유효정수값. 훗날 호환성을 위해 사용하지 않은채로 0으로 남겨둔다.
0	쓰레드의 업데이트 주기. 밀리세컨드 단위로 지정된다. 게임은 빠른 처리를 요구하므로 0이 적당하다.

주의할 점은 클라이언트의 접속은 비동기적라는 것이다. 즉, 이 Connect함수는 서버와의 접속성공여부와는 상관없이 접속을 시도한 것만으로 즉시 true를 반환한다. 서버에 접속될 때까지 블락상태가 되지 않는다는 말이다. 따라서 이 함수의 반환값만으로는 접속성공여부를 알 수 없고 IsConnected()로써 알아내야 한다. 아니면, 서버로부터 ID_CONNECTION_REQUEST_ACCEPTED메시지를 받으면 접속이 성공된 것이다.
친절한 호스트는 무효한 포트인 경우 ID_UNABLE_TO_CONNECT_TO_REMOTE_HOST라는 네트워크 메시지를 클라이언트에 알리기도 한다. 이러한 방법은 클라이언트에서 즉시, 연결대기를 중단할 수 있게 한다. 불친절한 호스트는 이러한 경우에 메시지를 통보하지 않아서 클라이언트에서 임의로 중단해야 할 것이다.
*RakNet*은 매우 빠르게 접속한다. 그래서 만약 짧은 시간에 접속이 성공하지 못하면, 접속은 바로 중지될 것이다.

서버의 연결
```
server->Start(2, 0, false, serverPort);
```

2	허용가능한 최대 클라이언트수를 정한다
0	연결유효정수값. 클라이언트접속과 마찬가지로 훗날 호환성을 위해 사용하지 않은채로 0으로 남겨둔다.
false 0	~~쓰레드의 우선순위에 대한 boolean값이다~~ 소스와 틀리다. 서버의 쓰레드업데이트 주기이다(밀리초단위). 보통 0으로 주어 최상의 속도를 낸다.
serverPort	서버의 포트번호

서버가 리스닝 상태, 즉 클라이언트의 접속을 받아들일 준비를 하겠다는 함수이다.

서버를 하나의 플레이어로써 허용하는 게임인 경우, 플레이가능한 실제 플레이어의 수는 서버가 지원하는 클라이언트 수보다 한명 많게 된다(서버도 게임에 참가하므로). 만약 서버가 전용서버이거나 클라이언트와 서버가 같은 시스템에서 게임할 수 있도록 프로그램했다면(비추천), 플레이할 수 있는 명확한 사람의 수는 그에 따라서 변경될 수 있다.

P2P의 연결
```
peer->Initialize(10, 60000, 0);
peer->SetMaximumIncomingConnections(4);
```

10	연결가능한 최대 Peer의 수. * 여러 peer들과 혼합하여 연결한다면, 받는 접속 수와 내보내는 접속의 합이다. * 순수 클라이언트로 사용한다면 1이 될 것이다. * 순수 서버로 사용한다면 접속가능한 클라이언트의 수가 될 것이다.
60000	포트번호. 클라이언트의 포트번호와 같은 개념이다.
0	~~우선순위~~ ~~0-보통, 1-높음 2-매우높음~~

	소스와 틀림. Peer의 쓰레드업데이트 주기이다. 낮은 퍼포먼스에서는 30ms가 권장되고, 보통 0을 쓴다.
4	허용하는 최대 받는 접속자수. * 현재 연결된 peer의 수가 이 수치이상이면 더 이상의 접속은 받지 않는다. * 이 수치가 peer의 최대연결갯수보다 크면 자동으로 최대연결갯수로 지정된다.

패킷수신

```
Packet *packet = client->Receive();
Packet *packet = server->Receive();
Packet *packet = peer->Receive();
```

Receive함수로 공통된다. **수신패킷큐**에서 가장 최근에 수신한 패킷을 반환하며, 패킷이 없다면 NULL(0)을 리턴한다.

Multiplayer class에서는 ProcessPackets()를 호출하여 패킷을 자동처리하게 할 수 있다. 패킷이 시스템메시지(PacketEnumeration.h에 이미정의된 ID)인 경우에는 해당 virtual핸들러가 실행된다. 이 핸들러는 디버그모드에서는 콘솔창에 패킷정보를 출력해줄 뿐, 아무런 역할도 하지 않는다.
만약 유저메시지인 경우는 ProcessUnhandledPackets가 자동으로 호출된다. 이것을 파생클래스에서 재정의하여 유제메시지프로씨져로 사용할 수 있다.

데이터는 엔진으로부터의 메시지(이미정의된 ID)와 동일한시스템 또는 다른 컴퓨터로부터의 다른 *RakNet*인스턴스로부터의 메시지가 있다.
이 두 메시지는 같은 방법으로 다루어진다(메시지나 패킷이나 같은 의미로 본다).

```
struct Packet
{
	PlayerID playerId;
	unsigned long length;
	unsigned long bitSize; // length와 같지만 bit단위로 나타낸다. length는 쓸모없게 되어서 나중에 호환성을 위해 남겨둔다.
	char* data;
};
```

PlayerID
이 패킷의 출처(송신자)이다. 즉, 연결된 모든 클라이언트는 자동으로 할당된 고유한 playerID를 가진다. 특정한 미리정의된 네트워크 메시지는 이 필드(playerID)를 사용한다.
예를 들면 ID_REMOTE_DISCONNECTION_NOTIFICATION라는 메시지는 서버가 어떤 클라이언트에게 다른 클라이언트가 디스커넥트된 것을 알리는데, 이 때 playerID는 그것이 어느 클라이언트인지 나타내는 것이 된다.
UNASSIGNED_PLAYER_ID는 "Unknown"에 대한 예약된 값이다.
length와 bitSize
이것은 data멤버에 할당된 사이즈를 나타낸다. length는 byte단위로 bitSize는 bit단위로 나타낸다.

패킷 분석
패킷을 받으면, 데이터를 분석해야 한다. 보통 데이터의 첫번째 바이트는 패킷타입을 나타내는 수치가 들어간다. 하지만 항상 그렇지는 않다. 쉽게하기위해서, Multiplayer.cpp에 ID를 얻기위한 함수가 준비되어있다.

```
unsigned char GetPacketIdentifier(Packet *p)
{
	if ((unsigned char)p->data[0] == ID_TIMESTAMP)
		return (unsigned char) p->data[sizeof(unsigned char) + sizeof(unsigned long)];
	else
		return (unsigned char) p->data[0];
}
```

이것은 패킷이 타임스탬프인 경우에만 패킷타입이 data의 첫번째에 있지 않으므로 그것을 골라내서 ID를 반환하는 것이다.
이것은 unsigned char를 반환한다. 이 반환값은 PacketEnumeration.h에 나타난 enum에 대응하는 값이다.

네트워크 엔진은 특정한 메시지를 오직 클라이언트에만 아니면 서버에만, 아니면 양쪽다 보낼 수도 있다. 자세한 설명은 PacketEnumeration.h에 나와있다. 염두할 중요한 사항은 ID_NEW_INCOMING_CONNECTION과 ID_CONNECTION_REQUEST_ACCEPTED의 메시지다. 이것들은 서버나 peer가 새로운 접속된 클라이언트를 받았거나, 클라이언트나 peer가 접속이 성공하였을 때 받는 메시지이다. 이 시점후부터 메시지를 보낼 수 있다.

만약 패킷의 ID가 미리정의된 ID가 아니라면, 다른 시스템에서 보낸 유저데이터를 받은 것이다. 이 때, 이 데이터를 디코딩해서 게임에 적절히 적용할 수 있다.

패킷 해제
데이터를 처리했다면 패킷을 할당해제해야한다. 단순히, DeallocatePacket로 보내버리면 된다. multiplayer class인 경우에는 자동으로 처리된다. 그래서 만약 multiplayer class를 사용한다면, 굳이 DeallocatePacket를 호출할 필요는 없다.

```
server->DeallocatePacket(p);
client->DeallocatePacket(p);
peer->DeallocatePacket(p);
```

이 세개의 인터페이스는 같다. Note! delete연산자로 패킷을 해제하지 말 것!

Multiplayer class는 단지 클래스(server나 client, peer)의 패킷을 읽어서 데이터의 첫번째 바이트를 분석하여 ID를 찾아내고, 패킷의 할당해제를 1주기로 모든 패킷을 다 읽을 때까지 반복할 뿐이다(ProcessPackets에서).
이 클래스는 템플릿화되어 있어서 client, server나 peer를 가질 수 있다. 비록 multiplayer class를 사용하지 않더라도 이것을 보는 것은 가치있는 일이다. 왜냐하면 패킷을 파싱할 때는 아마도 이와 같은 방식으로 할 것이기 때문이다. 또한 이 클래스에는 각 패킷ID를 문서화하여 유용할 것이다.

데이터의 송신

아래는 데이터송신의 최상의 방법을 보여준다.

```
const char* message = "Hello World";

For our server:                                    // 서버와 peer의 Send인터페이스는 같다
server->Send((char*)message, strlen(message)+1, HIGH_PRIORITY, RELIABLE, 0, UNASSIGNED_PLAYER_ID, true, true);

For our client:                                    // 클라이언트의 전송목적지는 언제나 서버다
client->Send((char*)message, strlen(message)+1, HIGH_PRIORITY, RELIABLE, 0);
```

첫번째 파라메터는 전송할 byte스트림이다.
두번째 파라메터는 전송할 byte스트림의 크기이다. 이 예에서는 Null종료문자열을 보내므로 message길이에 1을 더한 크기이다.
세번째 파라메터는 패킷의 우선순위이다. 우선순위는 단순하게 높은순위의 패킷이 낮은 순위의 패킷보다 먼저 전송된다.

```
enum PacketPriority
{
        HIGH_PRIORITY,
        MEDIUM_PRIORITY,
        LOW_PRIORITY,
        NUMBER_OF_PRIORITIES
};
```

네번째 파라메터는 다섯가지 값중 하나가 된다.

```
enum PacketReliability
{
```

	기능	장점	단점
UNRELIABLE,	UDP를 통해 직접전송된다.	UDP헤더안 50bytes 의 네트워크승인절차가 필요없다. 중요하지 않거나 패킷이 소실되더라도 새로운 패킷으로 대체가 가능한 매우자주보내야하는 데이터에 적합하다.	패킷순서가 달라질 수 있다. 송신버퍼가 꽉찬상황에서 송신실패된 패킷은 영영 도착하지 못할 수도 있다.
UNRELIABLE_SEQUENCED,	UNRELIABLE과 같지만 가장 최근의 패킷만 송신한다.	UNRELIABLE에 견주어 오버헤드가 거의 없다. 오래된 데이터로 현재 값이 갱신되는 것을 막을 수 있다.	패킷의 송신실패와 도착하더라도 수신실패가능성이 매우높다.
RELIABLE,	목적지까지 안전하게 도착하기위해 신뢰층에 의해 관리되는 UDP패킷이다.	패킷은 확실히 목적지에 도달한다.	재전송 및 인증과정으로 상당한 대역폭이 요구된다. 네트워크가 바쁘면 패킷도달이 지연될 수 있다. 수신측의 패킷오더링이 없다.
RELIABLE_ORDERED,	RELIABLE의 특성에 오더링이 더해진 것이다.	패킷은 보낸 순서대로 확실히 목적지에 도달한다. 매우 쉽게 프로그래밍할 수 있다. 오더링스트림으로 단점을 보완할 수 있다(패킷종류에 따라 분산시킨다).	재전송 및 인증과정으로 상당한 대역폭이 요구된다. 네트워크가 바쁘면 패킷도달이 지연될 수 있다. 지연된 패킷으로 인하여, 먼저 도착한 많은 패킷의 처리가 지연될 수 있다.
RELIABLE_SEQUENCED	RELIABLE의 특성에 최신 패킷만 취하는 형태이다.	패킷은 순서가 정해져서 보내진다 최근패킷이 먼저도착하면 그 이전 순서의 패킷을 기다릴 필요가 없다. 송신을 더 많이 분산시킬 수 있다. 가장 나중에 보낸 패킷은 반드시 도착한다	결국에는 무시될 신뢰층의 UDP 패킷들이 최근패킷이 도착한 것을 확인하기 위한 오버헤드로 인하여 대역폭낭비가 심하다.

```
}
```

RELIABLE과 RELIABLE_ORDERED는 대부분의 상황에 적합하다. 더 자세한 내용은 PacketPriority.h에서 참조할 수 있다.

다섯번째 파라메터는 어느 Ordering-스트림을 사용할지를 나타낸다. 이것은 같은 스트림상의 다른 패킷과 연관에서 상대적으로 패킷의 순서를 정하기위해 쓰인다. 지금 시점에서는 별로 중요하지 않다. 다만, 더 많은 정보를 보려면 **패킷송신**부분을 참조하면 된다.

서버와 peer에만 해당하는 사항

여섯번째 파라메터는 playerID이다. 이 값의 의미는 둘 중 하나다. 즉, **패킷을 보낼 대상**이거나 **패킷을 보내지 않을 대상**이다. 이것의 결정은 마지막 파라메터에 따라 결정된다.
일곱번째 파라메터는 접속된 모든 클라이언트에게 브로드캐스팅(방송)할지 안할지를 나타낸다. 이 파라메터는 여섯번째인 playerID와 같이 동작한다. 만약, 브로드캐스팅이 true이면, playerID는 이 중 제외될 대상을 나타내게 된다. 반면 브로드캐스팅이 false이면, playerID는 보낼 대상을 나타내는 것이다.

모든 클라이언트에 보내려면, playerID는 범위 밖이라는 뜻으로 미리정의된 UNASSIGNED_PLAYER_ID(IP:255.255.255.255 PORT:65535)로 지정하기만 하면 된다. 이것은 패킷을 중계하는데 매우 잘 작동한다. 왜냐하면 ~~packetID~~ playerID필드는 송신자가 누구인지를 나타내기 때문이다. 송신자를 Send함수의 playerID필드로 넘기고 브로드캐스팅을 true로 설정함으로써, 패킷을 송신자를 제외하고 - 보통 브로드캐스팅의 대상은 송신자는 제외되는 센스 - 모든 대상에게 방송할 수 있다.

~~여덟번째 파라메터는 보안을 위한 추가패킷을 제공하기위해 양쪽 시스템상에서 20bytes의 추가적인 다운스트림을 쓰고 추가적인 CPU사이클을 사용할지를 결정한다.~~
~~패킷보안은 중요하다. 왜냐하면 보안이 없다면 악의적인 유저가 게임을 방해하며, 서버에 폭주하기위해 패킷을 수정할 수도 있거나 아예 엉망으로 만들어버릴 수도 있기 때문이다.~~
~~*RakNet*은 이 파라메터가 false로 되더라도 자동으로 기본적인 보안을 제공한다. 그러나 매우 중요한 데이터를 위해 이 값을 true로 설정하는 것은 최상의 보안제공이 될 것이다.~~

셧다운
셧다운은 쉽고 거의 순식간에 이루어진다. 단순히 Disconnect만 호출하면 된다.

```
For our server:                         // peer와 동일
server->Disconnect(0);

For our client:
client->Disconnect(0);
```

서버나 클라이언트를 정지시키고, 동기화된 데이터도 정지시키고, 모든 내부데이터를 리셋한다.
이것은 네트워크쓰레드를 정지시키므로 *RakNet*은 불필요한 CPU사이클을 소모하지 않게 된다.
이 후, Start나 Connect메소드로써 언제든지 재시작을 할 수 있다.

디스커넥션메시지(ID_DISCONNECTION_NOTIFICATION)의 통지
서버를 닫으면 클라이언트에서는 데이터의 수신이 중지될 것이고 타임아웃에 의해 최종적으로 드랍될 것이다.
서버의 디스커넥션 이후의 Kick(서버인터페이스에만 존재)은 더 이상 아무의미도 없게 된다. 따라서 깔끔하게 처리하고 싶으면, 먼저 Kick()으로 모든 클라이언트를 드랍시켜서 클라이언트에서 디스커넥션메시지를 받게 할 수 있다.
또한, 남은 모든 패킷을 전송하도록 대기하는 블락시간(Disconnect의 인자)을 0보다 높게 주면, 디스커넥션메시지가 클라이언트에 도달할 가능성이 있다. 블락시간이 0인 경우에는 클라이언트로 디스커넥션메시지가 도달할 확률은 거의 없다. 이 블락시간의 설정은 클라이언트의 경우도 마찬가지로 된다.

개체제거
단순히 인스턴스를 팩토리의 DestroyRakClientInterface나 DestroyRakServerInterface중에서 적절한 곳에 넘기면 된다.

```
For our server:
RakNetworkFactroy::DestroyRakServerInterface(server);

For our client:
RakNetworkFactroy::DestroyRakClientInterface(client);
```

방화벽과 NAT유저에 대한 주의점
방화벽(Firewall)은 오직 권한있는 데이터를 어느 한 컴퓨터에서 전체적인 네트워크까지로 전달하게 하려는 하드웨어나 소프트웨어 유틸리티이다. *RakNet*은 네트워킹API이기 때문에, 다른 API나 어플리케이션과 마찬가지로, 방화벽은 무차별적으로 *RakNet*을 막아버릴 수도 있다.
이러한 것을 피하기 위해서, 약간의 트릭이 있다. 한가지 방법은 HTTP포트인 80과 같은 잘 알려진 포트를 *RakNet*이 사용하는 것이다. 이 방법은 종종 서버나 까다로운 클라이언트를 위해 쓰인다. 또 다른 방법으로는 클라이언트에 0번의 포트번호를 사용하게 한다. 이렇게 하면 자동으로 열린포트를 알아서 찾게 된다.

NAT이하 계층으로부터 제공되는 마스터서버를 사용하는 경우도 있다. 게임서버가 마스터서버로 연결이 완료되면, 그것은 어떠한 브로드캐스팅 IP나 포트상의 접속요청도 받아들일 수 있다(이것은 오직 UDP에만 해당한다).

PlayerID
Player ID란

Player ID란 무엇인가?
PlayerID는 네트워크상의 binary IP-주소(32비트)와 시스템-port(16비트)로 구성된 구조체(48비트)이다.

Binary IP Address
바이너리 IP주소는 32비트 메모리를 8비트씩 쪼개서 IP Address를 표현하는 것이다.

8비트	8비트	8비트	8비트
00000000	00000000	00000000	00000000
0~255	0~255	0~255	0~255

PlayerID가 필요한 경우를 보자:
- 서버가 특정한 하나의 클라이언트로부터 메시지를 받아 모든 다른 클라이언트에 중계하고 싶을 때 => *브로드캐스팅*으로 지정한 Send함수의 playerID인자를 송신자(메시지를 보낸 클라이언트)의 PlayerID(수신한 Packet::PlayerID필드값)로 기술할 수 있다.
- 게임상에서 지뢰와 같은 어떤 아이템은 특정 한 플레이어에게 속하고 킬에 대해서 적절한 사람에게 점수를 주기를 원할 때.
- 모든 클라이언트가 아는 각 플레이어에 대한 어떤 종류의 1:1맵핑을 싶을 때. 예를 들면, 각 플레이어가 고유한 점수를 갖고 있는 경우,
- P2P 네트워크상에서 특정 peer에게 메시지를 보내고 싶을 때.

고려할 사항
1. 패킷의 수신자는 패킷을 보낸 어떠한 시스템의 PlayerID라도 자동적으로 알 수 있다. 왜냐하면 PlayerID는 송신자의 IP/Port에서 결정되기 때문이다. 단지 서버가 송신자의 PlayerID만 알아야 한다면, 송신자는 자신의 PlayerID를 데이터에 따로 인코딩할 필요는 없다(Packet::playerID 참조).
2. 클라이언트/서버 모델인 경우, 클라이언트는 처음 패킷을 보낸 원래의 PlayerID를 **모른다**. 클라이언트의 입장에서는 모든 패킷은 서버에 의한 발생이다. 그래서 클라이언트가 다른 클라이언트의 PlayerID를 알아야 한다면, 데이터에 PlayerID를 포함시켜야 한다. 원래의 송신자로부터의 패킷을 받을 때, 송신측 클라이언트는 이 필드(데이터내의 PlayerID필드)에 자신의 PlayerID를 채워넣거나 서버에서 원래 발생한 패킷은 서버의 PlayerID를 이 필드에 기술해 넣는다.
3. PlayerID는 순차적으로 할당되거나 어떤 범위안에서 할당되지 않는다. 고급유저는 이것을 알 것이다.

타임스탬핑
Timestamping

같은 시점의 프레임상에서 다른 컴퓨터에 발생한 이벤트를 참조하는 방법
Timestamp(발송시간의 기록)는 단지 로컬시스템시간일 뿐이다. 불행하게도, 모든 시스템은 서로 다른 로컬시스템시간(디폴트 ms단위)을 가진다.
만약 네트워크상에서 단순히 로컬시스템시간을 전송한다면, 수신자는 다른 기계의 시간을 받는 것이다. 즉, 수신자는 오로지 자신의 시스템의 시간만 알기 때문에 송신자의 이벤트가 언제 일어날 지를 모르는 것이다.
*RakNet*의 타임스탬핑 특징은 수신한 시간을 자신의 로컬시스템시간으로 읽을 수 있게 하여, 다른 시스템의 로컬시간을 신경쓰지 않고 게임에만 집중하게 할 수 있다. 이것은 투명하면서도 자동적이다. 그리고 유동적인 핑(ping)에도 불구하고 매우 높은 정확도를 제공한다.

로컬시스템시간이 2000인 클라이언트에 이벤트가 발생한다고 하자(이 때, 서버의 로컬시스템시간이 12000이며, 다른 클라이언트의 로컬시스템시간이 8000이다).
만약 패킷에 조절된 시간이 기록되지 않았다면, 서버는 시간 2000을 받을 것이다 즉, 서버입장에서는 실제 수신한 *ping*/2의 시간(대략 100ms정도)에서부터의 10000ms의 전인 것이다.
비슷하게, 다른 클라이언트에서도 2000을 받고 그것은 이 클라이언트의 로컬시간인 8000으로부터의 6000ms전인 것이다.

다행히, *RakNet*은 이 문제에 대해 양쪽 시스템의 시간과 ping을 보정하여 다루는 방법을 제공한다. 상대시간을 사용하면, 서버는 대략 *ping*/2 ms전에 발생한 이벤트를 알 수 있고, 다른 클라이언트 역시 약 *ping*/2 ms전에 발생한 이벤트를 알 수 있는 것이다. 요약하여 말하자면, 패킷을 정확하게 인코딩하고 싶으면 타임스탬프를 사용할 수 있고, 그렇지 않으면 신경쓰지 않아도 된다.

주의
고해상도타이머로써 시스템시간을 얻으려면 GetTime클래스를 포함하는 것을 추천한다. 윈도우함수인 timeGetTime()을 써도 되지만 이것은 정확도가 낮다. 또한 타임스탬핑은 자동적인 핑에 의존하므로 StartOccasionalPing함수를 호출할 필요가 있을 것이다.
송신자가 보내는 ID_TIMESTAMP의 패킷을 수신자가 받으면 송신자의 로컬시간은 수신자의 로컬시간으로 자동으로 변환된 시간이 패킷안에 포함되어 있다

자동핑테스트
타임스탬핑은 핑에 의존하므로 항상 핑테스트를 해야 한다.
```
server->StartOccasionalPing();
client->StartOccasionalPing();
```

```
peer->SetOccasionalPing(true);
```

이 세가지 함수는 각자의 인터페이스에서 자동적으로 핑을 테스트하게 한다. 핑테스트는 대략 5초마다 이루어진다.
자동핑은 디폴트로 설정되지 않으며, 최초의 핑테스트는 설정하지 않아도 한번은 자동으로 이루어진다.

```
server->StopOccasionalPing();
client->StopOccasionalPing();
peer->SetOccasionalPing(false);
```

이것들은 자동핑테스트를 중지하고자 할 때 사용한다.

비트스트림

개요

설명

BitStream 클래스는 네임스페이스 *RakNet* 하에 있는 헬퍼클래스로써 비트열(bits)의 패킹과 언패킹을 위하여 **동적인 배열을 포장하는데 사용된다**. 이것의 세가지 이점은 :

1. 패킷을 동적으로 생성할 수 있다
2. 압축이 가능하다
3. 비트단위로 write할 수 있다

구조체를 사용하면, 구조체타입을 먼저 정의해야하고 또한 그것을 (char*)로 캐스팅하여 Send해야 한다. 비트스트림은 런타임에 데이터내용에 따라서 블록(여러바이트묶음)단위로 쓸 수 있다. 비트스트림은 내장형 타입(int, float와 같은)에 대해 압축할 수 있다.

압축은 매우 간단하며 이하의 알고리즘을 사용한다.

1. 상위절반의 모든 비트가 0인가(unsigned 타입인 경우)
   A. 그렇다 - 수치 1 을 Write한다
   B. 아니다 - 수치 0 과 상위절반을 Write한다
2. 하위절반에서도 반복한다. 이것은 데이터가 4비트가 남을 때까지 한다.

이것이 뜻하는 것은 데이터가 최대반이하의 범위이면 비트를 절약할 수 있다는 것이다. 그래서 이것이 진보된 케이스라는 것을 안다면, Write대신 WriteCompressed를 Read대신 ReadCompressed를 사용할 수 있다.

결국, 비트열(bits)을 쓸 수 있다는 것이다. 프로그래밍의 대부분의 시간동안 이 부분에 대해서 신경쓸 필요는 없다. 하지만, boolean값을 쓰면 자동으로 1비트만 사용할 것이다. 또한 이것은 암호화에도 유용하다. 왜냐하면 데이터는 더 이상 바이트로 정렬되지 않기 때문이다(boolean이 1비트만 소모하므로).

데이터 쓰기(Writing Data)

BitStream을 생성하라, 그리고 각 데이터타입에 맞게 Write메소드를 호출하라. 만약 데이터가 내장형이라면, 정확히 오버로드된 버전으로 호출될 것이다. 만약 데이터타입이 사용자정의형이라면, 우선 캐스팅해야 한다.

사용자정의형일 경우

```
bitstream.Write((char*)&myType, sizeof(myType))
```

참고 - 어떤 생성자는 파라메터로 바이트단위의 길이를 취한다. 만약, 전송할 데이터의 크기를 알고 있다면 BitStream을 생성할 때 이 크기를 지정하여, 생성 후 내부적인 재할당을 피하게 할 수 있다.

데이터 읽기(Reading Data)

데이터를 읽는 것은 역시 간단하다. 데이터타입에 맞는 생성자를 골라서 BitStream을 생성한다.

```
// Packet* 가 존재한다고 가정한다
BitStream myBitStream(packet->data, packet ->length, false);
```

주요 함수

생성자

```
BitStream(int initialBytesToAllocate);
```

초기에 얼마만큼의 바이트를 할당할지를 결정하기위한 생성자버전이다. 필요하지는 않더라도 이 방식으로 Write를 할 때, 재할당을 피할 수 있다.

생성자
BitStream(const char* _data, unsigned int lengthInBytes, bool _copyData);
이 생성자버전은 비트스트림에 초기데이터(_data)를 넣는다. 이것은 이미 존재하는 BitStream인 데이터 스트림을 해석하기위해 사용하며, 이 데이터스트림은 전송한 비트스트림을 받을 때 거의 항상 하는 과정이다. 순수하게 데이터를 읽기으려면 _copyData 에는 false를 준다(이 경우 생성된 BitStream은 _data를 단순히 가리킬 뿐이다). 만약, 내부적인 복사나 데이터의 저장 및 차후 변경여지가 있다면 _copyData 를 true로 준다(BitStream은 _data의 영역을 자기로 복사한다).

Write 함수군
이 쓰기함수는 비트스트림의 끝부분에 데이터를 쓴다. 데이터를 읽기 위해서는 Write와 상응하는 Read를 사용해야 한다

WriteCompressed 함수군
이 쓰기함수는 비트스트림의 끝부분에 데이터를 쓴다. 데이터를 읽기 위해서는 WriteCompressed와 상응하는 ReadCompressed를 사용해야 한다.

Read 함수군
이 읽기함수는 비트스트림에 이미 존재하는 데이터를 시작부분에서 끝으로 순서대로 읽는다. 비트스트림에 더 이상의 데이터가 없다면 이 함수는 false를 반환한다.

ReadCompressed 함수군
이 읽기 함수는 WriteCompressed를 사용하여 쓰여진 비트스트림의 이미 존재하는 데이터를 시작부분에서 끝으로 순서대로 읽는다. 비트스트림에 더 이상의 데이터가 없다면 이 함수는 false를 반환한다.

GetNumberOfBitsUsed
GetNumberOfBytesUsed
쓰여진 비트수 또는 바이트수를 반환한다

GetData
비트스트림의 내부데이터의 포인터를 반환한다. 이것은 malloc를 가지고 (char*)으로 할당된 것으로 데이터에 대해 직접접근이 필요한 경우를 위해 제공된다.

비트스트림은 비트스트림의 읽기/쓰기방식은 일반적으로 파일을 다룰 때의 읽기/쓰기 방식과 비슷하고 또한 파일포인터의 개념이 있다. 비트스트림은 간단히 하나의 파일스트림처럼 인식할 수 있다.

패킷 생성
패킷 생성

데이터를 선택하라
나만의 새로운 패킷의 타입을 생성하는 것은 원하는 데이터가 무엇이며, 그것을 전송하기위한 최상의 방법을 결정하는 문제이다. 여기 그 방법이 있다.

네트워크상에서 전송할 데이터를 결정하라. 예를 들면, 게임상에서 시한의 지뢰의 위치를 정해보자. 우리는 이하의 데이터가 필요할 것이다.

- 지뢰의 위치(float x,y,z). 아니면 3D Vector타입으로 대신할 수도 있다.
- 모든 시스템에서 동의하는 지뢰를 참조할 수 있는 방법. NetworkObject 클래스는 이것에 대해 완벽하게 작동한다. NetworkObject로부터 Mine 클래스를 상속했다고 가정하자. 그러면 보관해야할 것들은 그 지뢰에 대한 ObjectID를 얻는 것 뿐이다.
- 지뢰를 소유한 자. 누군가 지뢰를 설치하면 그것이 누구 것인지 알아야 한다. 데이터에 그 players 또는 playerID의 참조가 내장되어 있으면 완벽하다. 만약에 서버가 설치한다면, 지뢰의 playerID는 더미값(255와 같은 것)을 사용할 수 있다. 클라이언트가 게임한다면 지뢰의 playerID는 그 자신의 GetPlayerID를 사용하여 채울 수 있다.
- 지뢰를 설치한 시간. 10초후에 지뢰가 자동으로 터진다고 하자. 그러면 다른 컴퓨터들에서 서로 다른 시간에 지뢰가 터지지 않도록 하기 위해 정확한 시간을 얻어내는 것이 중요해진다. 다행히도 *RakNet*은 이러한 것을 Timestamping를 사용하여 다룰 수 있는 능력에 기반하고 있다.

구조체냐 비트스트림이냐

어차피, 데이터를 전송할 때는 캐릭터(문자, 1Byte단위)의 스트림을 전송할 것이다. 이 스트림으로 데이터의 인코딩은 두 가지 쉬운 방법이 있다.
한 가지는 구조체를 만들어서 그것을 (char*)로 캐스팅하여 쓰는 것이고, 다른 하나는 BitStream 클래스에 기반을 둔 방식이다.

구조체의 장점
- 구조체를 바꾸기가 매우 쉽고 실제 전송할 데이터가 무엇인지 알기도 쉽다.
- 송신자와 수신자 모두는 이 구조체를 정의한 파일을 같이 사용할 수 있어서, 수신측의 캐스팅에 따른 실수를 피할 수 있다.
- 데이터를 수신하는데 순서가 바뀔 염려도 없고 잘못된 타입으로 쓰는 경우도 없다.

구조체의 단점
- 많은 파일을 자주 변경해야하고 재컴파일을 해야한다(데이터 타입마다 구조체가 존재하므로).
- BitStream 클래스에서 자동으로 처리되는 압축을 할 수 없다.

비트스트림의 장점
- 비트스트림을 쓰기위해서 다른 파일의 수정이 필요없다. 단순히 BitStream을 생성하고, 원하는 순서대로 원하는 데이터를 쓰고 전송하면 된다.
- 적은 양의 비트를 사용한 Write/Read 메소드의 압축버전을 사용할 수 있다. 또한 bool타입은 단지 1비트만 사용한다.
- 동적으로 데이터를 쓸 수 있다. 즉, 특정 조건의 참, 거짓의 여부에 따라 특정한 값을 쓸 수 있다.

비트스트림의 단점
- 실수의 우려가 높다. 데이터를 쓴 방식과 다르게 읽을 수가 있다(잘못된 순서, 잘못된 타입 및 다른 실수 등).

구조체로 패킷 만들기

NetworkStructures.h 를 보자(이 헤더는 라이브러리에 포함되지 않고 응용프로그램에서 만들어 쓰는 것 같다)

이하와 같은 중앙에 커다란 구역이 있어야 한다:
```
// --------------------------
// YOUR STRUCTURES BELOW HERE!
// --------------------------

// --------------------------
// YOUR STRUCTURES HERE!
// --------------------------
            여기에다만 구조체를 정의하라는 것인가?

// --------------------------
// YOUR STRUCTURES ABOVE HERE!
// --------------------------
```

어디에 구조체를 위치해야할 지는 아주 명확해야 한다.

구조체를 사용하기 위한 형태는 일반적으로 두 가지가 있는데, 타임스탬핑을 포함한 것과 그렇지 않은 것이다.

타임스탬핑을 포함하지 않는 구조체
```
#pragma pack(1)
struct structName
{
            unsigned char typeId;                    // 사용자정의타입의 ID
            // 실제 데이터
};
```

타임스탬핑을 포함하는 구조체
```
#pragma pack(1)
struct structName
{
            unsigned char useTimeStamp;          // 이 부분은 ID_TIMESTAMP
            unsigned long timeStamp;               // timeGetTime()나 비슷한 기능을 하는 다른 메소드가 반환한 시스템시간을 넣는다
            unsigned char typeId;                    // 사용자정의타입의 ID
            // 실제 데이터
};
```

=> 구조체의 첫번째 바이트는 항상 패킷의 ID이다

이제 패킷을 채우자. 시한의 지뢰를 위해, 타임스탬핑을 사용하는 형태이어야 한다. 그래서 결과는 이하와 같다:
```
#pragma pack(1)
struct structName
```

```
{
	unsigned char useTimeStamp;		// ID_TIMESTAMP으로 정한다
	unsigned long timeStamp;		// getTime()으로 시스템시간을 넣는다
	unsigned char typeId;			// PacketEnumerations.h에서 추가된 타입 ID이다. ID_SET_TIMED_MINE으로 정의했
다고 하자
	float x,y,z;				// 지뢰의 위치
	ObjectID objectID;			// 지뢰의 ObjectID. 서로다른 컴퓨터에서 이 지뢰에 대한 공통적인 참조 방식으로 사
용된다
	PlayerID playerID;			// 이 지뢰를 소유한 플레이어
};
```

위에 코멘트 했듯이, typeId는 그 열거리스트에 추가되어야 수신콜에서 데이터스트림이 도착했을 때 이 패킷이 무엇인지 판단할 수 있다. 그래서 PacketEnumeration.h의 마지막에 ID_SET_TMED_MINE(또는 정하고 싶은 아무 것이나)를 추가한다.

ObjectID
이것은 NetworkTypes.h에 unsigned short로 재정의된 타입으로 서로 연결된 네트워크상에서 전역적으로 식별가능한 각 오브젝트의 고유한 ID를 나타낸다.
PlayerID와 혼동하지 말 것.

주의사항 - 구조체내에 포인터가 직접적으로나 간접적으로도 절대로 포함되어서는 안된다
이것은 구조체에 포인터나 객체포인터를 포함시키는 사람들에게서 흔히 발생하는 실수로써 포인터가 가리키는 데이터가 네트워크상에서 전달될 것이라는 생각에서 비롯된다.
그러나 그러한 것은 적용될 수 없다. 단지 포인터 주소만 전달될 것이다.

유용한 코멘트
여기에서 ObjectID를 objectID로, PlayerID를 playerId로 했는지 봤을 것이다. 왜 더욱 묘사적인 mineId나 mineOwnerId와 같은 이름을 사용하지 않았을까? 이러한 특정한 상황에서 묘사적인 이름을 사용하는 것은 어떠한 방법으로나 이득이 되지 않는다. 왜냐하면 패킷타입을 결정할 때까지는 이 변수들이 문맥상 무엇을 뜻하는지를 알 수 있고 그것 이외에는 아무것도 뜻하지 않는다. 일반적인 이름을 사용하는 이득은 코드를 지루함이 없게하고 이름을 재지정하는 일 없이, 빠르게 패킷을 다루기 위해 코드를 자르고 붙일 수 있는 것이다. 큰 게임과 같이 많은 패킷이 있다면, 이는 상당한 혼란을 줄여줄 것이다.

중첩된 구조체
중첩된 구조체를 사용하는데는 아무런 문제가 없다. 단지 첫번째 바이트는 언제나 패킷의 타입을 결정한다는 것만 알아두면 된다.

```
#pragma pack(1)
struct A
{
	unsigned char typeId;		// ID_A
};

#pragma pack(1)
struct B
{
	unsigned char typeId;		// ID_A
};

#pragma pack(1)
struct C		// Struct C는 ID_A 타입이다
{
	A a;
	B b;
}

#pragma pack(1)
struct D		// Struct D는 ID_B 타입이다
{
	B b;
	A a;
}
```

비트스트림으로 패킷 만들기

데이터를 적게 쓰자
위의 지뢰예시를 가지고 비트스트림으로 써보자. 데이터는 이전과 모두 같다.

```
unsigned char useTimeStamp;		// ID_TIMESTAMP으로 정한다
unsigned long timeStamp;		// getTime()으로 시스템시간을 넣는다
unsigned char typeId;			// PacketEnumerations.h에서 추가된 타입 ID이다. ID_SET_TIMED_MINE으로 정의했다고 하자
```

```
useTimeStamp = ID_TIMESTAMP;
timeStamp = getTime();
typeId = ID_SET_TIMED_MINE;

Bitstream myBitStream;
myBitStream.Write(useTimeStamp);
myBitStream.Write(timeStamp);
myBitStream.Write(typeId);
// Mine* mine 을 했다고 가정한다
myBitStream.Write(mine->GetPosition().x);
myBitStream.Write(mine->GetPosition().y);
myBitStream.Write(mine->GetPosition().z);
myBitStream.Write(mine->GetID());                    // 구조체에서는 ObjectID objectId 에 해당한다
myBitStream.Write(mine->GetOwner());              // 구조체에서는 PlayerID playerId 에 해당한다
```

만약 myBitStream을 RakClient::Send나 RakServer::Send로 보내려한다면, 이 시점에서는 내부적으로 구조체를 캐스팅한 것과 동일한 것으로 볼 수 있다.
이제 조금 더 향상시켜보자. 이 시한지뢰가 어떠한 원인에 의해서 (0,0,0)의 위치에 있을 때의 좋은 처리방식이다. 이것은 이하와 같다:

```
unsigned char useTimeStamp;            // ID_TIMESTAMP으로 정한다
unsigned long timeStamp;                  // getTime()으로 시스템시간을 넣는다
unsigned char typeId;                       // PacketEnumerations.h에서 추가된 타입 ID이다. ID_SET_TIMED_MINE으로 정의했다고 하자

useTimeStamp = ID_TIMESTAMP;
timeStamp = getTime();
typeId = ID_SET_TIMED_MINE;

Bitstream myBitStream;
myBitStream.Write(useTimeStamp);
myBitStream.Write(timeStamp);
myBitStream.Write(typeId);
// Mine* mine 을 했다고 가정한다
// mine이 (0,0,0)에 있으면, 이것을 단지 1비트로 나타낼 수 있다
if (mine->GetPosition().x==0.0f && mine->GetPosition().y==0.0f && mine->GetPosition().z==0.0f)
{
        myBitStream.Write(true);
}
else
{
        myBitStream.Write(false);
        myBitStream.Write(mine->GetPosition().x);
        myBitStream.Write(mine->GetPosition().y);
        myBitStream.Write(mine->GetPosition().z);
}
myBitStream.Write(mine->GetID());                    // 구조체에서는 ObjectID objectId 에 해당한다
myBitStream.Write(mine->GetOwner());              // 구조체에서는 PlayerID playerId 에 해당한다
```

이것은 잠재적으로 1비트 비용으로 네트워크상에서 3개의 float를 보내는 것을 절약할 수 있다.

문자열을 쓰자

BitStream을 오버로드하여 배열을 사용하여 문자열을 쓰는 것은 가능하다. 한가지 방식은 문자열의 길이를 쓰는 것으로써 이하와 같을 것이다:

```
void WriteStringToBitStream(char *myString, BitStream *output)
{
        output->Write((unsigned short) strlen(myString));
        output->Write(myString, strlen(myString);
}
```

디코딩 역시 비슷하다. 그러나, 매우 비효율적이다. *RakNet*은 StringCompressor라고 부르는 stringCompressor를 갖추고 있다. 이것은 전역인스턴스다. 이것으로 WriteStringToBitStream 은 이하와 같이 될 수 있다:

```
void WriteStringToBitStream(char *myString, BitStream *output)
{
        stringCompressor->EncodeString(myString, 256, output);
```

```
}
```

이것은 문자열을 인코딩하여 패킷을 훔쳐보려는 자에 의해 쉽게 읽히지 않게 할 뿐만 아니라, 압축도 가능하다. 문자열을 디코딩하기 위해서는 이하와 같이 사용하면 된다:

```
void WriteStringToBitStream(char *myString, BitStream *output)
{
        stringCompressor->DecodeString(myString, 256, output);
}
```

이 예에서의 256은 읽기와 쓰기의 최대 바이트 수이다. EncodeString에서, 문자열의 길이가 256보다 작으면 모든 문자열이 쓰일 것이다. 만약 256문자보다 더 크면, NULL종료문자를 포함한 256문자를 가진 배열으로 디코딩시에 잘려나갈 수가 있다.

Programmer's note:
단순히 (char*)으로 캐스팅하여 BitStream으로 직접 구조체를 쓰는 것 역시 가능하다. BitStream은 memcpy를 사용하여 이 구조체를 자신으로 복사할 것이다.
구조체를 가지므로, 포인터를 가지면 안된다.

패킷 송신

패킷 송신

Step 1: 데이터를 결정하라

필요한 데이터가 무엇이고 구조체로 할지 비트스트림으로 할지 결정한다.

Step 2: 권한을 결정하라

일반적으로 액션에 의한 결과를 전송하기보다 액션에 대한 트리거(발생상황)를 전송하려한다. 엄밀히 말하자면, 패킷의 발생지는 이하 3개의 범주에 속한다:

- 액션을 행하는 함수로부터
- 액션을 행하는 함수에 대한 트리거로부터
- 데이터 모니터링으로부터

이러한 각 방식은 여러 장점과 단점이 있다

액션을 행하는 함수로부터의 패킷송신

예시:
ShootBullet(총알 발사하는 함수임)라고 하는 함수가 있다. 이 함수는 발사(shot), 발사원인(shot origin), 발사방향(shot direction)과 같은 다양한 인자를 가진다.
ShootBullet에 진입할 때마다 이 상황(총알 발사했다? 발사할 것이다?)을 네트워크상에 알리기 위해 패킷을 보내고 싶다.

장점:
유지보수가 쉽다. ShootBullet는 많은 다양한 원인(마우스 입력, 키보드 입력, AI 등)에 의해 호출될 수 있다. 그리고 패킷이 송신된 모든 발생지의 추적에 대해 따로 걱정할 필요가 없다(ShootBullet이 발생지). 또한 이미 만들어진 싱글게임에 장착하기 쉽다.

단점:
Hard to program. If I use ShootBullet to initiate the packet, then what does the network call when it wants to perform this function? If ShootBullet initiates the packet, and the network calls ShootBullet, then another packet would get sent, creating a feedback loop. So I can either write another function, like DoShootBullet (sloppy), or pass a parameter to ShootBullet telling it whether or not to send a packet.
프로그램하기 까다롭다. 만약 패킷의 초기화를 위해 ShootBullet을 사용한다면, 네트워크(여기서는 패킷송신하는 입장의 시스템)가 이 기능(패킷을 만들고 송신)을 실행하려면 무엇을 호출할까? 만약 ShootBullet이 패킷을 초기화하고, 네트워크가 ShootBullet를 호출한다면, 피드백루프가 형성(오리지날 소스는 ShootBullet를 총알 발사로 사용할 것이고 이것을 네트워크 전송 부분에서 또 한번 호출한다는 뜻)되므로 또 다른 패킷이 송신될 것이다. 그래서 조잡하게 DoShootBullet(패킷을 안보내고 총알만 발사하는)과 같은 또 다른 함수를 써야하거나, 패킷을 보낼지 말지를 결정하는 인자를 ShootBullet에 넘겨버릴 수 있다.

I also have to consider authority. Can the client shoot the bullet immediately, or does the client need authorization from the server first to shoot the bullet? If it needs authorization then ShootBullet should send a packet and then return immediately, unless it was called by the network in which case it should not send the packet but do the action instead. The network may also need additional data that ShootBullet doesn't have, such as the number of bullets remaining. Sometimes I can get this from the context, sometimes not. It takes some practice and experience to code in this style and even I make bugs sometimes doing it.
또한 권한에 대해 고려해야 한다. 클라이언트는 총알을 즉시 발사할 수 있을까? 아니면 클라이언트는 총알을 발사하기 위해 우선 서버로부터 권한을 얻어야 하는가?
패킷을 송신하는 대신 액션만 실행해야하는 경우 네트워크에 의해 호출되지 않는 상황에서, 권한이 필요하다면 ShootBullet은 패킷을 보내고나서 즉시 리턴해야 한다. 또한 네트워크는 ShootBullet가 가지지 않는 남은 총알의

개수와 같은 추가적인 데이터를 필요로할 수 있다. 때때로 문맥상으로 이것을 얻을 수 있고 그렇지 않을 수도 있다.
이러한 스타일의 코딩을 위해서는 좀 더 연습과 경험이 필요하고 필자조차도 간혹 버그를 만드는 경우도 있다.

액션을 행하는 함수에 대한 트리거로부터의 패킷송신

예시:
ShootBullet 예제를 다시 활용하자. 그러나 이번에는 ShootBullet안에서 패킷을 송신하는 대신, ShootBullet에 대한 트리거로부터 패킷을 송신할 것이다. 예를 들면, 유저가 마우스를 클릭하거나 AI가 슈팅을 결정하거나, 스페이스바를 눌렀을 경우다.

장점:
이번엔 피드백루프에 대한 걱정할 필요없이 네트워크로부터 ShootBullet 함수를 호출할 수 있다. 또한 일반적으로 더 많은 정보가 함수밖에 존재하므로 네트워크에서 필요로한다면 이 데이터를 보내는 것이 더욱 쉽다.

단점:
High maintenance. If I later add another way to shoot bullets I may forget to send a packet for it.
높은 수준의 유지관리가 필요하다. 이 후 총알을 발사하는 다른 방법(트리거)이 추가되면 그것에 대한 패킷을 송신하는 것을 잊을 수도 있다.

데이터 모니터링으로부터의 패킷송신

예시:
플레이어의 체력이 0으로 될때마다 패킷을 보내려한다. 그러나, 실제로 체력이 0으로 되는 지점에서 이것을 정확히 할 수 없다. 따라서 아마도 플레이어를 업데이트하는 코드에 매 프레임마다 실행하는 어떤 함수를 추가한다. 이 함수가 체력이 최초로 0이되는 것을 감지하면, 바로 패킷을 보낸다. 그러면 이 패킷은 보내지고 또 다시 보내지는 일은 없게 될 것이다.

장점:
네트워킹 관점에서보면 매우 명확하다. 피드백에 대한 걱정이나 액션실행을 하는 함수를 변경할 필요가 없다. 누군가 모니터링하는 것을 바꾸지만 않는다면 유지보수가 전혀 필요없다. 네트워크는 매초마다 한번이상 패킷을 보내지 않는 효율적인 알고리즘을 구현할 수 있다.

단점:
디자인관점에서 보면 조잡하다. 오직 특정한 데이터타입만 사용될 수 있다. 모니터링하는 오브젝트가 리셋되면 모니터링 코드를 리셋해야하는 추가적인 작업을 해야한다. 프로젝트상의 다른 프로그래머들이 모니터링하는 데이터의 작동방식을 변경하려할 때, 그들은 이점(데이터작동방식을 바꾸면 모니터링 방식도 바꿔야 된다)을 알아야 한다.

Step 3: 어떤 신뢰도타입이고 필요한 오더링스트림이 어느 것인지 결정하라

PacketPrioirty.h 는 이것들에 대한 열거상수를 포함하고 있다. 우선순위는 세가지가 될 수 있다: *HIGH_PRIORITY*, *MEDIUM_PRIORITY*, *LOW_PRIORITY*
*HIGH_PRIORITY*의 패킷은 *MEDIUM_PRIORITY*전에 송신되고, *MEDIUM_PRIORITY*은 *LOW_PRIORITY*전에 송신된다.
신뢰도타입(Reliability)은 세부구현에 소개되어 있다. 보통 *RELIABLE_ORDERED*를 게임상의 패킷으로 쓰길 원할 것이다. 모든 Ordered-타입에 대해서, 이하에 소개된 Ordering-스트림을 쓰길 원할 것이다.

Step 4: 서버나 클라이언트의 Send 메소드를 호출하라

Send 메소드는 데이터를 변경시키지 않고 복사할 것이다. 그래서 여기부터는 프로그래머의 관점에서 해야 한다.

Ordering streams

Ordered-패킷(순서를 지정한 패킷)을 위해 32가지의 Ordering-스트림(오더링 스트림)이 있고 이 32가지 Ordering-스트림은 Sequenced-패킷에 대해 사용할 수 있다.
그러면 상대적 Ordering-스트림으로써의 스트림을 생각할 수 있다. 이것은 같은 Ordering-타입의 모든 패킷은 서로 상대적으로 순서가 정해져 있는 것이다(링크드리스트처럼 다음노드를 지정할 수 있는 개념). 이를 나타낸 가장 쉬운 방식의 예제가 있다.

모든 채팅메시지를 Ordering하고, 모든 플레이어의 이동 패킷을 Ordering하고, 모든 플레이어의 발사패킷을 Ordering하고, 모든 남은 탄약패킷을 Sequencing한다고 가정하자.
채팅메시지는 순서대로 도착하기를 원할 것이다. 그러나 채팅메시지가 미뤄지지 않기를 원할 것이다. 왜냐하면 더 일찍 보내진 다른 플레이어의 이동패킷을 받을 수 없기 때문이다. 플레이어의 이동패킷은 채팅메시지와는 아무런 연관이 없다. 그래서 그것들이 도착하는 순서가 어찌되었건 관여할 필요가 없다.
그러므로 이것들(채팅메시지, 이동패킷)에 대해서는 서로 다른 Ordering-스트림을 사용하면 된다. 즉, 채팅메시지는 0이고 플레이어 이동패킷은 1이 될 수 있다.

그러나, 플레이어의 발사패킷은 이동패킷에 관련되어 순서가 정해져야 한다. 왜냐하면 분명 잘못된 위치에서 발사하는 것을 원하지 않을 것이기 때문이다. 그래서 플레이어의 발사패킷은 이동패킷과 같은 스트림상(스트림 1)에 놓여져야 하고, 만약 이동패킷이 실제 발사패킷보다 먼저 송신되었지만 발사패킷보다 나중에 도착하면, 이 발사패킷은 이동패킷이 도착할 때까지 주어지지 않을 것이다(대기상태).

순차적 패킷(sequence)은 오래된 패킷을 떨궈낸다(드랍), 그래서 만약 패킷 2, 1, 3을 순차적으로 받았다면, 최종결과는 2와 1은 드랍될 것이고, 패킷 3만 받을 것이다. 이것은 탄약에 대해 유용할 것이다. 왜냐하면 탄약은 오직 줄어들기만 하기 때문이다. 만약 오래된 패킷을 받는다면 가지고 있는 가지고 있는 탄약수가 증가할 것이다. 즉 잘못된 것이다.

Sequenced-패킷은 Ordered-패킷과 다른 스트림군에 속하기 때문에, 0과 같은 원하는 아무 스트림 수치를 사용할 수 있다. 단, 이것은 채팅메시지와는 아무런 관련이 없다. 왜냐하면 채팅메시지는 Ordered-스트림군을 사용하지 Sequenced-스트림군을 사용하지 않기 때문이다(Ordered인지 Sequenced인지는 Send함수의 신뢰도인자에서 결정되는 것). Ordering나 Sequencing이 전혀되지 않은 패킷, 즉 UNRELIABLE와 RELIABLE은 시퀀스와는 아무런 관련이 없다. 이러한 패킷의 타입에 대한 Send함수의 orderingChannel인자는 무시된다.

> Ordering Stream은 간단히 네트워크시스템의 수신받는 패킷의 통로라고 보면 된다. 즉, IP주소하에 포트가 있듯이 포트 이하에 Ordered와 Sequenced의 2개의 문이 있고 이것들 이하에 오더링스트림이라는 32개의 문이 각각 존재한다고 보면 된다.

패킷 수신

패킷 수신

패킷의 수신

네트워크상에 어떤 패킷이 들어오면, 즉 Receive 함수가 0이 아닌 수를 리턴하면, 그것을 다루는 세 가지 단계가 있다. Multiplayer 클래스는 첫번째와 세번째 단계를 제공하므로, 만약 이 클래스로부터 파생한다면 단지 ProcessUnhandledPacket과 public 함수를 재정의하여 두번째 단계로 직접 건너뛸 수 있다.

Step 1. 패킷의 타입을 결정하라. 이것은 이하의 코드에 의해 리턴된다:

```
unsigned char GetPacketIdentifier(Packet *p)
{
	if ((unsigned char)p->data[0] == ID_TIMESTAMP)
		return (unsigned char) p->data[sizeof(unsigned char) + sizeof(unsigned long)];
	else
		return (unsigned char) p->data[0];
}
```

구조체의 취득 - Step 2

만약 원래 구조체를 송신했다면, 그것을 이하의 방식으로 캐스팅하여 얻을 수 있다:

```
// 만약 Multiplayer 클래스를 오버라이드 했다면 이 라인은 ProcessUnhandledPacket안에서 나타날 수 있다
if (packetIdentifier==/* 이 부분은 사용자정의타입 ID가 될 수 있다*/)
	DoMyPacketHandler(packet);


// 이것은 원하는 어느 곳에 위치시킨다. 게임을 다루는 상태클래스안에서는 아주 좋은 위치가 될 수 있다
void DoMyPacketHandler(Packet *packet)
{
	// 데이터를 적절한 구조체타입으로 캐스팅한다
	MyStruct *s = (MyStruct *) packet->data;
	assert(p->length == sizeof(MyStruct));
	if (p->length != sizeof(MyStruct))
		return;

	// Network Object와 그안에 정의된 매크로를 사용하여, 구조체 안에 기술된 오브젝트에 대한 포인터를 얻는다
	//// Network Object자체가 구조체 안에 포함되어있는 것이 아니라 이 오브젝트의 ID를 구조체가 포함하고 있는 것.
	MyObject *object = (MyObject *)GET_OBJECT_FROM_ID(s.objectId);

	// 이 패킷의 타입에 대한 처리할 것을 실행한다
	object->DoFunction();
}
```

유용한 코멘트

- 만약 실제 구조체를 생성한다면 발생하는 복사오버헤드를 피하기 위해 **패킷의 데이터를 적절한 구조체타입의 포인터로 캐스팅한다**. 그러나, 이 경우 만약 그 구조체안의 어떤 데이터를 변경시킨다면 그 패킷 또한 변경될 것이다. 이것은 원하던 것이 될 수도 아닐 수도 있다. 서버입장에서의 응답메시지이면 신중해야 한다. 이것은 의도하지 않은 버그를 발생시킬 수 있기 때문이다.
- 만약 패킷을 전송할 때 잘못된 ID나 잘못된 크기를 할당했다면, 비록 필요치 않더라도 assert는 찾기 힘든 버그를 잡아내는데 매우 유용하다

- if 문장은 누군가 서버나 클라이언트를 공격하기 위해 무효한 크기나 타입의 패킷을 보내려하는 경우에 유용하다. 이것은 연습상황에서는 절대로 발생하지 않는 일이지만, 안전에 대한 지장은 없다.

Step 3. 네트워크인터페이스의 virtual void DeallocatePacket(Packet *packet)=0; 메소드로 받은 패킷을 보내어 할당해제시켜라

비트스트림의 취득 - Step 2

만약 원래 비트스트림을 보냈다면, 그것을 썼던 순서대로 데이터를 조립하기 위해 하나의 비트스트림을 생성한다. 비트스트림은 패킷의 데이터와 크기를 사용하여 생성한다.
그리고나서 Read 함수를 Write 함수를 사용한 형태에 맞게 사용하며, ReadCompressed 함수는 WriteCompressed를 사용한 형식에 맞게 사용하고, 만약 조건에 따른 어떤 데이터를 썼다면 동일한 논리적인 분기처리에 따라야 한다. 이것은 이하의 예제에서 모두 나타난다. 이 예제는 위에서 보였던 지뢰예제에 대한 데이터를 읽는 것이다.

```
void DoMyPacketHandler(Packet *packet)
{
	Bitstream myBitStream(packet->data, packet->length, false); // false는 받은 데이터를 복사할 것이 아니기 때문에 효과적이다
	myBitStream.Read(useTimeStamp);
	myBitStream.Read(timeStamp);
	myBitStream.Read(typeId);
	bool isAtZero;
	myBitStream.Read(isAtZero);
	if (isAtZero==false)
	{
		x=0.0f;
		y=0.0f;
		z=0.0f;
	}
	else
	{
		myBitStream.Read(x);
		myBitStream.Read(y);
		myBitStream.Read(z);
	}
	myBitStream.Read(objectID);		// 구조체를 이용하는 경우는 ObjectID objectId 부분과 같다
	myBitStream.Read(playerId);		// 구조체를 이용하는 경우는 PlayerID playerId 부분과 같다
}
```

원격과 로컬
원격(Remote)는 현재시스템입장에서 다른 외부의 연결된 시스템이고 로컬(Local)은 자기 자신이다.
클라이언트입장에서는 서버가 원격시스템이고 서버입장에서는 클라이언트가 원격시스템이다.
PEER간에서도 마찬가지다.

The Network Object

Network Object

Network Object 클래스는 다른 컴퓨터상의 오브젝트를 공통적인 방식으로 참조할 수 있게 한다

Network Object 클래스는 그것으로부터 상속하여 쓸 수 있는 선택적인 클래스이며, 상속한 오브젝트에는 자동으로 식별할 수 있는 번호가 할당된다. 이는 멀티플레이 게임에서 매우 유용하다. 왜냐하면 만약 그러하지 않다면(식별번호가 없으면) 다른 원격시스템 상의 **동적인** 오브젝트를 참조할 길이 없기 때문이다.

중요사항:
Network Object가 작동하기 위해서는 서버가 동작중인지 아니면 클라이언트가 동작중인지 아니면 둘다 동작중인지를 알아야 한다. 외부에 인스턴스화된 서버와 클라이언트가 있고 그것들은 Multiplayer class를 사용한다고 하자.

Network Object는 **반드시** 서버나 클라이언트에 대한 엑세스를 할 수 있어야 한다. 사용자는 자신만의 인스턴스를 쓸 수 있지만, 그러면 작동시키기 위해서는 Network Object를 수정할 필요가 있다.

가장 간단한 예로, 어떻게 작동하는지 보자:
오브젝트의 ID를 얻기 위해 GetID()를 사용한다. 만약 ID가 할당되지 않았다면,
UNASSIGNED_OBJECT_ID(63335)를 반환할 것이다.
오브젝트의 ID를 설정하기 위해 SetID()를 사용한다.

이것을 잘못쓴다면 트러블에 빠지기 쉬우므로, Network Object에 printf와 puts으로 코멘트를 붙인 것이다. 디버그에서, 그것들을 언코멘트할 수 있고 자신의 메시지핸들러를 사용해서 어떤 무엇을 잘못했다면 경고와 에러를 받을 수 있다.

1. 새로운 ID를 재할당하려는 것이 아니면, 이미 ID가 할당된 오브젝트에 SetID를 호출하지 마라
2. 만약 어느 한 시스템상의 오브젝트를 지운다면, 이것은 그 오브젝트의 ID를 무효화해서 모든 시스템상의 그 오브젝트를 지울 필요가 있다
3. 랙으로 인하여 어떤 오브젝트는 한 오브젝트에는 존재하지만 다른 곳에는 존재하지 않을 수도 있다. 만약 objectID를 포인터로 변환하기 위해 GET_OBJECT_FROM_ID를 사용한다면, 반드시 0을 반환하는지 체크해야 한다.

서버에서:
ID는 자동적으로 할당되므로 GetID()는 항상 작동하고 SetID()는 호출하지 않는다. 서버가 새로운 Network Object를 생성하면, 클라이언트는 반드시 그것을 알아야 한다. 그래서 일반적으로 이하와 같이 한다:

```
MyObject *myObject = new MyObject;          // MyObject는 Network Object로
부터 상속했다
ObjectID objectId = myObject->GetID();      // ObjectID는 unsigned short의
타입재정의
```

ObjectID필드를 가진 패킷을 생성하고 클라이언트로 보낸다.

클라이언트가 패킷을 받으면:

```
MyObject * myObject = new MyObject;         // MyObject는 Network Object
로부터 상속했다
myObject ->SetID (objectId);                    // objectID는 이미 패킷에
서 포함된 것이다
```

클라이언트에서:
ID는 절대로 할당되지 않고, 서버로부터 받아야 한다. 만약 오브젝트를 생성하거나 생성하기를 원한다면, 방금 생성한 오브젝트에 어떤 아이디가 적합한지를 나타내는 패킷을 서버가 보낼 수 있는 방식으로 프로그램을 만들어야 한다. 만약 서버가 오브젝트를 생성한다면(또는 다른 클라이언트가 생성하고 서버가 그것을 알려준다면), 단지 평소와 같이 ID를 할당할 수 있다.

클라이언트상에서 오브젝트를 생성하는 가장 쉬운 방법은 서버에게 그 오브젝트를 물어보는 것이고 오직 서버가 응답하면 오브젝트를 만드는 것이다. ID_REQUEST_CREATE_OBJECT(여기서는 사용자정의메시지임)같은 그 무엇을 보내라. 그러면 오브젝트를 생성하고 그 패킷에 인코딩된 ID를 포함한 이 패킷의 송신자에게 응답하는 서버를 프로그램할 수 있다(단지 위에서 했던것과 같다) .

서버상에서는:

```
// ID_REQUEST_CREATE_OBJECT에 대한 핸들러부분
MyObject * myObject = new MyObject;         // MyObject는 Network Object
로부터 상속했다
ObjectID objectId = myObject ->GetID();
```

ObjectID필드를 가진 패킷을 생성하고나서 클라이언트로 보내라. 그 패킷은 요청한 오브젝트의 타입이 무엇이든지간에 ID_CREATE_OBJECT(여기서는 서버가 송신자로 보내는 오브젝트를 만들어도 된다는 사용자정의메시지)의 계열 중 어떤 것이 될 것이다.

클라이언트가 패킷을 받으면:

```
MyObject * myObject = new MyObject;         // MyObject는 Network Object
로부터 상속했다
myObject ->SetID (objectId);                    // objectID는 서버응답의
패킷에 포함된 것이다
```

주석에 따르면, 이 부분에서 어떠한 치트의 종류를 검출하는 것은 좋은 생각이다. 예를 들면 클라이언트가 50개의 탱크의 생성을 요청하고, 5초씩이나 걸렸다면, 무언가가 잘못된 것임을 알 수 있다. 일반적으로 *RakNet*은 어떤 종류의 패킷의 수정이나 중복을 막는다. 그러나 **무엇이든지** 주어진 충분한 시간 안에는 해킹당할 수 있으므로, 코딩하기 쉬운 만큼 별 지장이 있을 수 없다(쉬운 코드는 빨리 처리된다??).

프로그래밍 팁

게임상의 모든 것은 NetworkObject로부터 파생하지 않아도 된다. 오직 다양한 시스템상에서 어떠한 공통적인 방식으로 참조할 필요가 있는 것들만 이것이 필요하다. 만약 시스템마다 하나의 타입의 오브젝트를 참조하는 명확한 방법이 있다면, 이것(NetworkObject)이 굳이 필요하지는 않다.

NetworkObject에서 파생하는 경우
- 게임에서 특정한 순서없이 많은 적들이 배치되어 있다
- 게임에서 죽어면 삭제되는 많은 적들이 있다
- 게임에서 기름통의 반경 10피트안으로 플레이어가 걸어가면 터지는 것과 같은 트리거가 있다. 이 트리거는 NetworkObject로부터 파생될 수 있다

NetworkObject에서 파생하지 않는 경우
- 게임의 모든 적들은 하나의 배열에서 특정한 순서로 생성된다. 이러한 경우 단지 이 배열의 인덱스만 보내면 된다(ObjectID대신 인덱스)
- 게임에서 오직 하나의 성이 존재한다. 어떠한 패킷이 성을 참조한다면, 암시적으로 어떤 성인지 알 것이다
- 오브젝트는 네트워크상에서 절대로 참조되지 않는다. 예를 들면, 총에서 발사된 총알이다. 총을 쏘는 플레이어와 맞는 플레이어는 상호작용을 하지만, 총자체에는 신경쓸 필요가 없는 것이다.

다른 기능

NetworkObject* GET_OBJECT_FROM_ID(ObjectID x);
이 함수는 내부의 AVL balanced binary tree안에서 ObjectID를 찾고 그것의 NetworkObject에 대한 포인터를 반환한다. 이 후, 그 포인터는 무엇을 찾는지의 상황에 따라서 적절히 캐스팅할 수 있다(간단히 모든 네트워크에 걸쳐 고유한 ObjectID에 대응하는 고유한 NetworkObject를 찾아내는 것이다).

예시:
```
MyObject *myObject = (MyObject *) GET_OBJECT_FROM_ID(packet->objectId);
if (myObject==0)
	return;			// 오브젝트를 찾지 못했다
```

static unsigned short GetStaticItemID(void);
static void SetStaticItemID(unsigned short i);
이것들은 Network Object를 완전히 이해하지 못한다면 아마 필요하지도 쓸일도 없는 고급함수이다.
SetStaticItemID는 어떤 특정한 값에서 Object의 ID번호가 시작되도록 하게 하며, 서버가 자주 사용하는 ID번호인 높은 수치의 값을 얻기위해 쓸 수 있다. 이것은 이미 ID를 사용한 상태의 존재하는 게임으로 서버가 로딩될 때만 쓸모있다. 예를 들면, 서버상에서 0~1000의 ID를 사용한 게임을 저장하고 그것을 다시 시작하고 싶으면, 그 게임은 로드하여 SetStaticItemID(1001)을 호출(GetStaticItemID()는 사용한 최대 ObjectID인 1000을 반환할 것이므로 시작을 1001로 줄 수 있다)할 수 있다; 즉, 새로 할당되는 ID(1001이후의 ID)는 기존의 ID(0~1000의 ID)와 충돌하지 않을 것이다.

NetworkObject의 생성
NetworkObject가 생성되면 SetID로 지정하지 않아도 기본적으로 기존의 가장최근의 NetworkObject의 ObjectID보다 1증가된 수치의 ObjectID를 갖는다. 그러나 이 1 증가된 수치의 ObjectID가 이미 존재한다면 존재하지 않을 때까지 1씩 증가시켜서 ObjectID가 할당된다.

서버와 클라이언트의 SetID의 차이
SetID는 클라이언트에서도 호출가능하지만 서버에서 받은 ObjectID만 지정하여 호출할 수 있고, 클라이언트상에서 임의 값으로 호출하지 않는다.
반면, 서버는 SetID에 중복되지 않는 새로운 임의의 ObjectID를 지정하여 호출할 수 있다.

Distributed Objects
Distributed Objects

분산 네트워크 객체 시스템(the distributed network object system)

분산네트워크오브젝트시스템은 RakNet의 최상위층에 있다. 그 개념은 간단하다:
- 어느 한 시스템에서 생성된 오브젝트는 모든 시스템에서 생성된다.
- 어느 한 시스템에서 파괴된 오브젝트는 모든 시스템에서 파괴된다.
- 태그 메모리는 시스템간에 매치된다. 새로운 플레이어가 연결하면 이 오브젝트들은 그 플레이어의 시스템에서도 생성된다

이것은 개념상으로 매우 유용하다. 왜냐하면 이것은 게임오프젝트로 직접 유추되기 때문이다. 예를 들면, 20명의 멀티플레이어 게임에서 하나의 탱크는 실제 20번 생성된다. 그러나, 특정 플레이어에 관한 한 탱크는 단지 한 개일 뿐이다. 그 플레이어는 그 위치, 방위, 남은 포탄의 수 및 아머수치가 모든 시스템에서 같을 것이라고 기대할 것이다.

관례적으로, 탱크를 유지하기 위해서는 탱크에 대한 연속적인 커스텀 패킷을 세밀하게 만들 필요가 있다. 한 패킷은 위치를 나타내고, 또 다른 패킷은 탱크가 발사하는 것을, 또 다른 것은 탱크가 데이미를 입었는지를 나타내는 것이다. 분산네트워크오브젝트시스템을 사용하면, 이 3 멤버의 값과 모든 것을 자동으로 매치하여 동기화시킬 수 있다.

분산오브젝트시스템은 그 강력함도 있지만 결점도 있다

Strengths
- 구현은 단지 몇분만에 되고 한번 작동하면 잘못될 수가 없다.
- 새로운 플레이어에게 게임 데이터를 전송것에 대한 걱정이 필요없다.
- 싱크에서 벗어난 게임 오브젝트에 대한 걱정이 필요없다.
- 언제나 정확한 최종결과만 수신하기 때문에 오브젝트간의 상호작용의 복잡성에 대해 신경쓸 필요가 없다.
- 이미 존재하는 싱글플레이어게임에 네트워크를 장착하기 쉽다
- 보간법(Interpolation)이 장착되어 있다.

Weaknesses
- 타임스탬핑이 지원되지 않으므로 위치를 추론할 수 없다.
- 액션의 결과를 추적하는 것은 액션을 발생시키는 트리거를 송신하는 방식보다 정확하지 않다.
- 액션이 예측에 대해 대역폭낭비가 심하다. 예를 들면, 로켓이 20개의 폭뢰을 발사했다면 이 시스템은 이것들이 떨어질때까지 20개 오브젝트의 수신이 요구된다. 로켓이 발사되면 오직 그 로켓만 패킷에 쓸 수 있으므로 하나당 하나의 송신이 된다. 이 폭뢰들은 물리현상이 반복되는 것으로 가정하여 정확히 떨어질 것이다.
- 비록 배열을 가지고 다른 요소를 동기화하거나 그것을 구조체로 둘러싸서 이 오브젝트를 얻을 수 있음에도 불구하고, 현시점에서는 배열이나 포인터를 지원하지 않는다.

이러한 약점에도 불구하고, 이것은 많은 게임에 의해 사용되는 주된 시스템이다.

분산오브젝트시스템을 구현하는 방법

분산오브젝트시스템을 사용하기전에, RakClient나 RakServer의 인스턴스를 Distributed Netrowk Object Manager로 등록해야 한다.

```
DistributedNetworkObjectManager::Instance()->RegisterRakClientInterface(rakClient);
DistributedNetworkObjectManager::Instance()->RegisterRakServerInterface(rakServer);
```

Multiplayer 클래스는 자동으로 분산오브젝트네트워크메시지를 다룰 수 있다. 만약 Multiplayer 클래스를 사용하지 않는다면, 이 패킷들 타입에 대해 스스로 구현해야 한다:

==========

예전에 네트웍게임을 만들기 위해 사용했던 네트웍 라이브러리 Raknet의메뉴얼이다.

물론, 오래전 버전이기 때문에 지금이랑은 많이 다를 수 있다는 점은 참고하시길..

익히기는 그렇게 어렵지 않았으며 지금도 그럴것이다.

상용버전과 무료버전이 따로 존재 했던걸로 기억하는데 지금은 라이센스가 어떻게 진행되고 있는지는 모르겠다.

진짜 오래전인데, 홀펀칭 문제도 해결했었고

이 라이브러리를 이용하여 비상용게임이긴 하지만 서비스까지 진행했었다.

당시 메뉴얼을 다운받아 번역을 했었는데, 자료가 남아있어 공개해 본다.

http://www.jenkinssoftware.com/