Reliable UDP 구현과 활용

http://choiwonwoo.egloos.com/924906

다른 용도로 예전에 작성했던 글인데, 말투만 좀 수정해서 업데이트 합니다. (^^)

네트워크 게임에서 UDP는 물리적인 네트워크 Latency에 근접한 반응 속도를 나타낸다는 의미만으로도 기여하는 역할은 매우 크다고 할 수 있습니다.
다만 데이터의 신뢰도가 떨어지기 때문에 제한적으로 사용되는 경우가 많습니다.
좀 더 적극적으로 UDP의 단점들을 보완하는 쪽으로 접근해서, 데이터 전송에 신뢰성을 준다면 TCP의 기능 일부를 효과적으로 대체할 수도 있을 것입니다.
이런 Reliable UDP 구현은 해외 유명 게임과 네트웍 엔진에서도 많이 시도되었고, MO의 활성화와 더불어 기술상 큰 축이 되지 않을 까 생각됩니다.
(보안상의 이유로 공개적용 여부는 알 수는 없겠지만요.)

생소한 주제는 아니지만, UDP 활용을 중심으로 Reliable UDP를 단계별로 설계하고 구현하는 부분을 다루어 보겠습니다.

1. UDP의 특성

UDP(User Datagram Protocol)는 TCP와 비교해서 몇 가지를 주목할 특성을 가지고 있습니다. (단점이냐, 장점이냐를 보기 이전에 특성이라고 이해하면 접근하는 데 도움이 됩니다.)

- 연결에 대한 고려 없이 일방적으로 보내거나 받기만 한다
- 전송 데이터는 패킷 단위로 전송된다
- 전송된 데이터에 대한 특별한 처리가 없다

먼저 연결에 대한 처리가 없는 점은 단점이라기 보단, 일종의 특성이라고 볼 수 있습니다. 다만 연결의 처리가 필요한 게임에서는 프로그래머에게는 부담을 주는 요소로 보자면 단점이 될 수는 있을 것입니다. 시스템적으로 보자면 연결별로 별도의 포트를 따로 사용하지 않고, 연결에 대한 버퍼나 정보를 가지고 있지 않기 때문에 시스템적으로 더 적은 자원을 사용합니다. (자원 확보의 측면으로 접근은 큰 의미는 없어 보이므로 역시 장점이라기 보단 특성이라고 보면 맞을 거 같습니다.)

전송 데이터는 패킷 단위로 전송되며, 시스템 설정에 따라 크기 또한 제한을 받는데, 주로 작은 단위의 패킷을 교환하는 게임에서는 이 점이 단점이 될만한 부분은 아니라고 생각됩니다.

다만 전송 상태나 결과에 대한 어떤 처리도 없다는 점은 큰 특성을 가집니다.

- 데이터가 변형될 수 있다
- 순서가 바뀔 수 있다
- 도착하지 않을 수 있다
- 여러 번 도착할 수 있다

이런 특성은 UDP의 특성이기도 하지만 네트웍 자체의 특성이 그대로 드러난다고 보는 게 더 정확할 거 같습니다.
본질적으로는 네트워크 레벨에서 보자면 TCP 역시 이 제약을 벗어날 수 없다는 얘기지만, TCP 는 이 특성들을 모두 보정해주기 때문에 TCP를 다루는 입장에서는 신경쓸 필요가 없지만 UDP를 사용하는 입장에서 보면 부담이 될 수도 있습니다.
이런 단점 같은 특성들에도 불구하고, 장점되는 것은, 이런 에러 정정 과정이 생략되었기 때문에 빠르다(?)는 점입니다.
엄밀한 의미에서 빠르다는 말은 틀립니다.
하지만 TCP의 경우에는 중간에 데이터가 손실될 경우 손실된 이후의 데이터는 버퍼에서 대기를 해야 하기 때문에 실제 클라이언트에 처리된 데이터가 넘어오려면 단방향 네트워크 latency 이외의 지연이 생기기 때문에 결과적으로는 데이터를 처리하는 입장에서는 빠르다는 의미도 아주 잘못된 표현은 아닙니다.
(예를 들어 A, B, C, D를 전송했는 데 A가 손실되고 B, C, D만 전송된 경우, UDP는 api를 통해 B,C,D가 전달되지만 TCP는 A가 다시 재 전송되어 도착할때까지 B, C, D가 전달되지 않습니다.)

이런 특성 중에서 장점을 그대로 살리면서, 단점에 해당하는 데이터의 신뢰도 문제를 일부 해소해서 게임에서 적용해보자는 것이 이 글의 취지입니다.

2. Reliable UDP 디자인과 구현

기본적으로 UDP의 가장 큰 단점인 unreliable 한 특성을 보완하는 것이 주제의 핵심입니다. 단, 단순히 TCP의 특성을 그대로 구현하는 것이 아니라, UDP의 특성을 살리면서 TCP처럼 신뢰할 수 있는 데이터를 만드는 것이 목적입니다.

UDP와 비교했을 때 TCP의 Reliable한 특성을 세가지로 요약할 수 있습니다.

- 데이터의 변형이 없다.
- 데이터의 손실이 없다.
- 데이터가 순차적으로 도착한다.

이런 점들을 모두 극복하여 구현하는 것이 RUDP의 핵심 테마가 됩니다. 이런 점들을 보완할 수 있는 것들을 몇 가지 정리해보자면

먼저 데이터의 변형이 발생할 수 있는 점은, 에러 코드를 삽입해서, 에러 발생 여부를 검출하고, 에러가 있는 패킷은 무시하거나, 재 전송을 요구하는 방식으로 처리하는 것이 적당해 보입니다. (에러 정정 코드를 삽입해서 에러를 검출되면 에러를 복구하는 것도 생각할 수 있겠지만, 실제 UDP 층에서의 에러 율은 극히 낮으므로 오히려 불필요한 대역폭의 낭비를 가져올 수 있습니다.)

데이터가 소실되는 특성은, 상대쪽에서 받았다는 패킷을 보낼 때까지 지속적으로 보내는 방식이면 해결 가능합니다. (받은 쪽에서 소실되었다는 패킷을 전송하는 방식보다는 대역폭의 낭비가 많겠지만, 처리의 난이도 면이나 효율면에서는 이점이 있다고 보입니다.)

데이터를 순차적으로 처리한다는 점은, 받는 쪽에서 패킷의 카운트를 검사해서 순서대로 도착한 것인지 판단하고, 만약 중간에 도착하지 않은 패킷이 있다면, 나중 패킷은 저장해두었다가 앞의 패킷들이 모두 도착한 시점에 도착했다는 처리를 게임에 보내는 방식으로 처리 가능합니다.

이런 기본적인 처리 정책을 바탕으로 RUDP를 구성해보도록 하겠습니다.

* 연결 지향적인 특성

UDP는 이전의 전송 상태나 연결 상태에 대한 처리를 하지 않지만, RUDP는 상대방에 대한 정보를 가지게 되므로, 상대방과의 통신상태에 대한 정보를 필요로 하게 되며, 연결 지향적인 특성을 가지게 됩니다. (RUDP를 구현부에서는 연결에 대한 핸들을 만들어서 사용하면 편리합니다.)

```
len = recvfrom(socket, buffer, sizeof(buffer), 0, &addr, &addrlen);
packet = packet_decode(buffer, len);

con = find_connect(packet);
if (con == NULL)
    con = connected(addr, buffer, len);
if (con != NULL)
    con->recv_packet(packet);
```

* 패킷구성

전송하는 게임 패킷에 RDUP 관련된 헤더를 추가해서 네트워크 패킷을 구성합니다. 일반적으로 필요한 구성요소는 4가지 정도가 되겠습니다.

- 전송타입
- 패킷의 고유넘버
- 에러 검출코드
- 게임 패킷 데이터

* 데이터 전송 타입

RUDP에서 UDP의 특성을 보완하는 것은 전송을 보장하는 것과 순서를 보정하는 것 두 가지가 됩니다. 이런 특성을 토대로 3가지 타입의 패킷을 나눌 수 있습니다.

- UDP고유의 전송을 보장하지 않는 타입 (도착하지 않더라도 문제가 없다.)
- 전송을 보장하지만 순서를 상관없는 타입 (도착만 하면 된다.)
- 전송을 보장하며, 보낸 순서대로 받는 타입 (TCP 의 특성과 비슷)

첫 번째 타입은 그냥 수신 즉시 게임 클라이언트에 넘겨주며, 두 번째 타입은 즉시 처리하되 전송 받았다는 패킷을 상대방에게 보냅니다. 마지막 타입은 받았다는 패킷을 상대방에게 전송은 하며, 순서가 맞으면 즉시 처리하고 그렇지 않는 패킷은 버퍼에 쌓아둔 후 순서가 맞으면 순서대로 패킷을 처리합니다.

그리고 내부적으로 사용하는 ack 패킷이나, 연결 여부를 확인하는 패킷 타입들이 추가될 것이다.

* 패킷 고유 번호

패킷 별로 고유 번호를 보여 함으로써 패킷이 도착했는지 확인하는 ack신호에 사용하며, 이 값은 순차적으로 받는 처리를 할 때 사용할 수도 있고, 수신 쪽에서 재 전송된 데이터인지 파악하는 데도 사용할 수 있습니다.

만약 고유하게 1씩 증가하는 정책이라고 하면 아래처럼 이전에 처리한 패킷인지를 아래처럼 간단하게 검증 할 수 있습니다.

```
if (packet->ackcnt <= recnt_ackcount || find_recvpacket_id(recnt_ackcount)
    return ;    // 이미 처리한 패킷

if (recnt_ackcount+1 == packet->ackcnt)
{
    recnt_ackcount+=1;
    while(find_recvpacket_id(recnt_ackcount))
    {
        remove_recvpacket_id(recnt_ackcount);
        recnt_ackcount++;
    }
}   else
{
    insert_recvpacket_id(recnt_ackcount);
}
```

사용할 때 주의할 점은 언젠가는 오버 플로우 되는 시점부터는 비교문의 조건이 달라진다는 점이다. 즉 위의 예제에서 ackcount가 2바이트라고 가정할 경우 최근에 처리한 ackcount가 65535가 되는 시점에는 ack가 65535 보다 큰 값을 가질 수 없으므로 논리상 어떤 패킷도 수용할 수 없게 됩니다.

```
int _compare(unsigned short a, unsigned short b)
{
    unsigned short diff = (a - b) & 65535;
    if (diff == 0)
        return 0;    //    equal
    if (diff <= 65535/2)
        return 1;    //    greater
    return -1;        //    less
}
```

32767번째 전의 패킷을 처리할 일이 없다고 가정하면 위의 코드처럼 쉽게 처리가 가능합니다. (32767번째 전의 패킷은 과거의 패킷이지만, 더 앞의 패킷이라고 인식됩니다.) ack 카운트에 더 적은 비트 수를 할당하더라도 마찬가지로 처리할 수 있습니다.

* 에러 검출 코드

TCP는 전송 받은 데이터가 유효한지에 검증을 한 데이터를 어플리케이션에 전달하지만, UDP는 유효 여부에 대한 판단을 하지 않기 때문에 데이터의 변형이 있는 지에 대한 판단을 직접 해야 합니다. (사실 에러율은 낮아서 안해도 크게 문제 안될 수도 있지만, 가능하면 문제가 되는 확률을 대폭 줄여주는 게 좋겠죠 ?) 패킷의 CRC 나 Checksum 코드를 삽입해서 가볍게 처리하면 되겠습니다.

에러가 발생한 패킷은 그냥 못 받은 걸로 판단하면 흐름상 문제가 없을 것입니다.
최소한의 에러 검사라도 해둔다면 게임이나 서버 레벨에서는 어느 정도 신뢰하고 데이터를 사용할 수 있을 것입니다.

* 게임 패킷 데이터

게임 패킷 데이터는 실 데이터만 있으면 되지만, 크기가 아주 작은 데이터를 한꺼번에 많이 보내는 경우 nagle 알고리즘을 적용하면, 여러 게임 패킷을 하나로 합쳐서 보내 네트워크 효율을 높일 수도 있을 것입니다. (일반적으로 빠른 반응을 위해서 UDP를 사용하기 때문에 불필요한 경우가 많지만, TCP와는 달리 보내는 타이밍을 결정할 수 있기 때문에, 효율 손실 없이 처리가 가능할 것입니다. - 예를 들면 게임 클라이언트는 프레임 단위로 처리한다는 특성??)

* 패킷 보내기

보내는 경우에는 게임 데이터를 인코드해서 보내면 되며, 반드시 도착해야 하는 데이터는 재전송해야 할 수도 있으므로 리스트에 저장해준다.

```
int rudp_connect::send(char * buff, int len, udp_packet_type type)
{
    rudp_packet packet;
    int len;
    len = encode_packet(&packet, buff, len, type);
    if (ISRECHABLE(type))
        store_send_packet(&packet);
    return sendto(packet, len, & m_addr);
}
```

주기적으로 리스트를 검색해서 도착해야 하는 데이터를 검사해서 재 전송해줍니다. (상대방이 받았다는 ack패킷을 보내는 경우 리스트에서 삭제합니다.)
얼마 동안의 시간(RTO:Retransmission Timeout)동안 Ack패킷이 안 오면 재 전송해야 하는 지를 잘 결정해야 하는데, 짧으면 패킷 loss 시에 빠르게 다시 받아볼 수 있지만, 불필요하게 재 전송하는 경우가 많아질 것이고, (극단적으로는 항상 여러 번 전송할 수도 있다.) 너무 길 경우에는 반응이 느려져서 게임에서의 효율이 떨어질 수도 있을 것입니다. (상대방의 상황에 따라 가변하길 추천!!)

* 패킷 받기

받는 처리는 공통적으로 오류검사를 한 후에, 패킷 타입별로 처리를 해야 합니다.

```
void rudp_connect::recv(char * buff, int len)
{
    rudp_packet packet;
    int len;
    len = decode_packet(&packet, buff, len);
    if (ISINVALIDEPACKET(packet))
        return;
    switch(packet.type)
    {
        case udp_packet_type_a :
            parse_game_packet(packet.data, packet.len);
            break;
        case udp_packet_type_b :
            send_ackpacket(&packet);
            if (ISOLDPACKET(packet.ackcnt) == false)
            {
                update_recv_packetcount(packet.ackcnt);
                parse_game_packet(packet.data, packet.len);
            }
            break;
        case udp_packet_type_c :
            send_ackpacket(&packet);
            if (ISOLDPACKET(packet.ackcnt) == false)
            {
                update_recv_packetcount(packet.ackcnt);
                if (ISRECENTPACKET(packet.seqcnt) == true)
                {
                    parse_game_packet(packet.data, packet.len);
                    for(i=packet.seqcnt+1; 1; i++)
```

```
                {
                    p = find_recv_packet(i)
                    parse_game_packet(p->data, p->len);
                    remove_recv_packet(p);
                }
                update_recv_packet_seqcnt(i);
            }   else
                store_recv_packet(&packet, packet.seqcnt);
        }           break;
    case udp_packet_ack :
        remove_send_packet(packet.ackcnt);
        break;
        ...
  }
}
```

도착 여부를 알려주어야 하는 패킷인 경우 먼저 받았다는 패킷을 보내주고, 만약 도착했다는 패킷인 경우에는 보낸 패킷 리스트에서 해당 패킷을 찾아서 제거합니다.
그리고 순서대로 게임에 전달되어야 하는 패킷은 처리순서가 맞는 패킷이면 처리하고, 아니면 리스트를 만들어 저장합니다.
나중에 중간에 빠진 패킷이 처리되면 순서 대기중인 패킷을 검색해서 처리를 해 줍니다.

* 연결 검사

UDP의 경우 기본적으로 연결이 끊겼는지에 대한 이벤트가 없으므로 disconnect에 대한 패킷을 정의할 필요가 있습니다.
WSAECONNRESET 에러를 내는 경우에는 끊겼다고 판단할 수 있지만, 강제 종료된 경우(혹은 시스템이 강제 종료된 경우)에 한해서는 판단할 수가 없으므로 주기적으로 연결 여부를 확인하는 것이 좋습니다.

3. Reliable UDP 활용과 확장

이상적으로 구현되었다면, 손실되더라도 게임에 큰 영향이 없는 정보와 꼭 필요한 정보를 구별하여 처리할 수 있고, 꼭 필요한 정보도 순서가 보장되는 경우와 안되는 경우를 구별할 수 있습니다.

손실되도 괜찮은 정보로 분류되는 패킷의 경우도 실제 보면 손실되는 패킷의 빈도는 높지는 않기 때문에 적절히 활용하면 꽤 이득을 볼 수 있습니다.
특히나 이동처럼 반복해서 보내는 데이터의 경우는 손실되더라도 다시 갱신되기 쉽고, 실제 비주얼적으로 큰 영향을 안주는 정보의 경우에는 (화면 외곽에서 눈길이 안가는 캐릭터의 움직임??) 적극적으로 활용되도 좋을 것으로 생각됩니다.

TCP와 비교하면 순간적인 네트웍 상의 패킷 오류로 일정 시간동안 데이터를 받지 못하는 경우 (소위 랙)을 조금 줄일 수 있을 것입니다.
(예를 들어 캐릭터가 죽은 정보를 (신뢰+순서무관)하게 사용한다면 A, B, C 캐릭터가 순서대로 죽었다고 서버가 보낼때 A가 손실된 경우에 B, C 캐릭터는 죽고, 다음 타임에 A캐릭터가 죽겠지만 게임상에선 이상하게 느껴지지 않을 것입니다.
모두 tcp 라면 A 정보가 제 전송 된 후에 A,B,C 캐릭터가 죽겠죠. 실제로 TCP로만 구현된다면 B, C가 죽는 것만 딜레이되는 것이 아니라 B, C가 죽은 이후의 모든 액션들에 영향이 있겠죠.)

좀 더 적극적으로 생각해본다면 실제 각 게임에서의 활용될 부분은 많다고 생각됩니다. (내용 중에 생략된 장점 중에 한가지는 UDP는 개인 네트웍간의 제약이 적다는 점입니다. - P2P!!)

전송 방식 자체는 일반적인 형태가 아닙니다만 다룬 내용에서 좀 더 진보된 형태가 Torque Network Library 에 구현되어

있습니다. 데이터의 3 가지 type 외에 신뢰할 수 있는 상태(state)의 개념을 도입하고 있습니다.
예를 들어 상태가 A => B => C 로 바뀐 경우 A를 받고 C를 받으면 B는 못 받더라도 재 전송 요청하지 않는 식으로, 게임에서 중요한 상태의 개념을 신뢰도는 유지한 상태로, 최신의 정보만 유지하는 특성을 살리도록 최적화한 형태입니다.

5.

마지막으로 TCP와 비교를 하는 과정에서 오해가 생길까 말씀드리면, 주제 자체가 TCP를 대체하자는 개념은 아니며, TCP를 보완하는 측면에서 접근한 것입니다. 실제 lowlevel의 도움이 없기 때문에 tcp와 똑같은 기능만 사용한다면 효율이 떨어질 수 밖에 없습니다. (실제로 저의 경우 TCP와 기능을 분담해서 사용있습니다. ^^)

개인적으로는 아이디어만 가지고 재미로 구현했던 것인데, 지금은 프로젝트 진행하고 준비하는데 큰 힘이 되고 있습니다. (단, 저만의 생각입니다. ^^)

구현을 직접 다루는 내용도 아니고, 개념적으로 접근한 글도 아닌 애매한 포지션의 글입니다만, RUDP라는 주제만이라도 게임을 설계하는데 아이디어가 된다면 정리한 보람이 있을 거 같습니다. ^^