컴퓨터 네트워크 Report

실습 제목: 양방향 Go-Back-N 구현

실습일자: 2022년 5월 16일 (월)

제출일자: 2022년 6월 05일 (일)

학 과: 컴퓨터정보공학부

담당교수: 이혁준 교수님

실습분반: 월,수요일 4,3교시

학 번: 2018202065

성 명: 박 철 준

● 서론

1. 제목

컴퓨터네트워크 양방향 Go-Back-N 구현

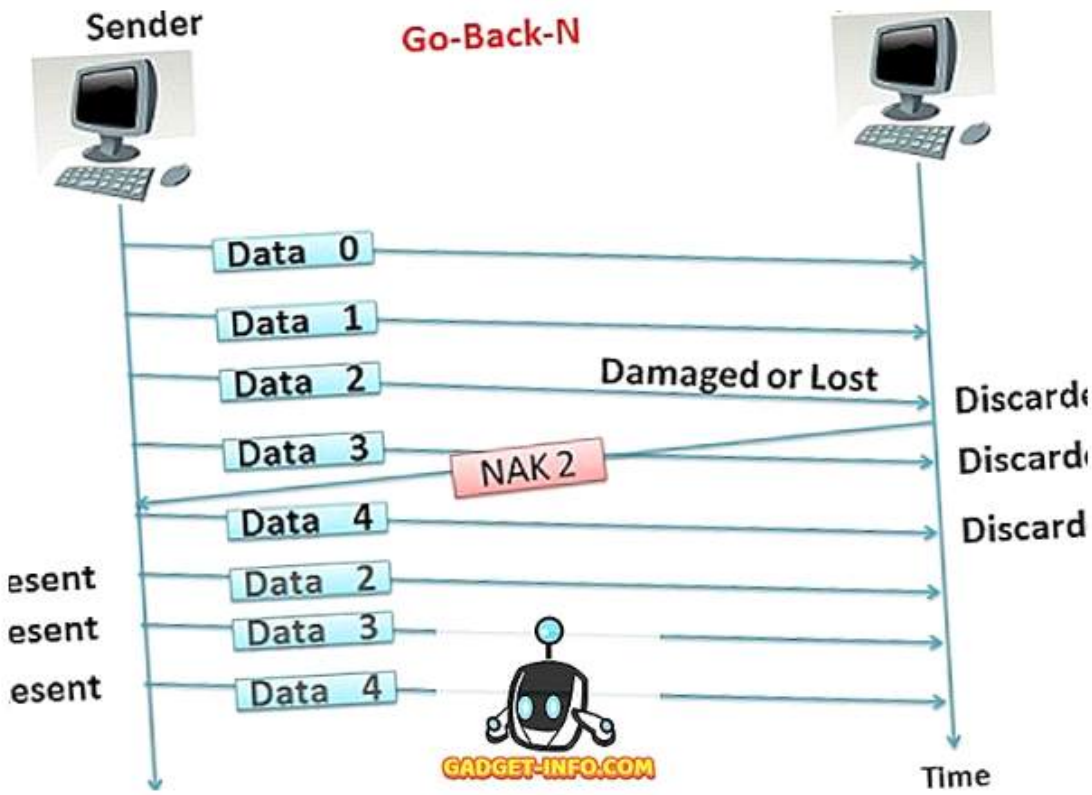
2. 목표

- C언어를 사용하여 양방향 Go-Back-N 프로토콜의 동작을 구현하여야 한다.
- 엑스트라 크레딧을 받기 위해선 8 bit or 16 bit 1's complement checksum operation with bit-wise addition operation is optional을 완성해야한다.

(이번 과제에 있어 bit-wise 연산 checksum 알고리즘은 구현하지 못하였습니다.)

3. 배경 지식

Go-Back-N 프로토콜은 슬라이딩 윈도우 프로토콜이다. 데이터 링크 계층에서 오류를 감지하고 제어하는 메커니즘이다. 센더와 리시버간에 프레임을 전송하는 동안 패킷이 손상되거나 손실되거나 수신 확인이 손실되면 센더와 리시버가 수행할 작업은 아래와 같다.

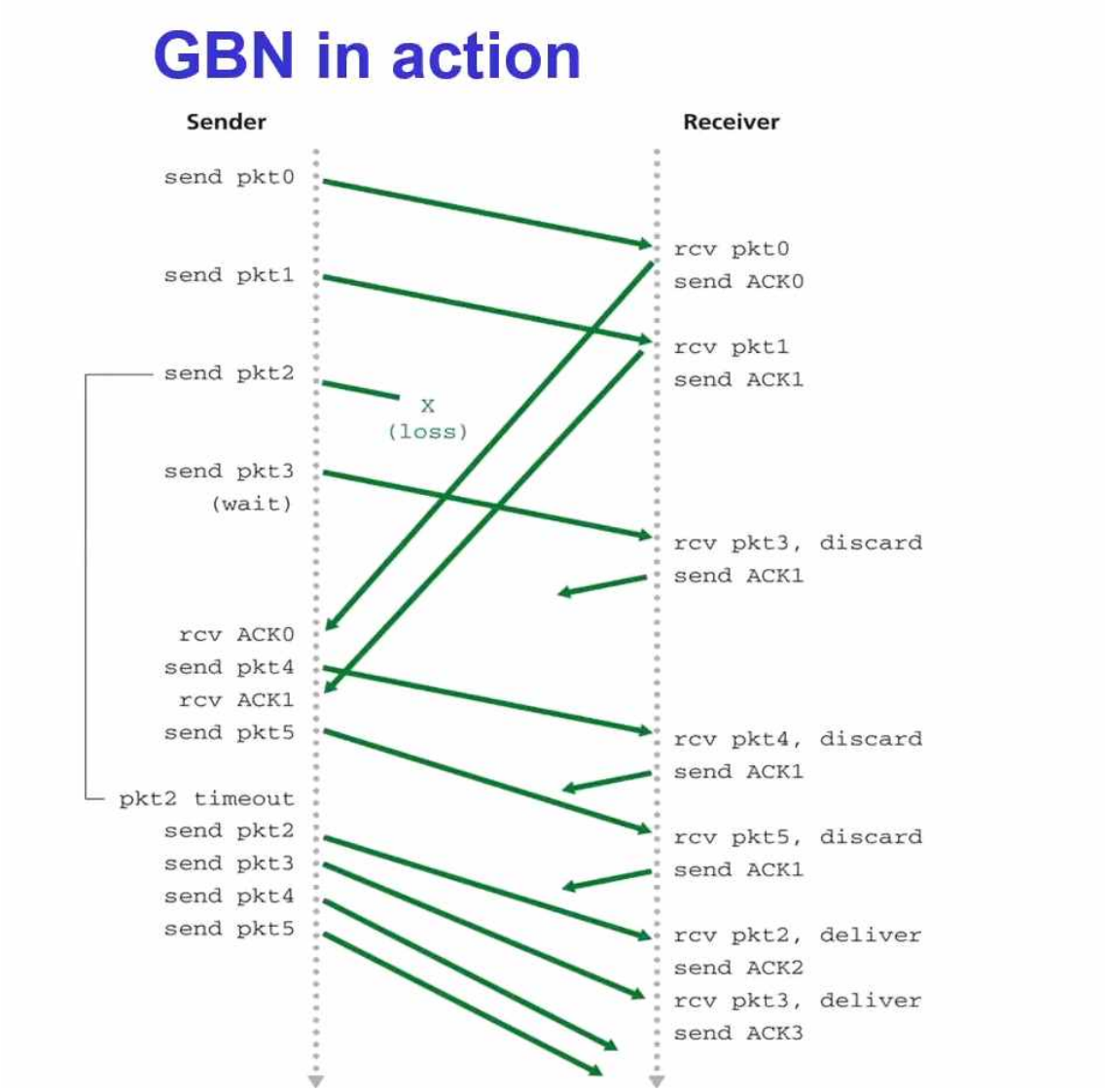


위 그림을 참고할 때 잘못된 패킷 즉 손상된 패킷에 대해서는 리시버는 재전송 될 것으로 예상되는 해당 프레임 번호와 함께 해당 프레임에 대한 NAK (부정 확인 응답)를 전송한다. NAK를 전송 한 후, 리시버는 손상된 프레임 이후에 수신 한 모든 프레임을 폐기한다. 리시버는 폐기된 윈도우 사이즈 버퍼내에 있는 패킷에 대해 ACK (확인 응답)를 보내지 않습니

다. 센더가 손상된 프레임에 대해 NAK를 수신하면 NAK가 참조하는 시퀀스 번호 이후부터 모든 윈도우 사이즈 버퍼내에 있는 패킷을 다시 전송한다.
다음의 경우는 손실의 경우로 리시버는 각 윈도우 사이즈 내 버퍼안에 있는 패킷의 시퀀스 번호를 확인하여 수신한다. 패킷의 시퀀스 번호가 스킵 되면, 리시버는 새롭게 수신된 프레임이 순서가 맞지 않게 수신됨에 따라 패킷의 손실을 용이하게 검출한다. 리시버는 손실된 프레임에 대해 NAK를 전송하고, 수신기는 손실된 프레임 이후에 수신된 모든 이 후 시퀀스의 패킷을 폐기한다. 리시버는 폐기된 프레임에 대해 어떠한 ACK (확인 응답)도 보내지 않는다. 센더는 손실된 프레임에 대한 NAK를 받은 후 NAK가 참조한 손실된 패킷을 다시 전송하고 손실된 패킷의 이후에 시퀀스에 대한 보낸 모든 패킷을 다시 전송한다.

분실 확인
센더가 ACK를 받지 못하거나 전송 사이에 ACK가 손실되거나 손상된 경우 센더는 시간이 끝나기를 기다리고 시간이 다되면 센더는 ACK를 받지 못한 모든 프레임을 다시 전송한다. 센더는 타이머의 도움을 받아 ACK 손실을 식별한다.
ACK 번호는 NAK (부정 확인 응답) 번호와 마찬가지로 리시버가 다음 순서로 기대하는 (expected) 패킷의 시퀀스 번호를 표시하여야 한다. 일반적으로 단방향의 고백엔에서는
데이터 링크 계층은 네트워크 계층 옆에 전송해야하는 윈도우 사이즈 버퍼 내의 패킷만 필요로 하므로 리시버의 윈도우 사이즈 버퍼의 크기는 1입니다. 센더의 윈도우 사이즈 버퍼의 크기는 'N-1' 와 같다.

하지만 일반적으로 다양한 고백앤 프로토콜 방식이 있으며 수업시간에 배운 방식은 다음과 같다.

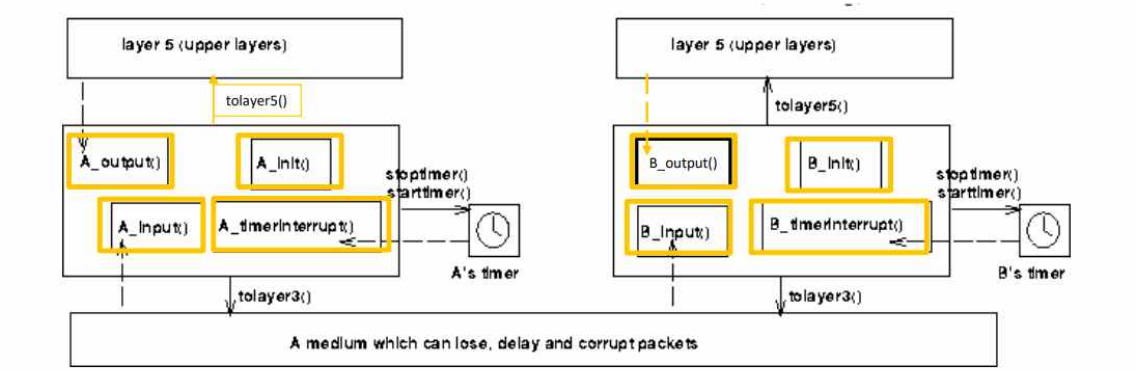


먼저 일반적인 고백엔 프로토콜의 특징을 가져오고 NAK를 발신하는 것과 NAK를 수신하는 부분에 있어서 차이점을 가지는데 이는 다음과 같다.
센더에게서 패킷이 리시버로 전달이 되면 패킷이 손상이 되지 않아 리시버로 도착하면 리시버에서는 패킷을 수신하고 ACK를 전달한다. 다음 패킷이 같은 방식으로 같은 영역에 오게 되는데 만약 이것이 손실되어 리시버에 도착하지 못하고 같은 방식의 같은 영역의 다다음 패킷이 만약 손실 없이 리시버에 도착하게 되면 이것은 out of order의 경우이며 이때 일반적인 고백엔에서는 NAK를 전달하지만 Nak의 전달 없이 지금까지 처리한 누적 ACK즉 그림에서 와같이 ACK1 값을 전송하게 된다. 그리고 센더측은 이때까지 보낸 패킷에 대한 리시버의 ACK를 수신하게 되는데 그림에서와 같이 pkt0과 pkt1의 대한 ACK0, ACK1은 받았지만 시간이 오래되어 time 아웃이 발생 되는데 이전에 ACK0, ACK1을 센더가 받으면서 윈도우 사이즈 버퍼내의 base값을 한칸씩 올려 base를 pkt2로 변경하여 주었기 때문에 타임아웃의

발생으로 윈도우 사아즈 버퍼내의 base 이후 부터의 모든 패킷이 재전송이 된다.
이처럼 패킷이 손실 없이 리시버로 전달 되었지만 중간에 손상되어 왔다면 (이를 판별하는 것은 checksum 알고리즘이다. checksum 알고리즘에 대한 결과 값을 패킷에 실어 보내게 됨)
리시버는 이 패킷을 무시하고 이 패킷에 대한 ACK가 아닌 다시 한번 누적 ACK를 보내게 된다. 다시 센더가 패킷 대한 ACK를 받지 못하여 추후에 다시한번 timeout이 발생하여 해당 패킷에 대한 재전송을 진행하게 된다. 즉 이렇게 볼 때 NAK에 대한 고려 없이도 고백엔 프로토콜을 구현할 수 있다.

- 전체 시스템에 대한 설명과 데이터 구조 도식화 및 설명 마지막으로 동작 원리에 관한 서술

이번 프로젝트에서 구현할 고백엔 프로토콜은 양방향으로 다음과 같은 구성도를 가진다.



즉 양쪽 두 사이드 모두 센더가 될 수 있고 이에 따라 리시버가 될 수 있다. 즉 패킷을 서로 주고 받을 수 있고 애크 또한 주고받을 수 있는 그런 프로토콜이다. 따로 잘못된 패킷을 받았다고 해서 NAK를 보내주거나 받아 재전송을 하는게 아닌 Timer를 이용하여 원하는 ACK가 오지 않았을 경우 다시 한번 패킷을 재전송하는 방식을 가진다.

패킷에 대한 데이터 구조는 다음과 같다.

패킷
layer5층으로부터 받은 메시지
패킷의 sequnce num
리시버로부터 받은 ACKnum
패킷의 손상 확인을 위한 checksum

먼저 layer 5층으로부터 받은 메시지를 담고 있으며 패킷의 sequence num 이것은 레이어 5층으로부터 몇 각각의 영역으로 몇 번째 받은 건지 확인하여 순서를 저장하여 준다. 또한 이 부분은 양방향 고백엔을 구현해야 함으로 이번 프로젝트로 구현하면 A와 B사이드에 대해 layer 5층으로부터 받은 메시지의 각각의 순서이다. 그 다음은 리시버로부터 받은 acknum 값 그리고 패킷의 손상 여부를 확인하기 위한 checksum으로 이루어져있다.

이번 프로젝트의 가장 어려운 점 중의 하나인 전체적인 프로그램의 동작원리는 다음과 같다.
가장 먼저 A영역에서 레이어 5층으로부터 메시지를 받아 패킷을 만들게 되면 프로그램 시작 초기임으로 패킷의 sequnce num은 A영역으로 메시지를 받은 횟수 즉 1이겠지만 ACKnum에 있어서는 한번도 ack를 받은 적이 없기 때문에 999를 저장하여 준다.
또한 이후 패킷의 손상확인을 위해 checksum을 계산하여 패킷 내부에 전달하여 주고 이를 A의 윈도우 버퍼에 저장하여 준다. 이를 A_output을 이용하여 B_input즉 B의 리시버에 전달하게 되면 만약 손실이 없다면 B의 리시버 즉 B_input이 실행 될것이고 손실 되었다면 리시버로부터 해당하는 패킷의 ack를 받지못해 타임아웃이 발생되어 재 전송 할 것이다. 이후 B_input이 제대로 실행되었어도 패킷이 손상되었다면 해당 패킷에 대한 ack를 만들어 센더

에 전달 할 수 없으므로 센더는 다시 해당 패킷의 ack를 받지 못하여 타임아웃이 발생되어 윈도우 사이즈에서 해당 패킷 포함 이후의 모든 패킷을 재 전송하게 된다. 손실 손상 없이 잘 도착하게 된다면 B_input은 B_output을 통해 A즉 센더에 ack를 보내줄 준비를 한다.
이후 과정이 매우 중요한데 만약 A영역에서 다시한번 레이어 5층에서 받은 다른 메시지의 패킷을 전달한다고 하면 다시 한번 위에 같은 경우를 반복하고 손실과 손상이 없는 경우 ack의 값을 기존 ack의 값과 누적해서 B_output은 A즉 센더에 ack를 보내줄 준비를 한다.
이렇게 ack값을 누적하여 보내줄 준비를 하다가 B의 센더에서 레이어 5층으로부터 패킷을 전달받아 A의 리시버에 패킷을 보내주게 되는데 이때 패킷에는 B의 리시버가 A의 센더에게서 지금까지 받았던 전달하지 않고 누적했던 ack의 값을 패킷의 acknum에 저장하여 한번에 보내주게 된다. 다시 이것을 A의 리시버가 받게 되면 패킷은 다시 레이어 5층으로 보내고 ack는 확인하여 윈도우 버퍼의 base값을 해당하는 ack의 값의 하나 이후의 값으로 재조정하여 후에 만약 타임아웃이 발생하였다 하면 지금까지 받은 정상 ack의 패킷을 제외하고 이후 ack를 받지 못한 패킷을 재전송 하게 된다. 또한 다시 정상적으로 ack를 받았다고 가정하고 돌아와서 이 경우도 앞서와 마찬가지로 A의 센더는 B리시버에 B센더가 보냈던 패킷을 정상적으로 수신하였고 이에따라 축척한 정상적인 ack의 누적값을 보낼 준비를 하게된다. 이것을 이제 서로 상호작용을 하여 작동을 시키게되면 이것이 양방향의 고백엔 프로토콜이 된다.
이러한 동작방식을 가정하고 코드를 설계하였고 이후 각 함수에 설명은 다음 목차에서 하겠다.

- 각각의 함수에 대한 설명

-getchecksum함수(extra credit 방식으로는 구현하지 못하였습니다.)

```
int getchecksum(struct pkt packet)
{
    // checksum 구하는 함수
    //엑스트라 크래딧 방식으로는 구현하지 못하였습니다
    int checksum = 0;
    checksum += packet.seqnum;
    checksum += packet.acknum;
    for (int i = 0; i < sizeof(packet.payload) / sizeof(char); i++)
        checksum += packet.payload[i];     //모두 더하여 checksum설정
    return checksum;
}
```

패킷의 체크섬을 제외하고 모든 데이터를 더하여 checksum에 저장하여 리턴한다. 후에 센더에서 한번 호출하여 계산한 결과값을 패킷에 저장하고 리시버에 전달한다. 이후 리시버에서 다시한번 받은 패킷에 대해 함수를 호출하여 체크섬을 제외하고 다 더해준 뒤 이값을 패킷으로 온 체크섬하고 비교하여 같은지 확인한다.

- 각 사이드의 초기화 함수

```
struct pkt pkt_A_window[1024];  //A output에서 보내는 패킷을 저장하는 장소이며 추후에 윈도우가 될 수 있음으로 장소의 사이즈는 넉넉히 잡아주며 정확한 윈도우 사이즈는 WINDOWSIZE로 정의
struct pkt pkt_B_window[1024];  //B output에서 보내는 패킷을 저장하는 장소이며 추후에 윈도우가 될 수 있음으로 장소의 사이즈는 넉넉히 잡아주며 정확한 윈도우 사이즈는 WINDOWSIZE로 정의

//A side 변수를 초기화
void A_init(void)
{
    memset(pkt_A_window, 0, 1024);
    send_base_A = 1;
    next_seqnum_A = 1;
    expected_seqnum_A = 1;
    A_ackstate = 0;
    A_my_acknum = 0;
}

//B side 변수를 초기화
void B_init(void)
{
    memset(pkt_B_window, 0, 1024);
    send_base_B = 1;
    next_seqnum_B = 1;
    expected_seqnum_B = 1;
    B_ackstate = 0;
    B_my_acknum = 0;
}
```

프로그램이 시작하면 init으로 프로그램 초기화 이후 수행되는 함수이며 각각의 초기값을 설정하여 준다.

- pkt_make함수

```
struct pkt pkt_make(int seqnum, int acknum, char* data)
{
    //packet을 만드는 함수
    //
    //새로운 packet을 생성하고 초기화하는 과정
    struct pkt new_pkt;
    new_pkt.seqnum = seqnum;
    new_pkt.acknum = acknum;
    for (int i = 0; i < sizeof(new_pkt.payload) / sizeof(char); i++) {
        new_pkt.payload[i] = data[i];
    }
    new_pkt.checksum = getchecksum(new_pkt);
    return new_pkt;
}
```

-

패킷을 만들어 초기화하는 함수

- 각 사이드의 output 함수

A 사이드와 B 사이드의 각각의 output함수는 같은 흐름을 가지고 똑같으므로 A사이드를 기준으로 설명하고자 한다.

```
A_output(struct msg message)
{
    //layer5에서 데이터 message를 받고 패킷을 만들어 A에서 B로의 패킷과 액크를 함께 보내기위해 호출되는 함수

    if (next_seqnum_A < send_base_A + WINDOWSIZE)
    {
        if (A_ackstate == 0)
            A_my_acknum = 999;   //다른 사이드의 sender에게서 받은 ack가 존재하지 않는 경우
        struct pkt received_packet_for_layer5;    //보낼 packet
        received_packet_for_layer5 = pkt_make(next_seqnum_A, A_my_acknum, message.data); //message를 추출하고 packet을 생성함
        pkt_A_window[next_seqnum_A-1] = received_packet_for_layer5;  //A 사이드의 window에 packet을 저장
        tolayer3(0, received_packet_for_layer5);   //packet을 B_input으로 전송 보냄
        if (A_my_acknum!=999)   //ack를 같이 보내는 경우 출력
        {
            printf("A_output : send_packet with ACK (ACK = %d, seq = %d) : ", received_packet_for_layer5.acknum, received_packet_for_layer5.seqnum);
            for (int i = 0; i < 20; i++)
                printf("%c", received_packet_for_layer5.payload[i]);
            printf("\n");
        }
        else //nak여서 acknum에 999를 보내는 경우 출력
        {
            printf("A_output : send_packet without ACK (seq = %d) : ", received_packet_for_layer5.seqnum);
            for (int i = 0; i < 20; i++)
                printf("%c", received_packet_for_layer5.payload[i]);
            printf("\n");
        }
        A_ackstate = 0;   // ack를 보냈다고 가정하고 ACKstate 를 0 으로 변경해주어 ACK를 보내지 않는 상태를 의미하도록 사용함

        if (send_base_A == next_seqnum_A)
        {
            //stoptimer(0, (float)TIMEOUT);//안전한 카운팅을 위한 타이머 껐다 키기
            starttimer(0, (float)TIMEOUT); //timer를 시작함
        }

        next_seqnum_A++;    //next sequence number를 1증가시킴

    }
    else
    {
        printf("A_output : Buffer is full.Drop the message.\n");

        //data 전송을 거부한다.
    }
}
```

먼저 레이어 5로부터 메시지가 들어오게 되면 A영역의 next_seqnum가 A의 베이스 더하기 윈도우 사이즈(윈도우 버퍼의 허용가능 사이즈 내에 있음을 확인) 보다 큰 경우 데이터의 전송을 거부하며 버퍼가 꽉찼음을 알리고 버린다. 작은 경우 기능을 수행하는데 초기의 A의 애크 스테이트 값이 0이기 때문에 애크넘을 999로 바꾸게 되고 패킷을 만들고 A의 윈도우에 저장을 한다. 그리고 tolayer3를 통해 layer3층으로 갔다가 B의 리시버 이동시킨다. 이때 만약 A의 acknum이 999인 경우 ack를 패킷과 같이 보내는 것이 아니기 때문에 이 경우를

고려하여 준다. 다음 과정으로는 ack를 보냈다고 가정하고 A의 ack state를 0으로 변경하여 준다 999여서 ack를 보내준 게 아니여도 이러한 과정을 하는 이유는 다시한번 A_output이 실행되었을 때 이전이 999여서 애크를 보내지 않았다고 해도 다음 수행에 있어서는 애크를 보내어 다시 보낼 애크가 없다는 것을 알려 주어야 하기 때문에 알고리즘 적으로 문제가 없다.
이후 A의 send_base와 next_sequm 즉 프로그램 시작 초기여서 윈도우 내 패킷이 없는지를 확인하여 타이머를 시작해주는 부분을 추가한 것이다.
이후 A의 next_sequm을 1 증가 시켜주어 후에 다시 이 함수가 실행될 때를 대비한다.

-각 사이드의 input 함수

output함수는 A로 설명하였기 때문에 input함수는 B로 설명하여 각 흐름의 정확한 설명을 용의 하게 하겠다.

```
B_input(struct pkt packet)
{
    //B의 영역의 receiver이며 A영역 에게서 packet을 받을경우 호출되는 함수
    B_ackstate = 1;//  B영역 센더가 ack를 보낼 수 있는 상태로 만들어 줌

    // 받은 패킷의 checksum 결과 비교
    if (getchecksum(packet) == packet.checksum)
    {
        //checksum에 이상이 없는 경우
        printf("B_input : recv packet (seq = %d) : ", packet.seqnum);
        for (int i = 0; i < 20; i++)
            printf("%c", packet.payload[i]);
        printf("\n");
        int acknum = packet.acknum; //packet의 acknum 추출
        int seqnum = packet.seqnum; //packet의 seqnum 추출


        /*
        //디버깅을 위해 추출값을 확인하는 출력문이다.
        //printf("seqnum : %d \n", seqnum);
        //printf("expected_seqnum_A : %d \n", expected_seqnum_A);
        //printf("next_seqnum_A : %d \n", next_seqnum_A);
        */
```

```
int ack_check = 0;//패킷이 데이터가 기대하는 시크넘을 만족하여 액크를 생산하는지 확인하는 변수

//receiver로서의 동작
if (expected_seqnum_B == seqnum)
{
    // Expected_seqnum == next_seqnum(즉 패킷의 seqnum)이유는 패킷을 만들 때 넥스트 시퀀스넘버를 넣어줌으로 기대하는 시퀀스는 패킷내부의 시퀀스 넘버이당.
    //(여기 부분 중요)
    tolayer5(1, packet.payload);    //layer5 에게 data 전달
    B_my_acknum = expected_seqnum_B; //acknum을 expected sequence number로 저장하며 갱신
    printf("B_input : got ACK (ack = %d)\n", B_my_acknum);
    expected_seqnum_B++;      //B의 expected sequence number를 1증가시킴 이유는 다음에도 같은 영역으로 전달하는 패킷을 받을 때 확인해야하고 이때 기대하는 시퀀스 넘버는 이 부분을 지난 다음
    :                         //숫자 1이 증가한 상태에서 비교 해야함으로 (여기 부분 중요)
    ack_check++; //ack를 체크함으로서 999가 들어왔을 때 초기의 ack를 가지고 있지 않아서 999가 들어왔는지 nak의 역할을 하는 999인지 판단한다.
}
else
{
    //out of order의 경우 drop
    printf("B_input : not the expected seq. (expected_seq = %d)\n", expected_seqnum_B);
}
```

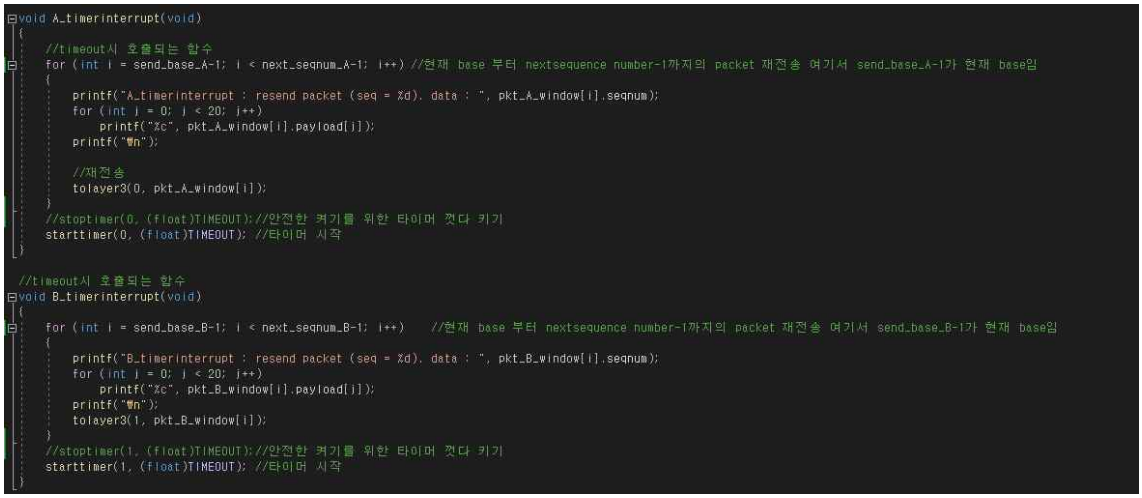
```
        }
        //sender로서의 동작
        if (acknum != 999&& ack_check !=0 )  //acknum 이 999가 아닐때, 즉 NAK가 아닐때
        {
            send_base_B = acknum + 1;
            if (send_base_B == next_seqnum_B)
            {
                //Stop timer
                stoptimer(1, (float)TIMEOUT);
                printf("B_input : stop timer.\n");
            }
            else
            {
                //Start timer
                stoptimer(1, (float)TIMEOUT);//안전한 켜기를 위한 타이머 껐다 키기
                starttimer(1, (float)TIMEOUT);//타이머 켜기 타이머 연장 느낌
                printf("B_input : start timer.\n");
            }
        }
        else if(acknum == 999&&ack_check == 0)//acknum이 실행 초기의 ack를 가지고 있지 않아서 999일때를 제외하고 즉 NAK의 역할을 하는 999일때
        {
            //Got NAK
            printf("B_input : got NAK (ack = %d), Drop\n", packet.acknum);// 네크는 처리 않고 그냥 버림
        }

    }
    else  //checksum에 이상이 있는 경우 받은 packet 버림
    {
        //Packet corrupted
        printf("B_input : Packet corrupted (seq = %d), Drop\n", packet.seqnum);//checksum에 이상이 있는 경우 받은 packet 버림
    }
}
```

A 아웃풋을 통해 B 리시버가 데이터를 받으면 B_input이 실행되는데 가장 먼저 실행되면 일단 손실은 발생하지 않은 경우이고 B 센더가 ack를 보낼 수 있도록 하기 위해 B_ackstate를 1로 만든다. 후에 받은 패킷을 앞성 위에서 설명한 getchecksum 함수를 사용하여 패킷의 체크섬과 비교하고 만약 이상이 있으면 패킷에 손상이 발생함을 출력하고 재전송을 기다린다. 만약 이상이 없다면 패킷을 받게 되고 패킷의 acknum과 seqnum을 추출한다.

또한 ack_check 변수를 선언하여 패킷의 데이터가 기대하는 seqnum을 만족하여 액크를 생산하

는지 확인할 수있게 한다.
B의 기대하는 시퀀스 넘버와 패킷의 seqnum이 같다면 B_input이 기대하는 시퀀스 넘버의 패킷의 seqnum이기 때문에 ack를 만들어야 하고 이를 수행한다. 또한 이때 layer5에 data를 전달해 준다.
그리고 만약 기대하는 시퀀스 넘버가 아닌 경우 기대하는 시퀀스 넘버의 패킷이 아니므로 따로 nak를 보내주지 않고 종료시킨다. 이는 어차피 해당 패킷의 ack를 받지 못하면 해당 패킷을 보낸 센더는 타임아웃 되어 다시 한번 해당 패킷을 재전송 할 것이기 때문이다.
이후 과정은 추출한 패킷의 acknum이 999가 아니고 ack_check 이 0이 아닐 때 즉 ack는 받았지만 기대하는 ack가 아니기 때문에 nak도 만들지 못하고 ack도 만들지 못하는 경우는 제외하고 실행하는 부분은 이것도 이번 프로토콜에서 가장 중요한 부분으로 이것은 B 센더 즉 B_output이 이전에 A리시버에 패킷을 보내고 A리시버가 해당 패킷의 ack를 만들어 A센더를 통해 ack가 B리시버 즉 B_input에 들어온 경우 이며 B센더가 보낸 데이터가 A에 잘 도착했고 layer 5에 전달 됨을 알았기 때문에 B의 베이스 값을 1증가 시켜 만약 타임아웃 되었을 때 잘 도착한 영역을 다시 보내줄 필요가 없으므로 이 과정을 하는 것이다. 그래서 이것이 센더로서의 역할이라고도 할 수 있다. 또한 위 과정을 거치면서 베이스와 넥스트 시퀀스 넘버가 같다면 윈도우 사이즈 내에 패킷이 없다는 것이고 이는 타이머의 스탑을 해줘야 하는 사항이다. 이것 외의 조건에서는 타이머를 연장 해야하는데 이 프로젝트는 각 사이드당 타이머를 하나 밖에 쓰지 않으므로 연장하여 줄 때는 기존의 타이머를 한번 껏다가 다시키는 방식으로 안전한 연장을 하게 하였다.
다음 과정은 ack가 999즉 NAK 이거나 프로그램 실행 초기에 ack를 가지고 있지 않아서 999를 가지는 것 위 두 가지가 남는데 acknum==99 && ack_check==0 즉 초기의 ack를 가지고 있지 않아서 999를 가지고 있는 경우를 제외한 진짜 NAK인 경우를 받았을 때 Nak를 받았음을 알리고 네크는 처리 않고 그냥 버린다. 여기서 이슈가 발생하며 해당 이슈는 이번 프로젝트에서 구현한 고백앤은 NAK를 따로 보내주 지 않았는 데 왜 NAK의 데이터가 오며 이는 어디서 오는 것일까에 대한 이슈에 대해서이며 이는 고찰에서 설명하고자 한다.
이로써 B_input의 동작을 전부 설명하였다. a와 b는 양방향으로 서로 상호 작용하며 동작하기 때문에 같은 흐름과 알고리즘으로 동작을하며 A_input에 대한 동작을 이해하고자 한다면 B_input에 대한 동작 과정을 A_input에 대한 동작 과정으로 거꾸로 생각하면 된다.

- 각 사이드의 timeinterrupt함수



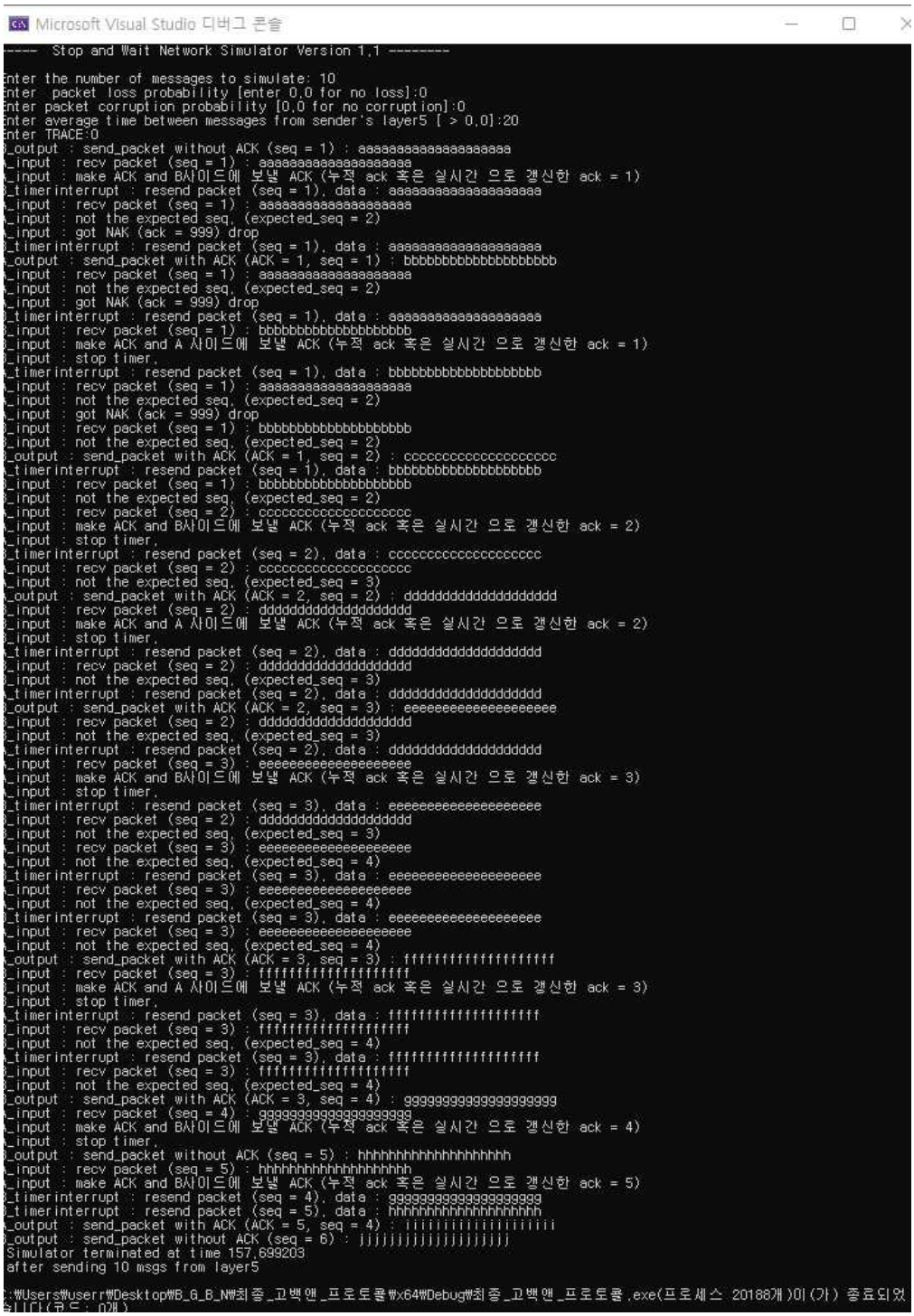
```
void A_timerinterrupt(void)
{
    //timeout시 호출되는 함수
    for (int i = send_base_A-1; i < next_seqnum_A-1; i++) //현재 base 부터 nextsequence number-1까지의 packet 재전송 여기서 send_base_A-1가 현재 base임
    {
        printf("A_timerinterrupt : resend packet (seq = %d), data : ", pkt_A_window[i].seqnum);
        for (int j = 0; j < 20; j++)
            printf("%c", pkt_A_window[i].payload[j]);
        printf("₩n");

        //재전송
        tolayer3(0, pkt_A_window[i]);
    }
    //stoptimer(0, (float)TIMEOUT);//안전한 껴기를 위한 타이머 껏다 키기
    starttimer(0, (float)TIMEOUT); //타이머 시작
}

//timeout시 호출되는 함수
void B_timerinterrupt(void)
{
    for (int i = send_base_B-1; i < next_seqnum_B-1; i++)    //현재 base 부터 nextsequence number-1까지의 packet 재전송 여기서 send_base_B-1가 현재 base임
    {
        printf("B_timerinterrupt : resend packet (seq = %d), data : ", pkt_B_window[i].seqnum);
        for (int j = 0; j < 20; j++)
            printf("%c", pkt_B_window[i].payload[j]);
        printf("₩n");
        tolayer3(1, pkt_B_window[i]);
    }
    //stoptimer(1, (float)TIMEOUT);//안전한 껴기를 위한 타이머 껏다 키기
    starttimer(1, (float)TIMEOUT); //타이머 시작
}
```

만약 어느 사이드든 패킷의 손실 혹은 손상으로 타임 아웃이 발생하게 된다면 위 함수가 실행되고 위 함수는 윈도우 버퍼 사이즈 내에서 문제가 발생한 패킷부터 이후의 모든 패킷을 재전송하게 되고 타이머를 다시 시작하게 된다. 여기서 send base_A-1부터 하는 이유는 실행초기 base를 1로 두고 초기화 하였고 패킷 만들어 윈도우 버퍼에 집어넣을 때 윈도우 버퍼의 위치를 해당 패킷의 다음 시퀀스에 -1 한 값 즉 초기에 버퍼의 0번째 위치부터 들어가기 때문에 현재 base는 base값에 -1을 해주어야 한다.

- 결과화면 캡쳐 및 화면에 대한 설명

-실습 방법 (시나리오)

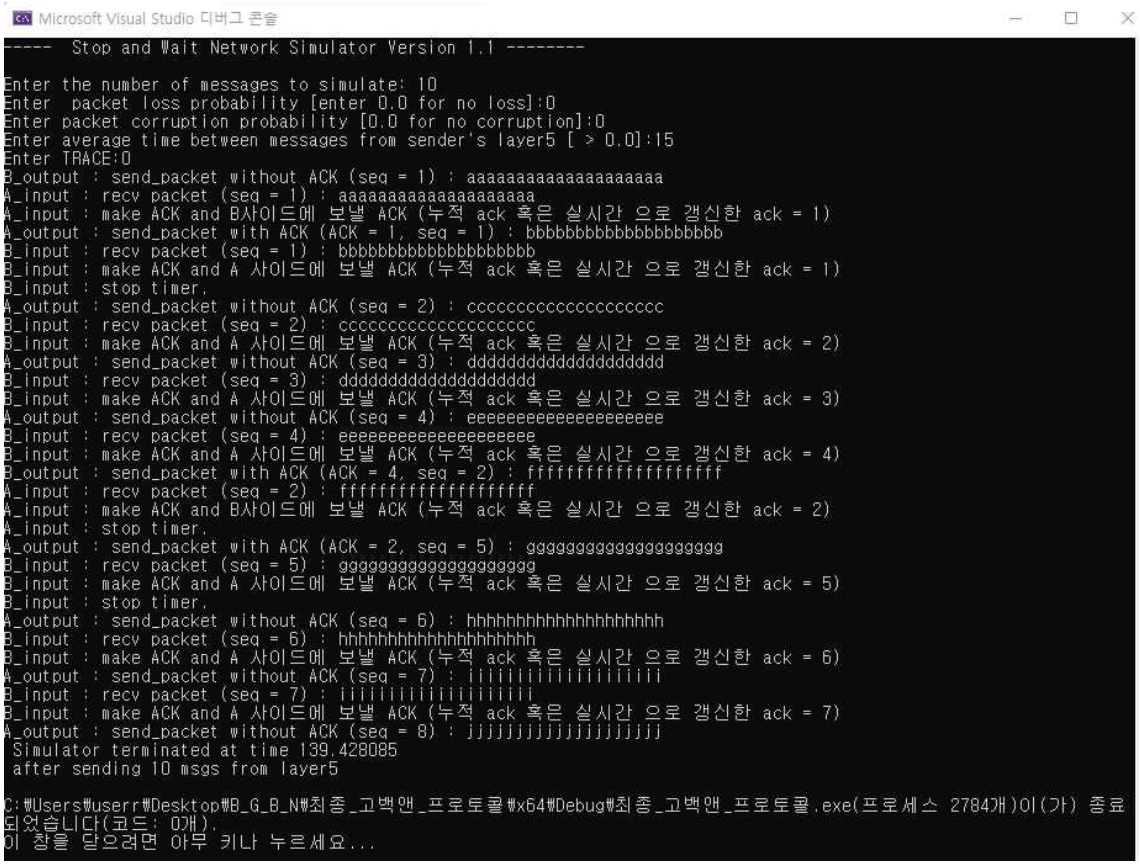
1. 손실과 손상이 없으며 레이어 5로부터 총 메시지를 10개를 받고 메시지를 받는 간격은 타임 아웃의 설정시간(10초)보다 크다고 가정할 때(20초로 두겠다.)



B 사이드와 A 사이드가 서로 상호 작용을 하며 각 사이드의 가장 먼저의 패킷을 보내는 과정

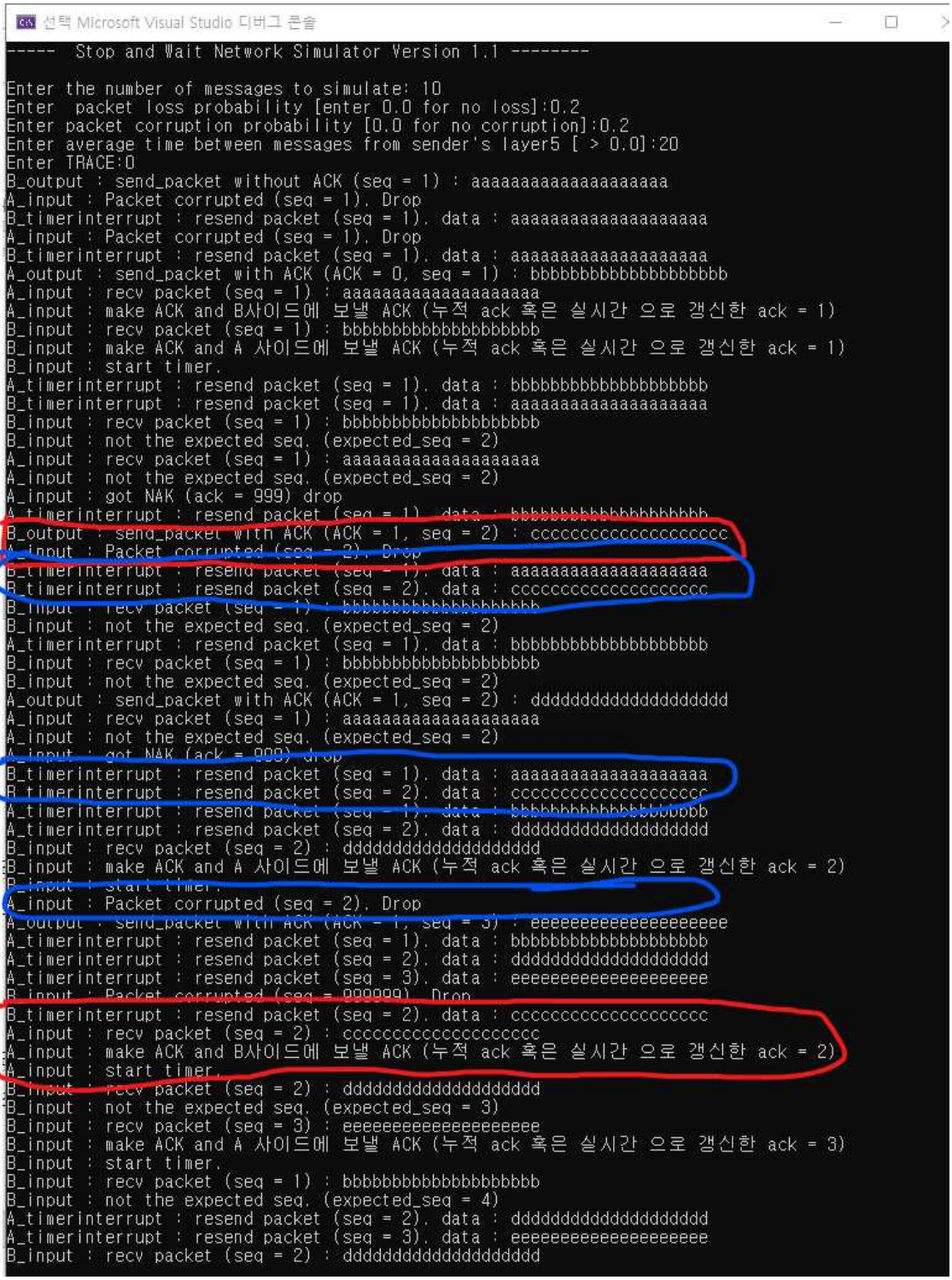
을 제외하고 패킷과 ack를 함께 보내어 받는 모습을 볼 수 있고 여기서 중요하게 B_output이 aaaaaaaaaaaaaaaaaaa데이터를 보내서 A_input이 받았지만 계속해서 B_output이 재전송하는 이유는 레이어 5층에서 메시지를 받는 시간이 타임아웃 시간보다 크고 또 A_output이 aaaaaaaaaaaaaaaaaaa데이터를 가 잘왔다는 ack를 데이터 bbbbbbbbbbbbbbbbbbbb와 함께 보내어 B_input이 aaaaaaaaaaaaaaaaaaa의 잘왔다는 ack와 데이터 bbbbbbbbbbbbbbbbbbbb와 함께 받고 잘왔다는 ack를 수신한 이후에 더 이상 타임인터럽트가 발생해도 aaaaaaaaaaaaaaaaaaa의 데이터는 보내지 않는 걸 알 수 있다. 마찬가지로 나머지 데이터 패킷에도 적용된 모습을 알 수 있다.

2. 손실과 손상이 없으며 레이어 5로부터 총 메시지를 10개를 받고 메시지를 받는 간격은 타임 아웃의 설정시간(100초)보다 작다고 가정할 때(15초로 두겠다.)



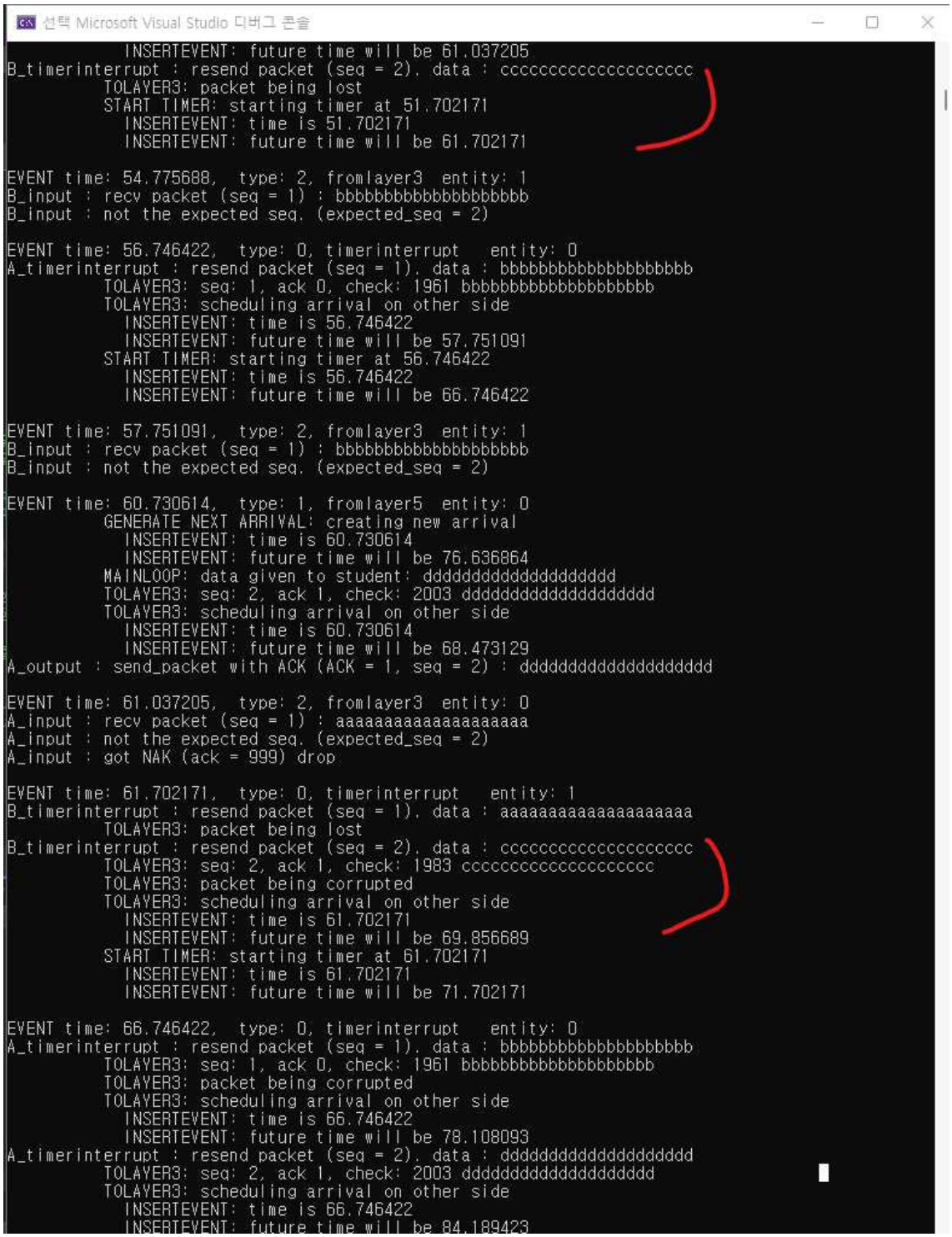
이 경우 또한 위의 경우와 같이 B 사이드와 A 사이드가 서로 상호 작용을 하며 각 사이드의 가장 먼저의 패킷을 보내는 과정을 제외하고 패킷과 ack를 함께 보내어 받는 모습을 볼 수 있고 따로 손실과 손상이 발생하지 않았고 타임아웃 시간내에 ack를 주고 받으며 재전송 할 필요가 없는 루틴으로 돌아가기 때문에 타임아웃은 보이지 않아 재전송하지 않았다.

3. 손실(0.2) 손상(0.2)가 있으며 레이어 5로부터 총 메시지를 10개를 받고 메시지를 받는 간격은 타임 아웃의 설정시간(10초)보다 크다고 가정할 때(20초로 두겠다.)

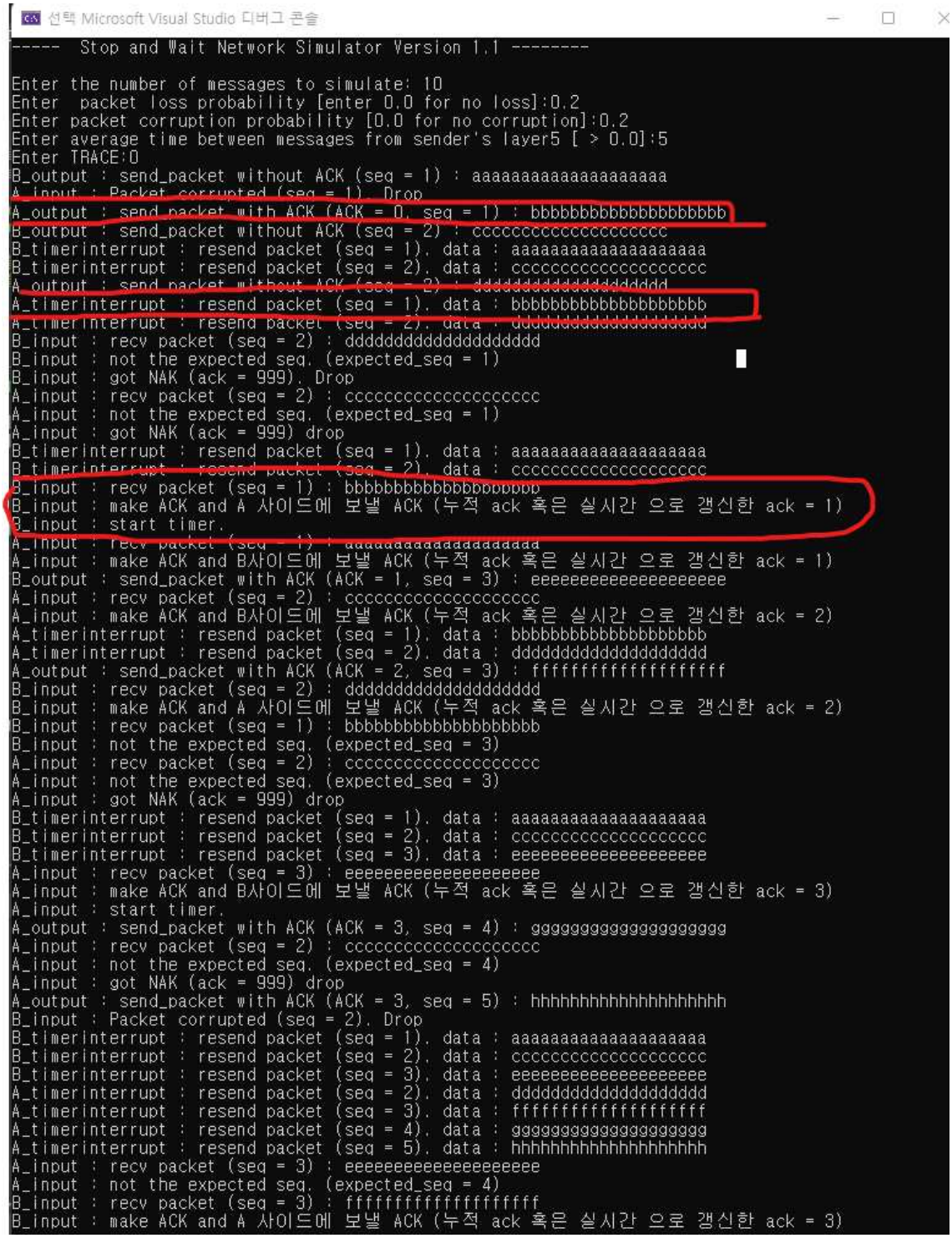


ccccccccccccccccccc데이터가 손실이 났고 표시와 같이 재전송 받아 다시 패킷을 수신한 것을 알 수 있다. 밑에 표시(빨간 표시)가 ccccccccccccccccccc의 오류로 인한 재전송이고 위에도 ccccccccccccccccccc로 인한 재전송이 있는데 이는 aaaaaaaaaaaaaaaaaaa의 데이터의 오류로 인한 윈도우 버퍼내의 오류가 난 시퀀스 이후 모든 패킷 재전송이라

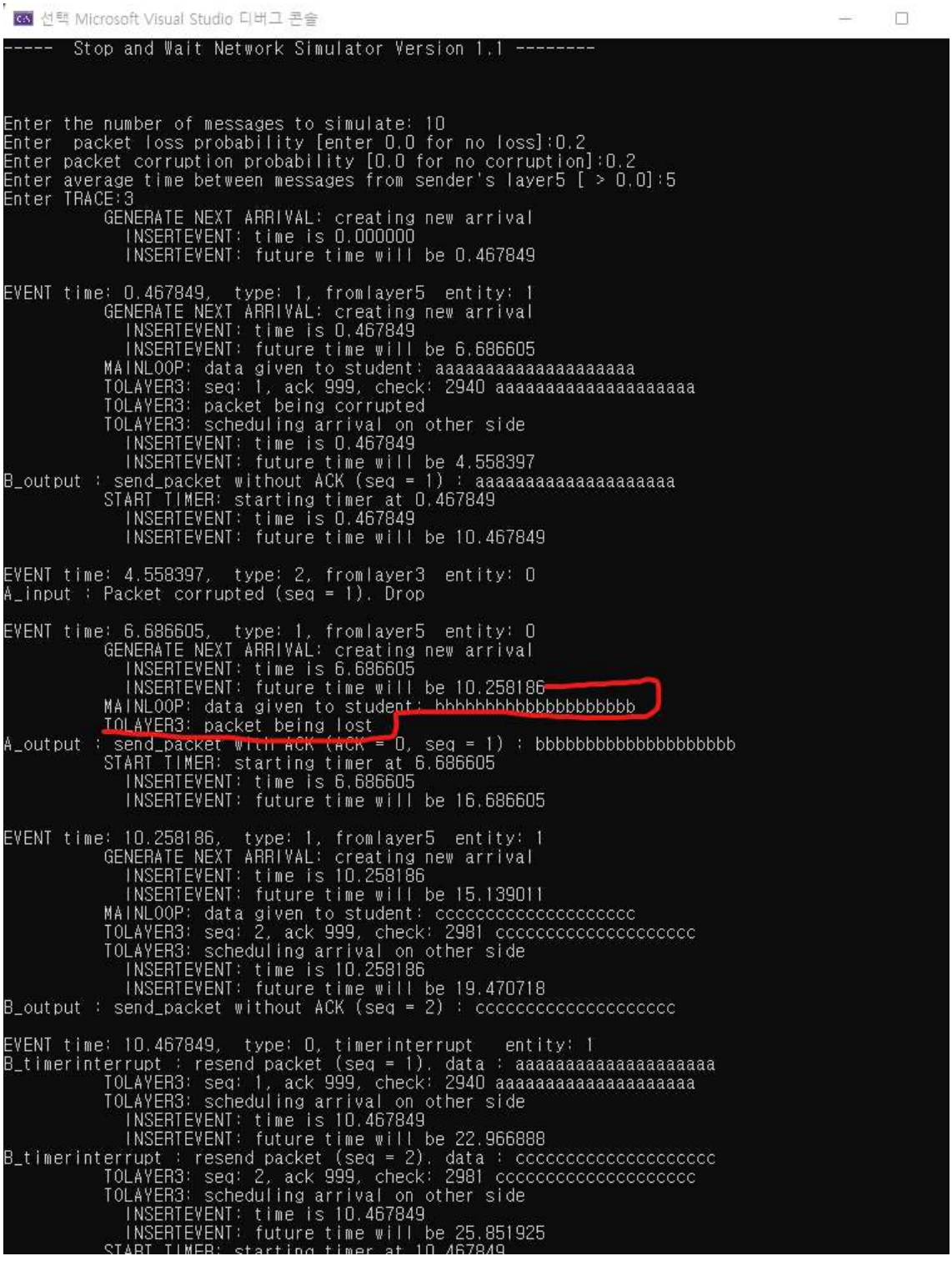
ccccccccccccccccccc도 같이 재 전송 되는 모습을 볼 수 있는데(파란 표시) 이때 A_input을 통해 ccccccccccccccccccc를 받지 못 했다는건 손실이 일어나거나 손상이 일어남을 알 수 있다. 마지막 파란 표시를 참고하면 2개의 재전송 중 하나는 아무런 A_input의 행위가 없으므로 손실하나 마지막 파란 표시를 보면 시퀀스 2즉 ccccccccccccccccccc 데이터에 대한 총 1번의 손상을 볼 수 있다.
Trace 3을 통해 이를 확인 할 수 있다.



4. 손실(0.2) 손상(0.2)가 있으며 레이어 5로부터 총 메시지를 10개를 받고 메시지를 받는 간격은 타임 아웃의 설정시간(10초)보다 작다고 가정할 때(5초로 두겠다.)



A_output이 bbbbbbbbbbbbbbbbbbb데이터를 B_input으로 보냈지만 아무런 실행이 일어나지 않고 타임 아웃을 통해 재전송 되어 이후에 도착한 것을 보면 분명 패킷이 손실 된 경우이며 이는 Trace3을 통해 알 수 있다.


선택 Microsoft Visual Studio 디버그 콘솔
----- Stop and Wait Network Simulator Version 1.1 --------
Enter the number of messages to simulate: 10
Enter packet loss probability [enter 0.0 for no loss]:0.2
Enter packet corruption probability [0.0 for no corruption]:0.2
Enter average time between messages from sender's layer5 [> 0.0]:5
Enter TRACE:3
GENERATE NEXT ARRIVAL: creating new arrival
INSERTEVENT: time is 0.000000
INSERTEVENT: future time will be 0.467849
EVENT time: 0.467849, type: 1, fromlayer5 entity: 1
GENERATE NEXT ARRIVAL: creating new arrival
INSERTEVENT: time is 0.467849
INSERTEVENT: future time will be 6.686605
MAINLOOP: data given to student: aaaaaaaaaaaaaaaaaaaa
TOLAYER3: seq: 1, ack 999, check: 2940 aaaaaaaaaaaaaaaaaaaa
TOLAYER3: packet being corrupted
TOLAYER3: scheduling arrival on other side
INSERTEVENT: time is 0.467849
INSERTEVENT: future time will be 4.558397
B_output : send_packet without ACK (seq = 1) : aaaaaaaaaaaaaaaaaaaa
START TIMER: starting timer at 0.467849
INSERTEVENT: time is 0.467849
INSERTEVENT: future time will be 10.467849
EVENT time: 4.558397, type: 2, fromlayer3 entity: 0
A_input : Packet corrupted (seq = 1). Drop
EVENT time: 6.686605, type: 1, fromlayer5 entity: 0
GENERATE NEXT ARRIVAL: creating new arrival
INSERTEVENT: time is 6.686605
INSERTEVENT: future time will be 10.258186
MAINLOOP: data given to student: bbbbbbbbbbbbbbbbbbbb
TOLAYER3: packet being lost
A_output : send_packet with ACK (ACK = 0, seq = 1) : bbbbbbbbbbbbbbbbbbbb
START TIMER: starting timer at 6.686605
INSERTEVENT: time is 6.686605
INSERTEVENT: future time will be 16.686605
EVENT time: 10.258186, type: 1, fromlayer5 entity: 1
GENERATE NEXT ARRIVAL: creating new arrival
INSERTEVENT: time is 10.258186
INSERTEVENT: future time will be 15.139011
MAINLOOP: data given to student: cccccccccccccccccccc
TOLAYER3: seq: 2, ack 999, check: 2981 cccccccccccccccccccc
TOLAYER3: scheduling arrival on other side
INSERTEVENT: time is 10.258186
INSERTEVENT: future time will be 19.470718
B_output : send_packet without ACK (seq = 2) : cccccccccccccccccccc
EVENT time: 10.467849, type: 0, timerinterrupt entity: 1
B_timerinterrupt : resend packet (seq = 1), data : aaaaaaaaaaaaaaaaaaaa
TOLAYER3: seq: 1, ack 999, check: 2940 aaaaaaaaaaaaaaaaaaaa
TOLAYER3: scheduling arrival on other side
INSERTEVENT: time is 10.467849
INSERTEVENT: future time will be 22.966888
B_timerinterrupt : resend packet (seq = 2), data : cccccccccccccccccccc
TOLAYER3: seq: 2, ack 999, check: 2981 cccccccccccccccccccc
TOLAYER3: scheduling arrival on other side
INSERTEVENT: time is 10.467849
INSERTEVENT: future time will be 25.851925
START TIMER: starting timer at 10.467849

즉 시나리오에 대한 결과 화면을 볼 때 프로그램 구현이 설계와 맞게 구현되었으며 손실이 난 경우 재전송, 손상된 경우 재전송, 초기의 999를 재외한 acknum 999를 받아 NAK로 인식하여 드롭하는 것, 패킷을 잘 받아 센더에게 보내줄 ack를 만드는 것을 볼 수 있으며 가장 중요하게 A 사이드 B사이드 서로 센더에서 리시버로 리시버에서 센더로 유기적으로 상호작용하며 패킷을 주고 받고 하는 것을 볼 수 있어 양방향 고백앤을 잘 구현했음을 알 수 있다.

● 고찰

이번 과제를 진행하면서 발생한 이슈는 다음과 같다. 또한 이것에 대한 고찰을 같이 서술하고자 한다.

1. NAK를 고려하지 않아 설계에 있어 NAK를 사용한 적이 없지만 초기의 ack가 없어서 acknum이 999인 패킷을 제외하고 프로그램 수행과정에 있어 acknum 999을 수신하는 경우에 대한 처리에 관한 이슈

NAK를 acknum=999로 가정하고 프로그램을 설계하였는데 가정만 할 뿐 따로 프로그램에서 패킷이 잘못되었을 때 NAK(acknum=999) 대한 전송 없이 반대 영역에서 해당 패킷에 대한 잘왔다는 ACK가 안오기 때문에 일정시간이 지나서 타임아웃이 발생하도록 하여 재전송을 구현하여 잘못된 패킷의 수신 문제를 해결하였다. 단지 (acknum=999)의 경우 프로그램 초기 패킷을 만들 때 ACK가 없어 (acknum=999)를 태워서 패킷을 보내주거나 같은 사이드에서 output을 연달아 해버려 가장 먼저의 output에 ACK를 태워 패킷을 보내주었기 때문에 다음 순서의 output에는 보낼 ACK가 없어 (acknum=999)를 태워 패킷을 보내주게 된다. 이때 만약 해당하는 패킷을 받았지만은 반대 영역의 센더가 잘 받았다는 ack를 보내주지 않아 계속해서 타임아웃이 발생하여 재전송한다면 이것을 받는 리시버 영역에선는 (acknum=999)의 해당하는 패킷을 이미 받았기 때문에 not expected sequence의 패킷이며 got (acknum=999)를 하게 되는데 프로그램 설계에 있어 전제조건으로 NAK를 acknum=999로 가정하였으므로 NAK를 수신한 것으로 처리하였다. 그리고 또한 NAK를 수신하여도 이를 무시하고 버려서 따로 프로그램 구현 요구 사항을 위반하지 않는다.