Assignment2 - SR(Selective Repeat) 프로토콜 구현

이름

2016032897 산업공학과 김기범

목적

Assignment 1에서 구현했던 **Put** 방식에서 비트에러, 전송지연, 패킷 유실 문제가 발생 시 **SR** 프로토콜을 이용하여 Server에게 In - Order하고 Packet Loss 없이 모든 Packet을 전송을 구현한다.

Assignment 1과의 차이

Assignment1에서 구현했던 방식에서 **CD, LIST, GET**방식은 동일하게 작용한다. 하지만 **PUT** Method를 이용할 때, Window의 개념이 적용된다. Window의 개념이 도입되면서 Window Size는 5이고, range는 0부터 15까지이고, **send base(보내고 있는 Packet에서 가장 왼쪽 값)**, **nextseqnum** 그리고 서버에서는 **receive base**(Ack한 packet중 가장 왼쪽 값) 개념이 적용된다. 각각은 이벤트 사건 발생 설명시에 같이 설명할 것이다.
Assignment 1에서는 Sequence를 이용하긴 했지만, 큰 의미가 있지 않아 모두 0으로 셋팅하고 전송했었다. 하지만, SR프로토콜을 위해서 Random Sequence번호가 적용된다. 따라서, Server에서 Client에게 ACK을 보낼 때도, Sequence 번호가 0이 아닌 Client가 Server에게 보냈던 Message의 **sequence** 번호를 전송한다. Sequence를 이용하기 위해서는 **Server에서도 Client의 첫 시작 Sequence**를 알아야한다. 따라서, 데이터를 보내기에 앞서 Client입장에서 첫 sequence 번호를 담은 제어메세지를 Server에게 전송하여 알려주도록 한다.
Assignment 1에서는 CheckSum이 모두 0x0000으로 **bit error**가 발생하지 않았지만, 2에서는 0xffff으로 전송되는 경우가 존재하여 유의해야 한다.
Assignment1에서는 Timewatch가 없었지만, 2에서는 크기가 16개인 long 배열을 생성하여 해당 **index**가 **sequence**번호이고 안에 담겨있는 **값**(long)이 파일을 전송했을 때의 **시간**을 가리키게된다. 전송 중이 아니라면 0으로 셋팅하여 비교대상에서 제외시킨다..

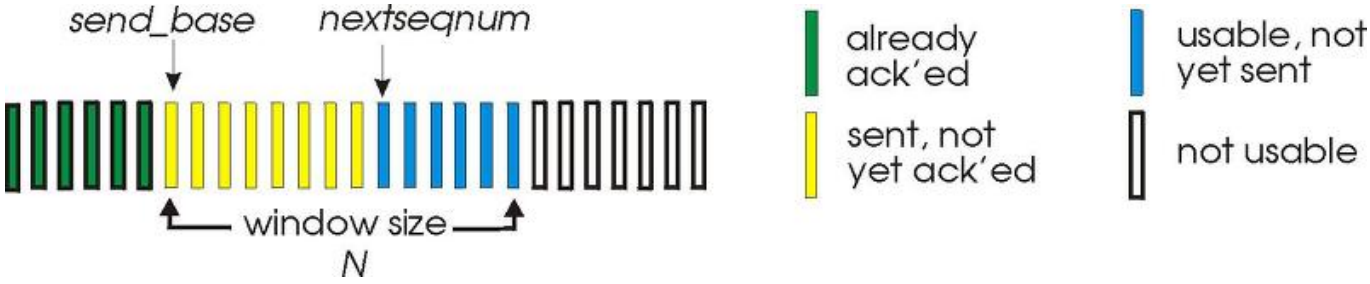
각 이벤트(Drop, Timeout, Bit Error) 사건 발생시의 구현 방식

참고사항 - 본 과제와 관련 있는 함수는 SocketClient.java(Client)에서
`sendData()`,`sendToServer()`,`timeCheck()`,`denoteAck()`,`packetMake()`,`InWindow()`이다.
Server에서는 DataSocket.java에서
`getFromClient()`,`bufferToUpperLayer()`,`checkRcvSequence()`,`makeAckPacket()`이다.

Client

이벤트 설명 전에 변수 설명이 필요하다. **send base**는 서버에게 보낸 Packet의 Sequence 번호 중 가장 왼쪽에 있는 번호(사진참고)를 의미한다. **nextseqnum**은 데이터를 보낼때마다 하나의 번호씩 증가하게 되는데, 이는 보낼 수 있는 데이터의 sequence 번호를 의미한다. **window size**는 아래 그림과 같이 보낼 수 있는 최대의 데이

터 갯수를 의미하면서, send base에서 window size를 더한만큼의 범위 내의 packet을 관리하게 된다.



Timeout시 보낼 데이터

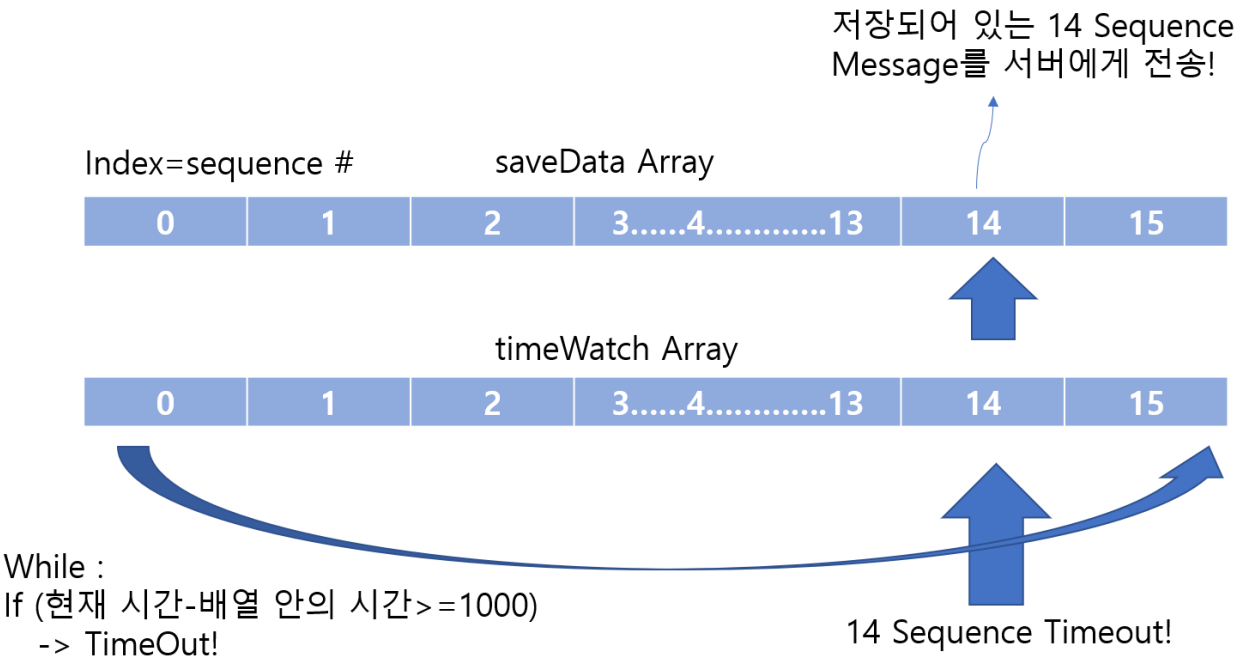
timeout 시간(1초) 이후에 데이터 메세지를 서버에게 보내기 위해 저장시킨다. 크기가 16*1005인 byte type의 **saveData** 배열을 형성한다. 16은 Sequence의 **range**의 크키이고, 1005는 **Packet Format**을 유지시켰을 때의 배열 크기가 1005인 **데이터메세지** 크기다.2차원 배열에서 첫 번째 index가 곧 sequence 번호이다. 이는 메세지가 **drop**되었거나, **timeout**을 고의적으로 시켰을 때, **Bit Error**시켰을 때, 해당 sequence 번호 index 값에 데이터 메세지를 저장시킨다. **Timeout**이 발생하고, 다시 서버에게 데이터 메세지를 보내야 될 때, **saveData**에 있는 데이터 메세지를 보내면된다.

타이머

타이머를 위해 sequence range가 0부터 16으로 크기가 16인 long type의 **timeWatch** 배열을 생성한다. 배열의 index는 곧 **sequence 번호**를 의미한다. 일반적으로는 서버로 보내지 않은 Packet에 대해 0으로 셋팅되어있지만, 데이터 메세지를 서버에게 보낼 때는 현재 시간을 `System.currentTimeMillis()`을 통해 저장시킨다. while() 사이클 한 번 돌 때마다, timeWatch 배열을 **탐색**하며 보낸 시간이 1초이상 지났는지 확인한다. 1초이상 지난 Packet이 존재한다면 배열의 index 값을 확인한다. 곧 sequence번호를 의미하기 때문에 **saveData** 배열에서 데이터 메세지를 가져와 서버에게 보낸다. 이때는 **bit error**가 발생하지 않는다는 가정하에 **checksum**을 0x0000으로 초기화 시킨 후 전송한다. 데이터 메세지를 보냈다면 다시 타이머를 돌린다. 이 때, **inWindowSend** 변수를 증가시키는데, 이는 Window 내에서 실제로 서버에게 보낸 **Packet 수**를 의미한다. 즉, **Drop**시킬 Packet을 전송하려고 할 때는 **inWindowSend**값을 증가시키지 않는다.

inWindowSend 변수를 좀 더 자세하게 설명하자면, **Transmission Delay**가 **RTT**보다 훨씬 짧다고 생각하여 Window 내에서 서버에 보낼 수 있는 Packet은 한 번에 전송시킨다. 이 때, 실제로 전송한 Packet의 수를 나타내는데, 서버에게 실제 Packet을 보낸 후, **inWindowSend**변수를 보고 파악 한 후, 그 갯수만큼 ACK Message를 받는다.
timeout 발생시 **delayPacketInWindow**의 변수를 하나 감소시킨다. **한 Window**내에서 **여러 Packet**이 drop되었다고 해보자. timeout 발생 시 timeout된 갯수만큼의 Packet을 한 번에 전송시키기 위한 변수다. 즉, 한 Window내에서 여러 packet이 Loss되고, Timeout이 발생한다면 Transmission Delay가 RTT보다 짧다고 생각하여 하나의 Packet을 보내고 ACK 메세지를 받는 것이 아니라, 한 번에 Timeout된 패킷을 전송한 후, 한 번에 ack

메세지를 받도록 한다.



Window 내에서 acknowledge하는 방법

Sequence번호가 0부터 15이므로 크기가 16인 **ackCheck** 배열을 생성한다. 만약, **in-Order** 순으로 ack 메세지가 오지 않았으면 즉, ack sequence 번호가 send base와 같지 않으면 ackCheck 배열에서 sequence번호에 해당하는 값을 **1**로 셋팅하면서 acknowledge했다는 것을 표시한다. default 값으로는 0으로 acknowledge하지 않았다는 것이다.

ACK 메세지를 받았을 때

서버로부터 ACK메세지를 받으면, `denoteAck()`함수를 호출하게 된다. 여기에서는 서버가 보낸 ack message의 Sequence 번호를 확인 한 후 **타이머**를 0으로 셋팅하면서 타이머를 끈다. 이는 timewatch의 index가 **sequence 번호**인 배열 값을 0으로 셋팅하는 것과 같다.(**타이머** 참조) 그 후, ack sequence 번호가 **send base**와 같은지 확인한다. send base와 같다면, in-order 순으로 ack message가 왔다는 것을 의미하므로 **achCheck**배열을 통해 acknowledge하지 않은 최소의 Sequence번호를 확인한 후, send base로 지정한다. 다음 sequence 번호의 acknowledge했는지 확인하는 과정에서 다음 sequence가 **acknowledge**했다면, 서버에게 데이터메세지를 보낸다. 이는 send base가 **오른쪽으로** 한 칸 이동했고, **Window내**에 데이터를 보낼 수 있는 sequence번호가 생겨서 데이터를 서버에 보냈다는 것을 의미한다. 만약 ack sequence 다음에 acknowledge한 sequence가 **여러개** 존재한다면, 이는 winodw가 오른쪽으로 여러 번 움직이는 것과 같고, 데이터를 보낼 수 있는 sequence 번호가 여러개 생긴다는 것과 같다.
send base와 같지 않다면, ack sequence가 Window내에 있는지 `InWindow()`함수를 통해 확인 한 후, **ackCheck** 배열에서 sequence번호의 index 값을 **1**로 셋팅하며 acknowledge했다는 것을 표시한다. **ackCheck**배열은 위의 "Window 내에서 acknowledge하는 방법"을 참고한다.
Data Channel를 서버로부터 받은 **ack메세지** 수와 Client가 Server에게 보내야하는 **데이터메세지** 수를 비교하여 유지시킨다. 실제로 서버에게 보내야 하는 데이터 메세지의 수(**데이터 청크를 1000으로 나눈 몫+1**)와 서버로부터 실제로 받은 ack의 수(**realAckNum**)를 비교하여 유지시키는데, ack message를 받았을 때, **realAckNum**을 증가시켜 하나의 데이터메세지를 서버가 제대로 받았다는 것을 표시한다. 만약 데이터 Chunk가 4560byte라 해보자. 그러면 1000으로 나눈 몫+1인 5개의 ACK Message를 받으면 Data Channel은 해제시킨다. 이런 가정을 할 수 있는 이유는 현재 과제에서는 서버로부터 오는 ACK Message는 정상적으로 데이터 메세지를 받았을 때 보내기 때문이다. TimeOut의 상황은 약간 달라지는데 아래에서 자세히 설명할 것이다.

Client에서 Server에게 데이터를 보낼 때

Client에서 Server로 데이터를 보낼 때, Window Size이상으로 데이터를 보내지 못한다. 이 때, **sendCount** 변수를 이용하여 Client에서 서버로 보낸 데이터메세지 갯수를 확인한다. **sendCount**변수는 Packet에 이벤트가 생겼을 때도 증가하여 Client는 보냈다고 생각하게 한다. Data를 보내기 전에 고의적으로 Drop해야 하는지, timeout 시켜야하는지, bit Error를 발생시켜야 하는지 확인해야 한다.

- **Drop**
  Drop에 해당하는 Packet을 보내야 할 차례가 온다면, 실제로 서버에게 **보내지 않고**, 몇 번째의 Packet인지의 **howmany**변수, Client에서 서버에게 보냈다고 생각하는 **sendCount**변수를 증가시켜 Client입장에서 보냈다고 생각하게 한다. 이 때, **saveData**에 sequence번호에 해당하는 배열 index에 데이터 메세지를 저장시키고, 타이머를 `System.currentTimeMillis()`를 통해 킨다. 그리고, 같은 Window 내에서 **Drop**되는 Packet이 여러개 존재할 수 있다. **delayPackInWindow**변수를 증가시켜 한 Window내에서 몇 개의 Packet이 **drop**됐는지 알 수 있다. 이 변수의 용도는 위의 **타이머**를 통해 알 수 있다.
- **timeout**
  Timeout 이벤트는 timeout을 의도적으로 발생시키기 위해 **2초** 뒤에 전송시키는 것이다. 2초 뒤에 전송시키기 위해서는 **ArrayList**의 **delaySendData** 변수에 sequence 번호, **시작시간**, 데이터 메세지를 저장시켰다. Timeout Packet이 전송 될 때, 마찬가지로 Timer를 킨다. 타이머를 확인 할 때, 잠시 **delaySendData**의 크기를 확인한다. size가 0이상이라면 timeout을 의도적으로 발생시켜 2초 뒤에 전송될 Packet이 존재한다는 것을 의미한다. **delaySendData**에 2초 이상 지난 Packet이 있는지 확인 하고 있으면 Server에게 전송한다. 타이머는 위에서도 언급했지만, while()에서 데이터 메세지를 먼저 서버에게 보내고, 배열에 있는 타이머를 한 번씩 확인한다.
  -> Timeout으로 2초 뒤에 **재 전송**하고 ACK Message를 받기 위해서는 데이터 채널을 이전보다 오래 유지시켜야 한다. 이를 위해 timeout 시켜야되는 Packet의 수만큼 서버로부터 AckMessage를 더 받게하면 된다. "**ACK 메세지를 받았을 때**"를 참고하면, 데이터 채널을 유지시키는데 실제로 받은 ACK Message의 수가 중요하다. 따라서, **realAckNum**(실제로 받은 ACK Message의 수)를 timeout 시켜야하는 Packet의 수만큼 늘리면 된다.
  이 때, Server에서도 **Timeout** 시켜야하는 Packet 수를 알아야 더 데이터 채널을 유지실 수 있기 때문에, 데이터 교환을 하기 전에 **Timeout Packet의 수**를 제어 메세지를 통해 전달한다.
- **Bit Error**
  Bit Error의 경우 정상적으로 보내는 경우와 같지만, CheckSum을 0xFFFF으로 변경한다. **Bit Error**를 의도적으로 변경시킬 Packet의 수는 **bitErrorList**에 담겨있다. 만약, 서버에게 보낼 데이터 메세지가 checksum이 변경되어야 할 차례의 Packet이면 변경한다. 하지만, ACK 메세지는 오지 않고 타이머가 정지되지 않기 때문에 **Timeout**이 발생하여 재전송하게 된다. 재전송할 때 Timeout은 위의 "**타이머**"를 보면 된다.

Server

Server에서는 Client에 비해 간단하다. Client와 마찬가지로 **데이터 Channel**를 유지시키는데 Client에게 보내는 ACK 메세지의 수(**count**변수)와 받아야하는 데이터의 수(**share**변수)를 비교한다. 만약 Client에게 보내는 ACK Message 수와 Client로부터 받아야하는 데이터 수가 같게 된다면, 더 이상의 연결은 무의미하여 해제시킨다. Client에서도 설명했지만, Server로부터 2개의 제어메세지를 먼저 받아야한다. 하나는 Client가 시작하는 **Sequence** 번호이고, 다른 하나는 **Timeout Packet**(일부로 2초뒤에 전송할 Packet)의 수를 받는다. Server에서는 Timeout Packet 수만큼 ACK Message를 Client에게 더 보내도록 하고 데이터 채널을 유지시킨다. 받은 Sequenece 번호는 Server에서 rcv base 번호로 지정한다. rcv base는 window size 5 이내에서 Client의 받은 ack message의 가장 왼쪽에 있는 sequence 번호를 의미한다.

BitError 발생시

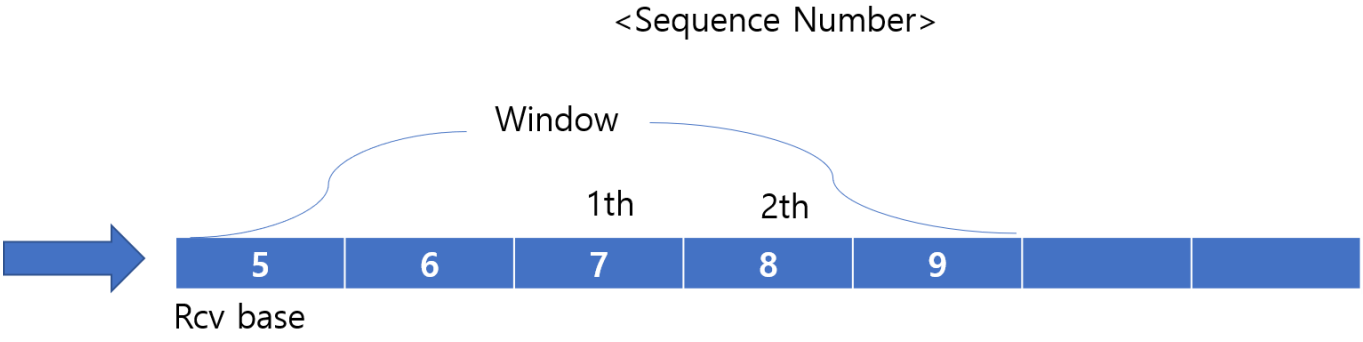
Client로부터 데이터메세지가 오면 맨 먼저 **checkSum**을 확인 한다. checkSum이 데이터메세지에서 어디 있는지는 Assignment 1 보고서를 참조하면 된다. checkSum이 0xFFFF라면 **Bit Error**가 발생한 것으로 생각하여 Ack Message를 보내지 않고, Client가 **timeout**이 발생하여 재 전송하도록 유도한다.

데이터 메세지 Sequence 번호 검사

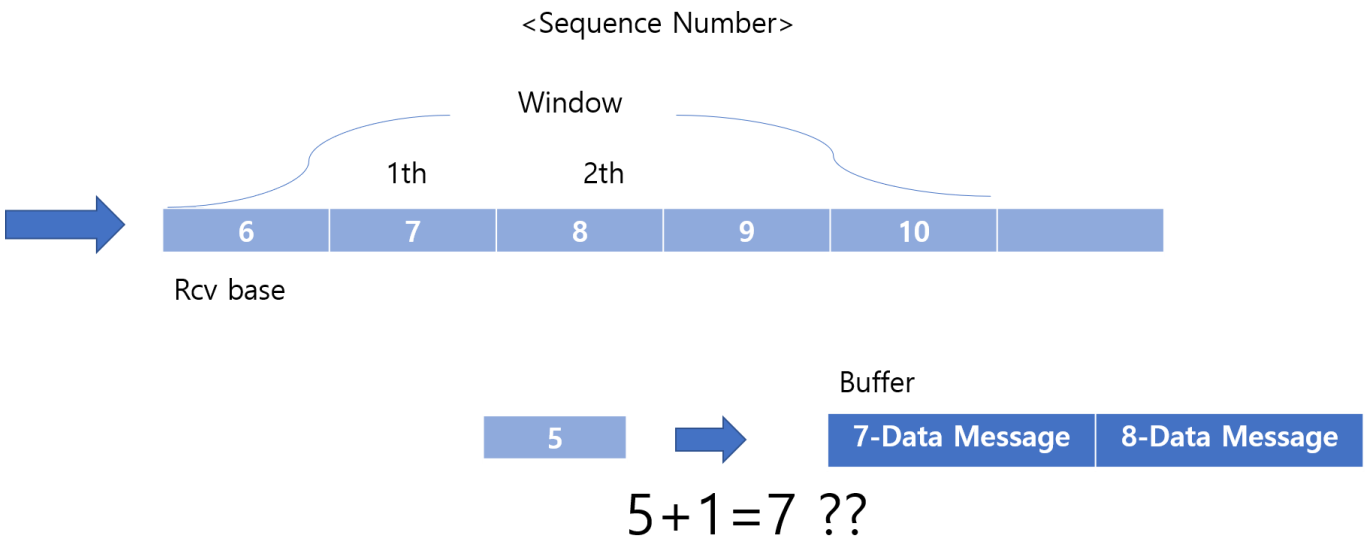
만약 In-Order로 왔으면 Upper Layer에 데이터를 바로 전송하면 된다. **In Order**로 데이터메세지가 온다는 것은 rcv base와 데이터 메세지의 Sequence번호가 같다는 것을 의미한다. **rcv base**번호가 아닌 다른 Sequence 번호로 데이터 메세지가 온다는 것은 **Out order**를 의미한다. in-order로 데이터 메세지가 왔으면 다음 sequence부터 acknowledge하지 않은 sequence를 찾아 **rcv base**로 설정한다. acknowledge 검사는 아래의 Out-Order에서 설명할 것이다.
이론적으로는 **Upper Layer**가 **Application Layer**이지만 여기에서는 Upper Layer를 **totalData**라는 배열로 모든 데이터 Chunk로 채워져야하는 배열이라고 생각하면 된다. Upper Layer에 데이터를 전송하는 것은 **totalData**에 데이터를 순서대로 넣는 것을 의미한다. 순서대로 넣기 위해서 **saveCount**의 변수를 이용하여 **totalData**에 데이터가 들어갈 때마다 값을 증가시켜 차례대로 data가 들어가지도록 한다.
만약 **Out- Order**로 왔으면 **checkAck**배열에서 해당하는 Sequence 번호의 index 값을 1로 설정하여 **acknowledge**했다는 것을 표시한다. **checkAck**배열은 해당 Sequence번호가 Acknowledge했는지 확인하는 배열로 Client에서 **achCheck**배열과 동일한 기능을 한다. 그리고 버퍼에 넣어놓는다. 버퍼의 기능은 아래에서 자세히 설명할 것이다. 버퍼에 넣어놨으면 Client에게 ACK 메세지를 전송한다. 이 때, **count**변수를 증가시켜 Data Channel를 유지할지 결정한다. Server 초기에 설명 했듯이, 만약 **count**를 증가시켰더니 Client로부터 받아야하는 데이터수와 같게된다면 연결은 무의미하여 해제시킨다. 단, **Timeout Packet**이 존재한다면, 그만큼의 ACK Message를 더 보내야한다.

Buffer

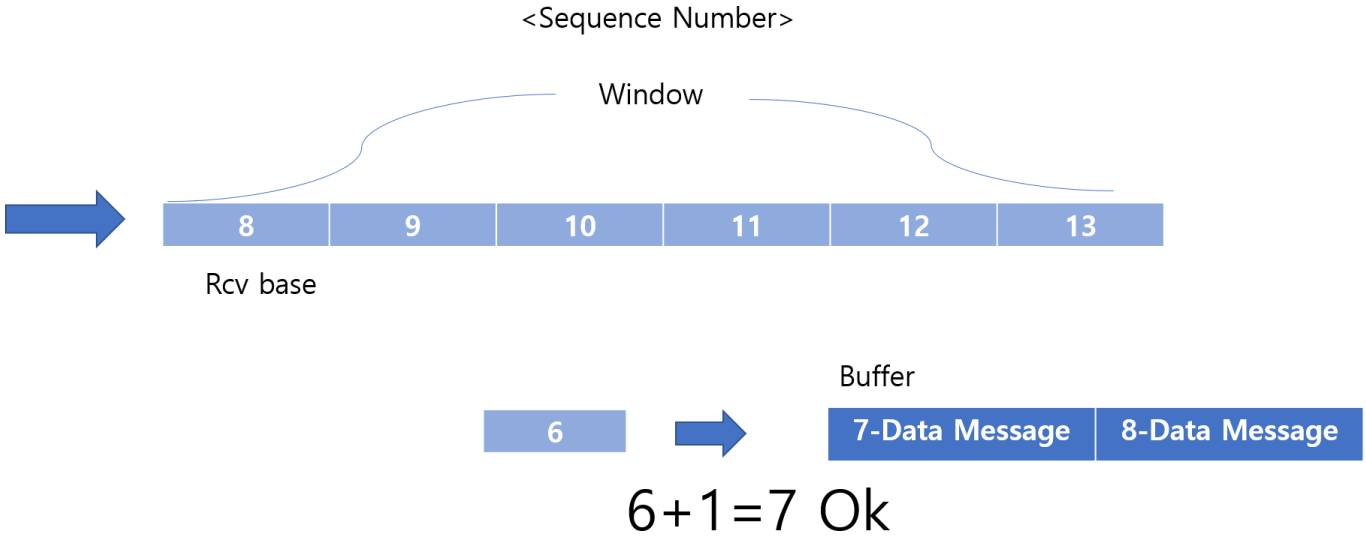
Buffer는 **Out-Order**로 데이터메세지가 왔을 때 **임시** 저장되는 공간이다. Buffer를 ArrayList의 형태로 **buffer**변수를 통해 구현했다. ArrayList에는 데이터메세지 그 형태 자체, 즉, byte의 길이가 1005인 배열 객체가 저장된다. 이해를 위해 아래의 그림을 참고해보자. 아래의 칸에는 Sequence번호를 나타낸 것이고, nth는 Window에서 n번째로 온 ACk Sequence다. 나머지 5,6,9는 **drop**, **timeout**, **biterror** 이벤트로 Acknowledge하지 않았다. Client에서는 Sequence번호 순서대로 데이터 메세지를 보낼 것이고, 번호 순서대로 이벤트가 발생할 것이다. 즉, 이벤트가 발생하더라도 5,6,9순서로 Client에서 데이터메세지가 재전송될 것이다.



위의 그림에서는 Out-Order로 데이터 메세지가 오기 때문에 7번, 8번 순서대로 데이터메세지가 **buffer**라는 ArrayList에 넣어질 것이다. Timeout으로 Client에서 Sequence 5번의 데이터메세지를 보냈다고 해보자.



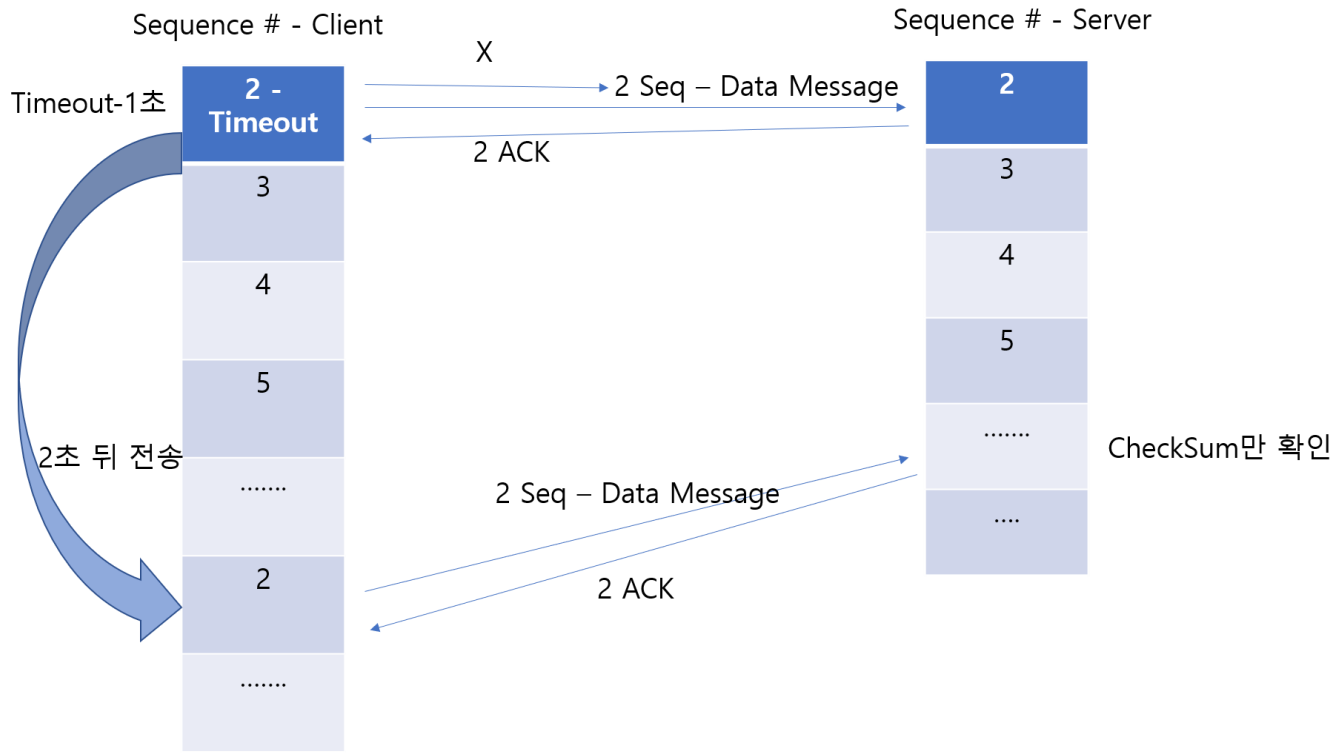
rcv base에 맞는 Sequence가 왔으므로, 5번의 데이터메세지를 Upper Layer에 즉, **totalData**배열에 **Data Chunk**를 복사시킨다. 그리고 rcv base에 해당하는 sequence 메세지를 받았으므로 acknowledge하지 않은 다음 sequence인 6번이 rcv base가 될 것이다. 이 때, 5번의 Sequence는 **Buffer**에 데이터메세지가 존재하는지 확인한다. 이는 Out Of Order로 데이터메세지가 온 것이 있는지 확인하는 작업과 같다. 존재한다면, **Buffer** 첫 번째의 Sequence가 자기 자신 5번 Sequence 다음의 데이터메세지(6번)인지 확인한다. 아니라면 Pass하고 Client에게 **ACK** Message를 보낸다.



Client에서 서버로 Timeout으로 6번의 데이터 메세지를 보냈다고 해보자. Server에서는 rcv base 번호에 맞는 Sequence의 데이터메세지가 왔으므로 먼저 **totalData**배열에 자신의 Data Chunk를 복사시킨다. 그리고 Acknowledge하지 않은 다음 Sequence인 **8**번이 **rcv base**가 된다. 여기에서 **Buffer**라는 ArrayList에 데이터메세지가 존재하는지 확인한다. 만약 있더라면, Buffer의 첫 번째 Sequence와 비교한다. 그림과 같이 Sequence가 6이 다음이 Buffer의 첫 번째 데이터메세지의 Sequence라면 7번의 데이터메세지도 **totalData**배열에 복사시켜 Upper Layer에 올린다. 그 다음에도 만약 버퍼에 7번 Sequence 다음인 8번 Sequence 번호의 데이터메세지를 갖고있으면 **totalData**배열에 복사시킨다. **totalData**에 복사시킨 버퍼의 데이터메세지는 삭제시킨다. 만약 8 Sequence의 데이터메세지가 아니라 9 Sequence의 데이터메세지라면 Pass하고 기존 6번의 **Ack Message**를 Client에게 보낸다.
totalData에 저장시킬 때마다 **saveCount**변수를 증가시켜 순서대로 Data Chunk가 복사되게 한다.

TimeOut

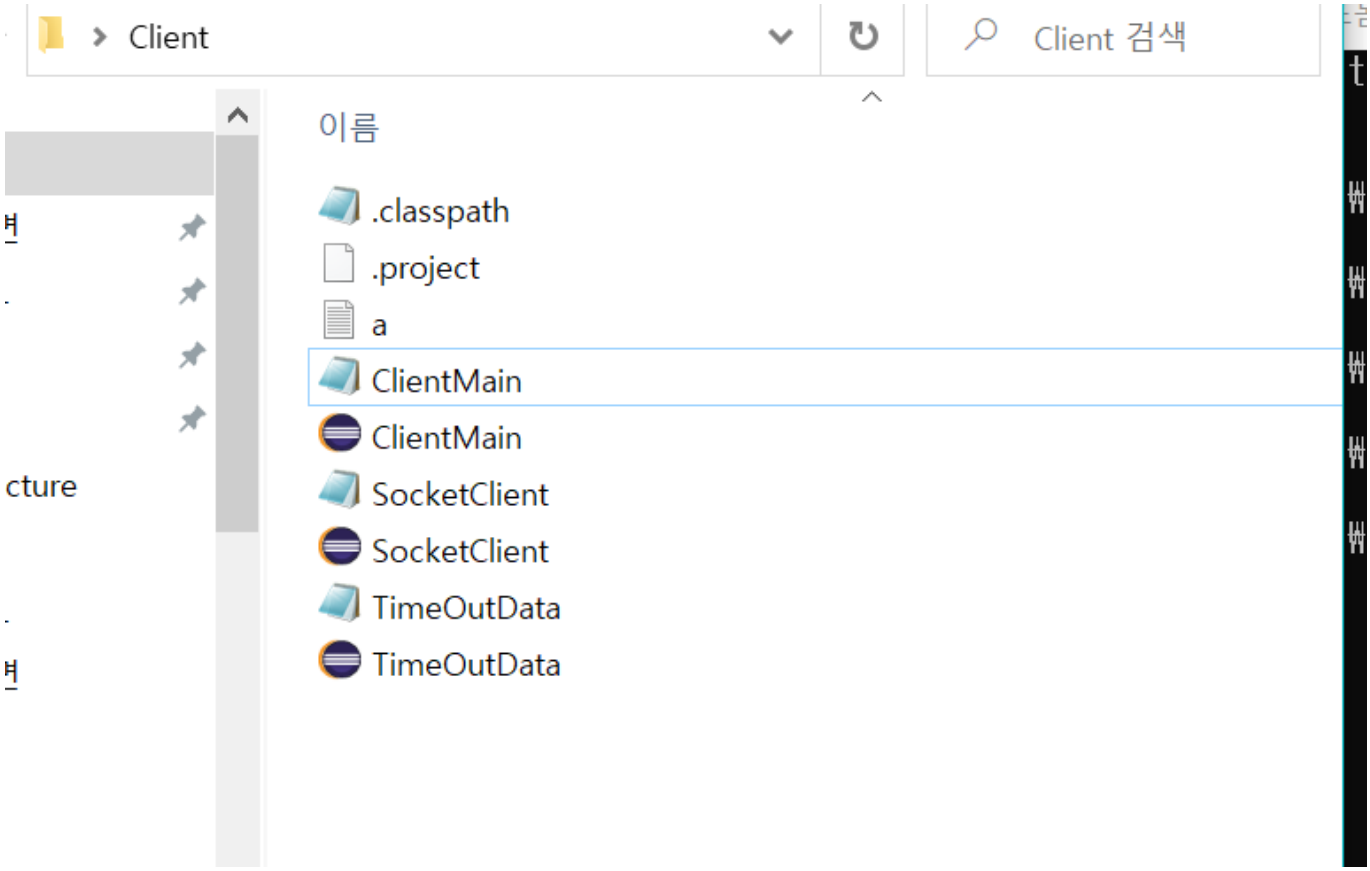
Drop과 bit error 이벤트는 Client에서 Server로 한 번의 데이터 메세지만 전송된다. 하지만 Timeout 이벤트는 총 2번의 데이터 메세지가 서버에게 전송된다. 한 번은 Timeout(1초)이 발생하여 전송하게 되는 것이고, 다른 한 번은 보냈다고 생각한 시점의 **2초** 뒤에 재전송하여 서버에게 전송된다. 이 때는 Server에서는 초기에 말했듯이 Client에게 받은 **Timeout Packet**의 수만큼 더 받도록 Data Channel를 유지시킨다.
Timeout(1초)이 발생하여 전송된 데이터메세지는 정상적으로 처리하여 **totalData**에게 보내지거나 **Buffer**에게 임시 저장될 것이다. 하지만 고의적으로 **2초** 뒤에 전송시킨 Packet은 **Checksum**만 확인 한 후, Client에게 ACK Message를 전송한다. 이 때, 2초 뒤에 전송한 데이터 메세지임을 표시하기 위해 cmd에서 "Sequence 번호 resend"로 표시했다.



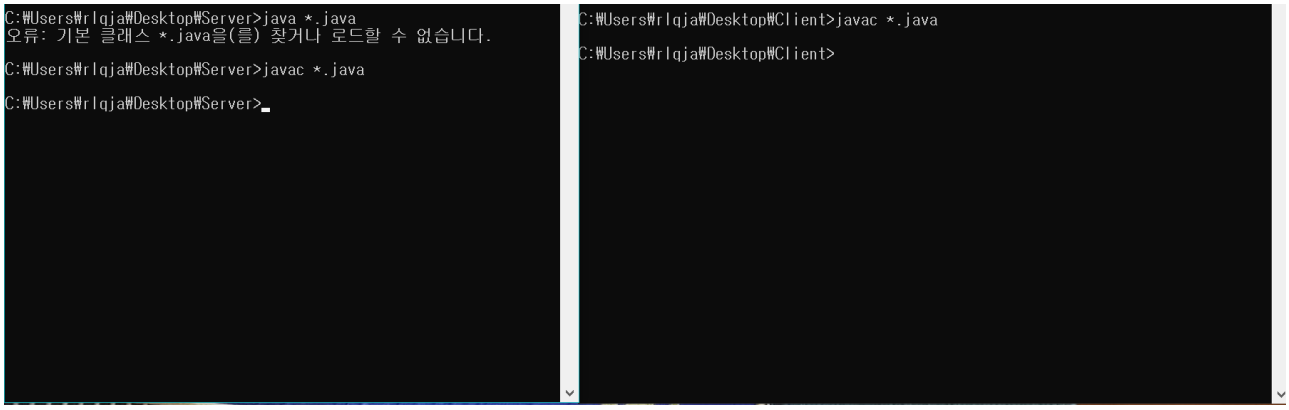
구현 절차

구현은 Window 10의 cmd에서 진행한 것이다. 서버에 해당하는 Java 파일은 **ServerMain.java, DataSocket.java, ChangeDirectory.java**로 3개로 구성된다. Client에 해당하는 Java 파일은 **ClientMain.java, SocketClient.java, TimeOutData.java**로 3개로 구성되어있다. 그러고 서버에 해당하는 Java 파일을 하나의 디렉토리에 넣어놓고, Client에 해당하는 Java 파일은 다른 하나의 Directory에 넣어놓는다.

실행 전에 PUT으로 Client에서 Server로 옮길 데이터를 **Client Directory**에 넣어놓는다.

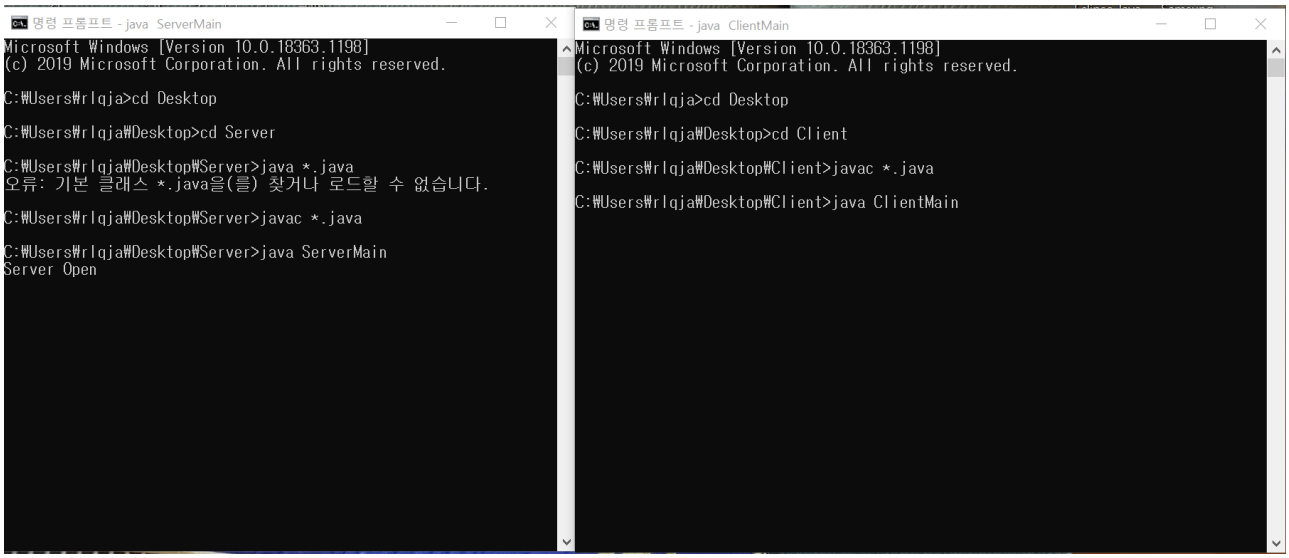


1. cmd창을 2개를 연다. 하나는 Server용이고 하나는 Client용이다. 하나의 cmd에서는 Server java 파일이 담긴 곳으로 directory를 옮긴다. 다른 하나의 cmd는 Client java 파일이 담긴 곳으로 Directory를 옮긴다. 옮긴 후, 아래의 사진과 같이 **Compile**한다. 컴파일은 javac *.java를 통해 모두 컴파일시킨다.(여기에서는 Client Directory명으로 "Client"를, Server Directory명으로 "Server"를 썼다.)

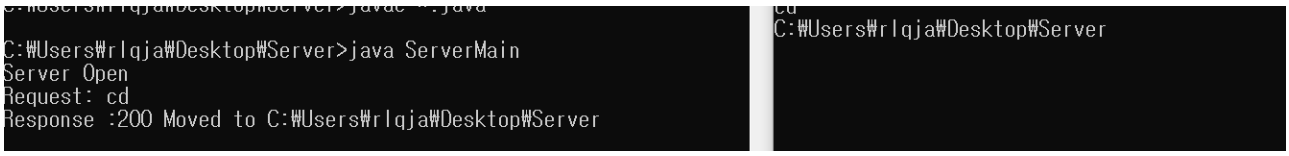


- 만약!! 실행도중 에러가 생긴다면 ctrl+c를 통해 취소한 후, 2번부터 다시 실행하면 된다.

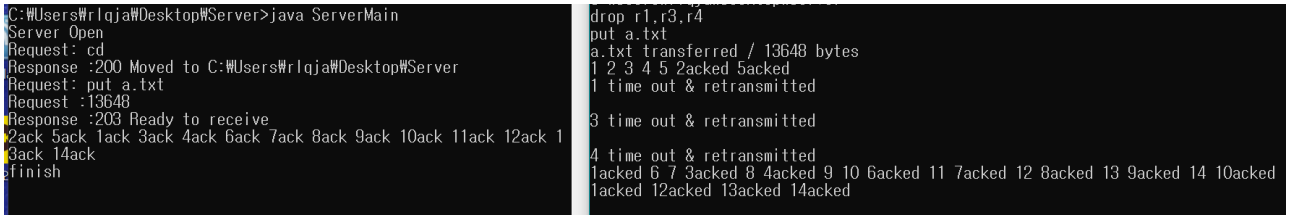
1. Server를 **java ServerMain**을 통해 실행시키고, Client는 **java ClientMain**을 통해 실행시킨다. 그러면 아래의 그림과 같은 상황일 것이다.



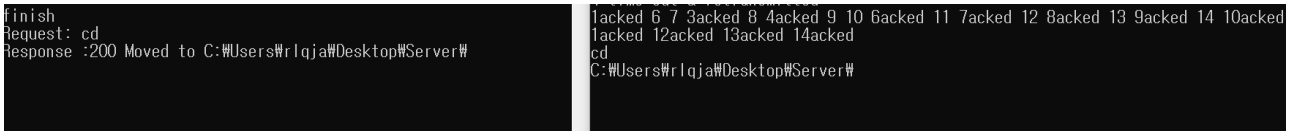
2. 연결이 잘 되어있는지 **CD**를 통해 확인한다.



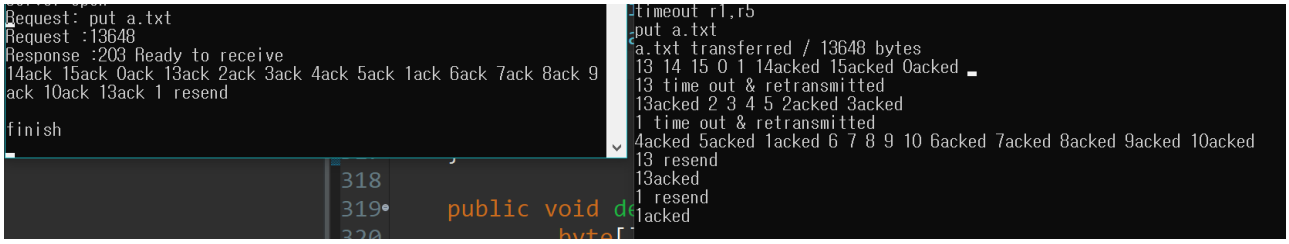
3. Event에 해당하는 Method를 입력한다. e.x) 여기서는 drop r1,r3,r4를 해볼 것이다. **drop r1,r2,r3**를 입력하고 Enter를 누른 후, PUT **보낼파일명**을 입력하고 Enter를 누른다. 여기서는 a.txt를 보낼 것이다. 그러면 아래의 사진과 같이 보여질 것이다. 왼쪽이 Server창이고, 오른쪽이 Client창이다. Server에서는 받은 Sequence 번호를 입력하게 했으며, Client에서는 그냥 숫자는 **보낸 데이터메세지**의 Sequence 번호, 뒤에 acked 붙은 것은 서버로부터 받은 **ACK message**의 Sequence 번호, "# time out & retransmitted"는 event로 인해서 1초뒤에 보낸다는 입력이다.



4. 그러고 다시 연결이 잘 되어있는지 확인한다. 생략 가능하지만, 원활한 수행을 위해서다.



5. 다른 Event를 해볼 것이다. TimeOut Event를 예로 들면, **timeout r1,r5**를 입력할 것이다. 그러고 put "파일명"을 입력한다. 여기서 PUT은 **대소문자** 구분하지 않아도 된다. 아래의 그림에서 timeout이 발생하여 재전송되는 것은 **# time out & retransmit**으로 표시한다. 그리고 2초 뒤에 재 전송되는 것은 **# resend**로 표현하여 고의적인 2초 지연 전송을 표시한다.
Server에서는 받은 **데이터메세지**의 Sequence를 순서대로 표현한 것이다.
여기서도 왼쪽이 Server창, 오른쪽이 Client 창이다.



6. 마지막으로 bit error를 해볼 것이다. 여기서는 예를 들어, **biterror r3,r8**을 해볼 것이다. enter 친 후, **put 파일명**을 입력한다. 여기서도 왼쪽이 Server창, 오른쪽이 Client 창이다. Client 입력값과 Server의 입력값은 4번에서 drop Event시에 입력되는 표시 의미가 같다.