컴네 중간 2

▼ Transport Layer

- 4계층인 전송 계층은 서로 다른 호스트에서 실행 중인 프로세스 사이의 논리적인 통신을 제공한다.
    - 논리적인 통신이란, 통신이 진행되는 호스트가 실제로는 멀리 떨어져 있더라도 직접 연결되어있는 것 처럼 보이게 해준다는 의미이다.
- 전송 프로토콜(TCP, UDP)은 종단 시스템에서 두 가지 역할을 수행한다.
    - sender: 프로그램의 메시지를 세그먼트들로 분리하고, 네트워크 계층으로 넘긴다.
    - receiver: 세그먼트들을 메시지로 다시 조립하며, 이를 응용 계층으로 넘긴다.
- 이 과정을 예시로 들자면
    - 열 둘의 아이가 다른 집의 다른 아이 열 둘에게 편지를 보낸다.
    - 집 = 호스트
    - 아이 = 프로세스
    - 메시지 = 편지봉투 내부의 편지.
- 네트워크 계층이 호스트 사이의 논리적인 통신을 제공한다.
    - 전송 계층은 다른 호스트 내부의 프로세스 사이의 통신을 제공한다.
    - 네트워크 계층의 서비스에 의지하며, 또한 강화시킨다.
- Sender는 다음과 같은 행동을 한다.
    1. 응용 계층에서 메시지를 전달받는다.
    2. 세그먼트의 헤더 값을 결정한다.
    3. 세그먼트를 만든다.
    4. 세그먼트를 IP(네트워크 계층)에 넘긴다.
- Receiver는 다음과 같은 행동을 한다.
    1. IP로부터 세그먼트를 전달받는다.
    2. 헤더 값을 체크한다.

3. 응용 계층의 메시지를 추출한다.
4. 소켓을 통해 프로그램에 메시지를 demux한다.

- 전송 계층의 프로토콜은 두 종류가 있다.
  - TCP : Transmission Control Protocol
    - 신뢰가능한, 순서가 유지되는 전송
    - 혼잡제어(Congestion control)
    - 흐름제어(Flow control)
    - 연결의 setup
  - UDP : User Datagram Protocol
    - 신뢰불가능한, 순서없는 전송
    - 전송같은 무조건 필요한 기능만 수행하고, 오류 제어 등의 다른 부가적인 확장이 없는 프로토콜.
  - 이용할 수 없는 서비스
    - 딜레이 보장, 대역폭 보장이 없음
    - = 처리율의 보장도 없음!

▼ Multiplexing and Demultiplexing(다중화와 역다중화)

- sender = mux
  - 여러 소켓으로부터 추가 정보를 얻는다(목적지 주소등)
  - 이 정보를 전송한 데이터의 헤더에 추가한다.
- receiver = demux
  - 수신된 데이터를 적절한 소켓으로 전달하기 위해 헤더 정보를 사용한다.
- 역다중화가 작동하는 방법.
  - 호스트가 IP 데이터그램을 수신한다.
    - 각 데이터그램은 출발, 도착지의 IP 주소를 가진다.
    - 각 데이터그램은 한개의 전송계층 세그먼트를 수송한다.
    - 각 세그먼트는 출발, 도착지의 포트번호를 가진다.
  - 호스트는 IP 주소와 포트번호를 사용해 세그먼트를 적절한 소켓에 전달한다.

- 연결 없는 역다중화(UDP의 통신)
  - 소켓이 만들어지면, host-local 포트번호만 가지고 있다.
    - DatagramSocket mySocket1 = new DatagramSocket(12534);
  - 호스트가 udp 소켓에 보낼 데이터그램을 만들때, 목적지의 IP주소와 포트 번호를 구체화해야한다.
  - 만약 수신 호스트가 udp 세그먼트를 받으면
    - 목적 포트번호를 확인한다.
    - 그 포트번호에 해당하는 소켓으로 세그먼트를 보낸다.
  - IP/UDP 데이터그램이 서로 다른 출발 IP 주소와 다른 출발 포트넘버를 가진다고 해도, 동일한 목적지 포트 번호를 가지고 있다면 같은 소켓으로 보내진다.
- 연결 지향형 역다중화(TCP의 통신)
  - TCP 소켓은 총 4-튜플로 구분된다.
    - 출발 IP 주소
    - 출발 포트 번호
    - 목적 IP 주소
    - 목적 포트 번호
  - 따라서 수신자는 4개의 값을 모두 사용하여 적절한 소켓에 세그먼트를 보낸다.
  - 서버는 동시에 여러 TCP 소켓들을 지원해야할 수도 있다.
    - 당연히 각 소켓은 4개의 값으로 구분된다.
    - 각 소켓은 다른 클라이언트와 연결된다.
- 따라서
  - UDP : 도착 포트번호만으로 역다중화
  - TCP : 총 4개를 사용해서 역다중화.

▼ 비연결 전송 : UDP(RFC 768)

- 장식 없음, 뼈대만 있는 인터넷 전송 프로토콜
- 최선을 다하는 서비스를 제공함 따라서
  - 세그먼트가 증발해도 모름

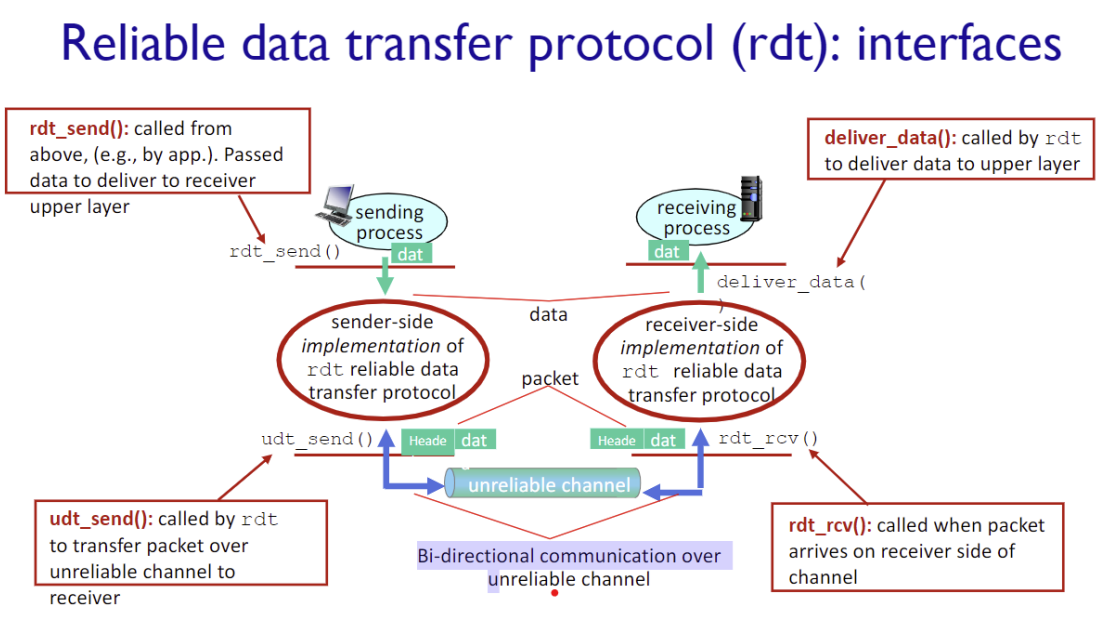
    - 순서가 뒤죽박죽일 수 있음.
- 연결이 없다는 것은.
    - UDP 송 수신자 사이의 연결(handshaking)이 없음
    - 각 UDP 세그먼트는 다른 세그먼트와는 완전히 독립적으로 다뤄짐.
- 그럼 UDP의 존재 이유가 뭣이니?
    - 연결이 없다.
        - 연결은 RTT 지연을 늘릴 수도 있다.
    - 단순하다.
        - 송수신자가 연결 상태를 가질 필요가 없다.
    - 헤더의 크기가 작다.
        - 8바이트 헤더를 가진다. TCP는 20 + A의 헤더 크기를 가진다.
    - 혼잡 제어가 없다.
        - UDP는 원하는 만큼 빠르게 전송할 수 있으므로, 혼잡 제어가 발생하지 않는다.
- UDP의 사용 장소
    - 멀티미디어 스트리밍(손실 내성, 전송률 민감)
    - DNS
    - SNMP
    - HTTP/3
        - UDP를 사용하면서도 신뢰가능한 전송이 필요하다.
        - 이럴 땐 응용 계층에서 신뢰성과 혼잡제어를 추가해준다.
- UDP : 전송 계층 행동
    - UDP 송신자는 다음과 같은 행동을 한다.
        - 응용 계층에서 메시지 받기.
        - UDP 세그먼트의 헤더 값을 결정
        - UDP 세그먼트 생성
        - 세그먼트를 IP에 넘긴다.
    - UDP 수신자는 다음과 같다.

        - IP에서 세그먼트를 받는다.
        - UDP 헤더 값의 checksum을 확인한다.
        - 응용 계층 메시지를 추출하고
        - 메시지를 소켓을 통해 프로그램에 역다중화한다.
- UDP 세그먼트는
    - 16비트의 출발, 도착 포트 번호가 있다.
    - 길이와 checksum이 있다.
    - 길이는 가변적인 세그먼트의 끝을 알기 위해 필요하고
    - checksum은 중복 검사(오류 검출)을 위해 필요하다.
- checksum
    - 전송된 세그먼트 내부의 오류를 검출한다.
    - 송신자는
        - 세그먼트가 16비트 정수를 나열한 것이라고 생각한다.
        - 이 세그먼트 합의 보수를 checksum으로 저장한다.
    - 수신자는
        - 받은 세그먼트에서 checksum을 계산한다.
        - 계산한 checksum과 세그먼트의 checksum을 비교한다.
            - 다르다면 = 오류 검출
            - 같다면 = 오류가 없을 확률이 높다(있을 수도?)
                - 오류가 났어도, checksum이 같을 가능성은 있다.
- 요약하자면 UDP는
    - no frills한 프로토콜이다.
        - 세그먼트가 손실, 서순이 발생할 수 있다.
        - 최선을 다한다 = 보내놓고 잘 가길 기도한다.
    - 그래도 장점이 있다.
        - 셋업이나 연결이 필요 없다 = RTT가 추가로 발생하지 않는다.
        - 네트워크 서비스가 손상되어도 기능할 수 있다.
        - 헤더의 크기가 작다.

        - checksum을 통해 신뢰성을 약간 향상할 수 있다.
    - UDP 위에 응용 계층에서 추가적인 기능을 만들수도 있다.
        - HTTP/3의 경우, 신뢰성이나 혼잡 제어등을 추가 구현한다.

▼ RDT

- 신뢰가능한 서비스의 추상적 접근
    - reliable channel로 통신하면 됨.
- 실제 구현
    - 송 수신측의 RDT 프로토콜이 unreliable한 채널로 통신하게 됨
- 여기서 RDT 프로토콜의 복잡성이 나옴.
    - 신뢰할 수 없는 채널의 특징에 따라 복잡도가 달라짐.
    - lose, corrupt, reorder data....등의 특징이 있음.
- 또한 송신자와 수신자는 서로의 상태에 대해 전혀 알지 못함.
    - 메시지가 수신됐는지 여부는 알 수 없음!.
- RDT의 인터페이스는 다음과 같다.

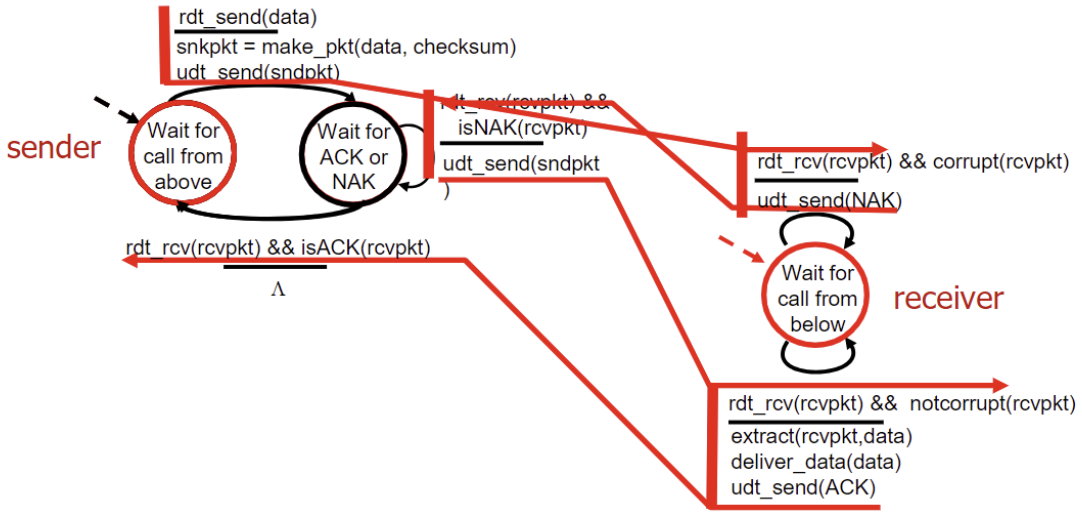


- rdt_send() → udt_send() → rdt_rcv() → deliver_data()
- FSM을 사용해서 송, 수신자를 구체화한다.
    - 이벤트 (전이를 발생시키는 이벤트)

    - 액션으로 구성된다.(전이를 위해 일어나는 액션들)
  - reliable = 데이터의 변형이 전혀 없다
    - 이상적임. 거의 없는 경우.
    - 비트는 결과적으로 깨지기 마련!
    - 따라서 신뢰성을 보장해주기 위한 4계층 프로토콜(transfer protocol)을 넣어 주겠다.
- rdt 1.0
  - 이상적인 상황 = 신뢰 가능한 채널에서 신뢰 가능한 전송을 하겠다.
    - no bit error (네트워크는 바이트를 쓰지 않는다)
    - no loss of packets
  - 전송자와 수신자가 서로 다른 FSM을 가진다.
    - 전송자는 채널에 데이터를 보낸다
    - 수신자는 채널에서 데이터를 받는다.

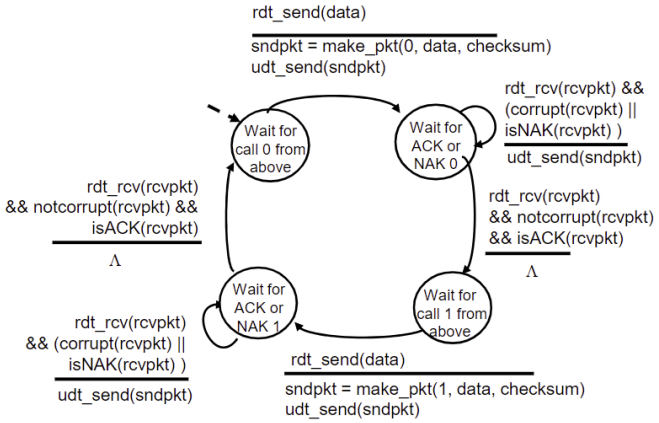
- RDT 2.0
  - 가정 = bit error는 필연적으로 발생할 수 밖에 없다!
    - 완전 신뢰 가능한 채널은 없다.
    - 전송 채널에서 비트 플립이 일어날 수 도 있다.
  - 이 에러를 회복해야한다!!
    - error detection = checksum을 통해서
    - + error recovery
  - 사람 간 대화에선 어떻게 오류를 회복하는가?
    - 다시 말하기 = 재전송.
  - 응답을 보내시오 = 데이터 구조
    - ACKs

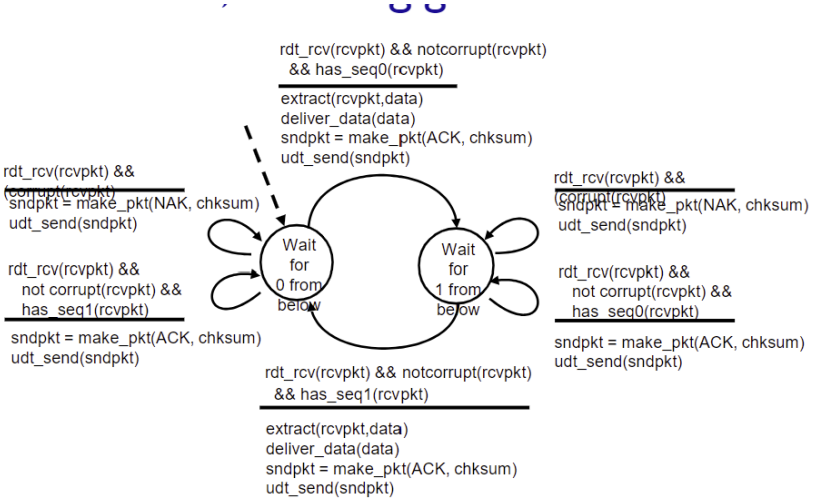
        - 나 이거 받았음....
        - 대다수의 네트워크는 ACKs를 사용.
    - NAKs
        - 나 이거받았음... 이거 빼고 다시 보내주샘...
        - 근데 이건 이론상만 존재하고, 실제 사용은 없음.(구현이 어려움....)
        - NAK이 ACK 보다 합리적으로 보인다.
            - 못받은 5개에 대한 NAK을 보내는거 보다
            - 받은 995개에 ACK를 보내는건 비효율적이다.
    - 근데 왜 NAK을 안쓰고 ACK을 쓰는가?
        - ACK이 먼저 등장했으므로 관성?
        - 수신 측에 받지 못한 패킷을 구별하는 로직이 추가로 필요함.
        - 특정 패킷만 재전송 해야됨. = 더 복잡한 상태 관리.
        - 패킷 손실이 빈번한 네트워크에서 NAK에 의한 재전송이 많아져 오버헤드가 증가할 수 있음.
    - NAK을 받아야 재전송이 시작된다!
        - UDP는 이 과정이 없다.
- Stop and wait
    - 한 패킷을 보냈으면, 응답(ACK, NAK)이 오기 전에는 다시 보내지 않슴니다.
- FSM이 굉장히 복잡해짐
    - 보냈으면 ACK이든 NAK이든 올때까지 계속 기다린다.
    - 답장이 오면 다시 보낸다.
    - 받는 쪽은 상태가 하나면 충분하다.
        - 패킷이 올때까지 계속 대기한다.
        - 오면 답장하고 다시 대기한다....
    - 근데 보내는 쪽에선 상대방이 어떤 상태인지를 모름(커튼)



- 2.0은 심각한 결점이 있다.
  - ACK/NAK가 오염되면 어뜨캄?
    - 송신측은 수신측에 무슨일이 일어났는지 모름.
    - 단순히 재전송을 할 순 없음 = 중복이 생길수도 있으니까!
  - 중복 handling
    - 만약 ACK/NAK이 오염되면 재전송한다.
    - 송신자는 sequence number를 각 패킷에 추가한다.
      - stop and wait이 있으므로, binary(0, 1)이면 충분함.
      - 막 32비트 64비트 필요 없음.
    - 수신측은 중복 패킷을 버린다(받지 않는다)
- RDT 2.1
  - 중복 패킷을 해결하기 위해 sequence #를 0, 1로 만든 것.
    - state가 두 배로 늘어남!
  - 송신측은 응답받은 ACK/NAK의 오염을 필수적으로 체크해야됨.
  - 수신측은 받은 패킷이 중복인지 아닌지도 확인해야함!
    - FSM이 seq #의 값이 0이여야하는지 1이어야하는지 결정함.
  - 수신 측은 아까 보냈던 ACK가 제대로 갔는지 확인할 수 없음.
  - 근데 만약에 ACK가 안오면 어캄

- 무한대로 대기해야됨...

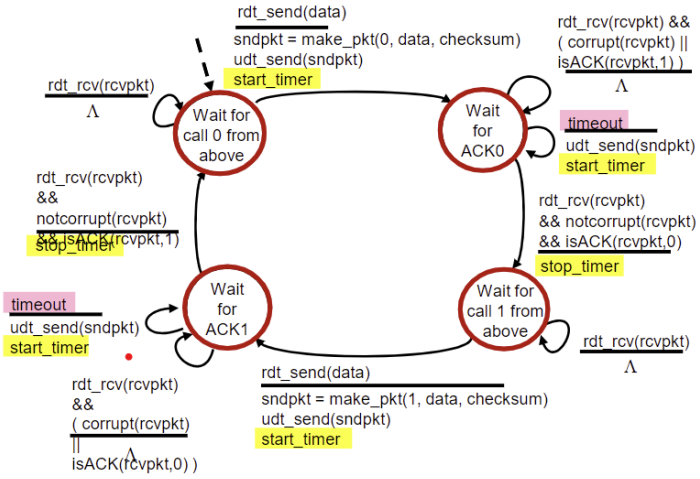




- RDT 2.2
  - NAK을 쓰지 말고, ACK만 사용하자.
- RDT 3.0
  - 새로운 가정 = channels with errors and loss
    - 즉 패킷이나 응답(ACK)이 가다가 자빠져서 없어질 수 있음!
  - 송신측은 'reasonable'한 시간동안(= timeout)ACK을 기다림.
    - 이 시간이 지나면 그냥 다시 전송해버림.
  - 만약 딜레이로 패킷이나 응답이 지연된거라면
    - 재전송에 의해 중복된 패킷이 생김! = seq #로 해결해야됨.

    - 수신측은 무조건 ACK될 패킷의 seq #를 확인해야함.
- FSM에 timer가 추가된다!

rdt3.0 sender



- 총 4가지의 상황이 있을 수 있다.
    1. 손실 없이 정상적으로 통신
        - pkt0 → ack0 → pkt1 → ack1 ....
    2. packet loss
        - pkt0 → ack0 → (pkt1) → timeout → pkt1 → ack1
    3. ACK loss
        - pkt0 → (ack0) → timeout → pkt0 → ack0
    4. ACK delay로 timeout이 먼저 실행되는 경우.
        - pkt0 → ack0 → pkt1 → (ack1)→ timeout → pkt1 → ack1 → pkt0 → ack1(무시) → ack0 → pkt1
- 기능적으론 잘 동작하나, 성능은 썩 좋지 못하다.
    - 프로토콜이 물리적 자원의 사용을 제한하기 때문(stop-and-wait)
        - 즉 응답을 기다리는 시간동안 아무것도 안하기 때문에!
    - trasmission delay = L/R(패킷 길이 / 전송률)
    - + RTT의 딜레이가 ACK을 받아오는데 필요하다.

        - 그럼 rdt 3.0의 성능은 (L / R) / (L/R + RTT)가 된다.
- 파이프라인 프로토콜
    - stop-and-wait으로 인해 대량의 데이터에서 대기 시간이 증가하는 경우.
        - 송신자는 ACK를 받지 않고 다수의 패킷을 전송.
        - seq #가 증가하며, 송수신자 사이의 패킷을 버퍼링함.
    - Go-Back-N
        - 수신자가 누적된 ACK를 전송
            - seq #의 갭이 있으면 ACK하지 않음.
        - 송신자는 ACK 받지 못한 가장 오래된 패킷부터 재전송
            - 모든 패킷이 단 하나의 타이머를 공유함.
    - Selective Repeat
        - 갭이 있더라도, 패킷들을 수신측의 버퍼에 저장
        - = 수신자는 모든 패킷에 대해 ACK 응답을 함.
            - 이 비순차 패킷들은 정렬하여 상위 계층에 전달한다.
        - 송신자는 각 개별 패킷의 타이머를 가지고
        - ACK 못받은 패킷만 개별적으로 재전송
    - TCP는 이 둘을 섞어 사용한다.

▼ 숙제?

- FSM의 주요 구성 요소
    1. 상태
        - 시스템이 존재할 수 있는 다양한 조건이나 상황입니다. 각 상태는 특정 상황이나 맥락을 나타냅니다.
    2. 초기 상태
    3. 최종 상태
    4. 전이
        - event와 action을 통한 상태의 변화.
    5. 입력 알파벳
    6. 출력 알파벳

- 신뢰성있는 전송을 위해 처리해야할 오류들.
    1. 비트 에러
        - 패킷 내부의 개별 비트가 flip되는 오류
    2. 패킷 손실
        - 패킷이 도달하지 못하는 경우.
    3. 패킷 중복
        - 동일한 패킷이 수신자에게 여러 번 도착하는 경우
    4. 패킷 순서 변경
        - 패킷이 전송된 순서와 다르게 수신자에게 도착하는 경우.
- ‘송수신자가 서로의 상태를 알 수 없다’
    - 동기화의 부족
        - 송신자가 보낸 패킷이 성공적으로 수신, 처리되었는지 알지 못하며, 수신자는 송신자가 이 사실을 알고있는지 알지 못한다는 것.
    - 비동기적인 통신
        - 네트워크 통신은 비동기적으로 발생하여, 메시지가 서로 다른 시간에 송수신될 수 있음.
    - 송수신자의 독립성
        - 송수신자는 서로 독립적으로 작동하므로, 송신자가 타임아웃 등으로 패킷이 손실됐다고 간주하고 재전송하면, 중복된 패킷을 받을 수도 있음.
    - 따라서 RDT는 ACK, NAK등의 자료 구조와 타임 아웃, 재전송과 같은 메커니즘을 사용해서 양 측이 일관된 상태에 도달하게끔 해야됨.