3️⃣ 전송 계층 1

Transport services and protocols

- 네트워크 어플리케이션 프로세스 간에 logical communication 제공
- transport protocol → end system(OS)에 구현되어 있음
- send side(보내는 쪽)에서는 어플리케이션에서 메세지가 내려오면 segment로 나눠서 network layer로 내려보냄
- receive side(받는 쪽)에서는 segment들을 다시 합쳐서 application layer로 올려보냄

transport layer vs. network layer

network layer → host간에 logical communication

transport layer → process 간에 logical communication

Internet transport-layer protocol

- TCP (reliable, in-order delivery)

  → congestion control, flow control, connection setup
- UDP (unreliable, unordered delivery)

  → no frills extension of “best-effort” IP (최선은 다하지만 보장하지는 않음)
- services not available

  → delay guarantees(packet switching 때문), bandwidth guarantees

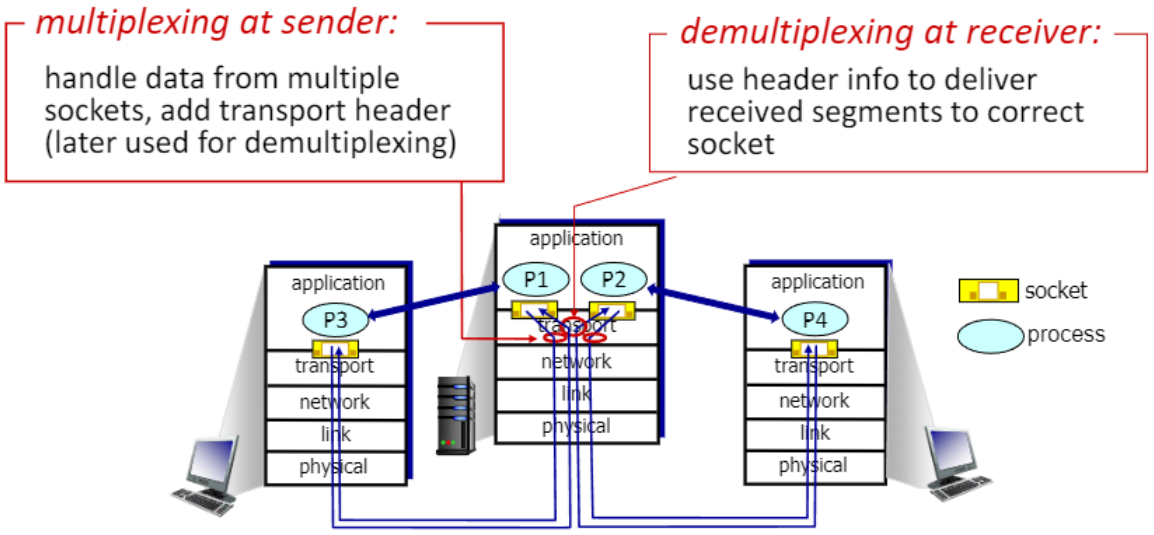
multiplexing / demultiplexing

soket이 두 개 열리는데 지나가는 길이 하나

- sender에서 multiplexing이 일어남

  → 내려갈 때 multiplexing
- sender에서 demultiplexing 일어남

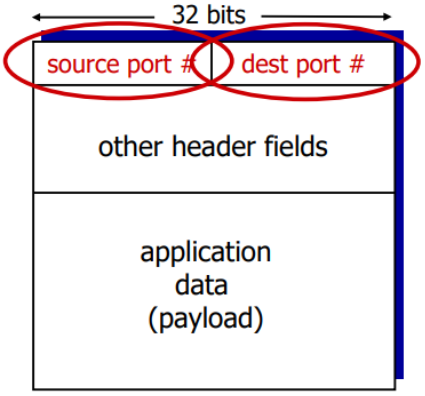
  → 올라갈 때 demultiplexing ~ 올라갈 때 port number를 보고 갈라짐



Demultiplexing

- host가 IP datagram을 받으면 각 datagram에 source IP address와 destination IP address 존재 → packet header에 적혀있음
- 각 datagram은 하나의 transport layer segment를 가지고 있음

  → 각 segment는 source와 destination port number를 가짐
- host가 IP 주소와 port number를 이용해서 segment가 어디로 가야할지 정해줌



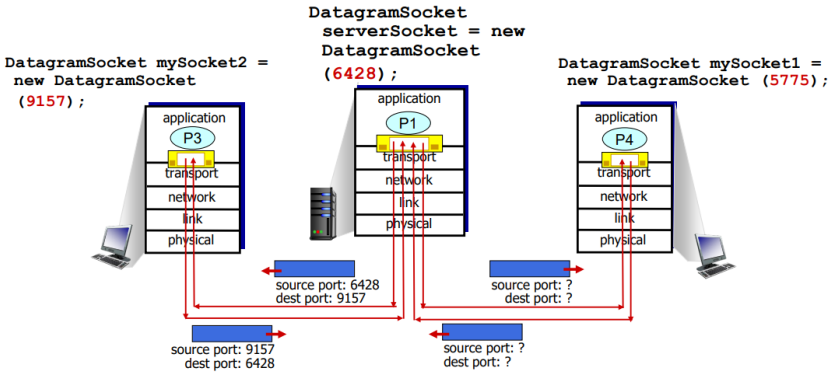
TCP/UDP segment format

Connectionless demultiplexing ⇒ UDP

host가 UDP segment를 받았을 때 segment에 있는 destination port number를 확인

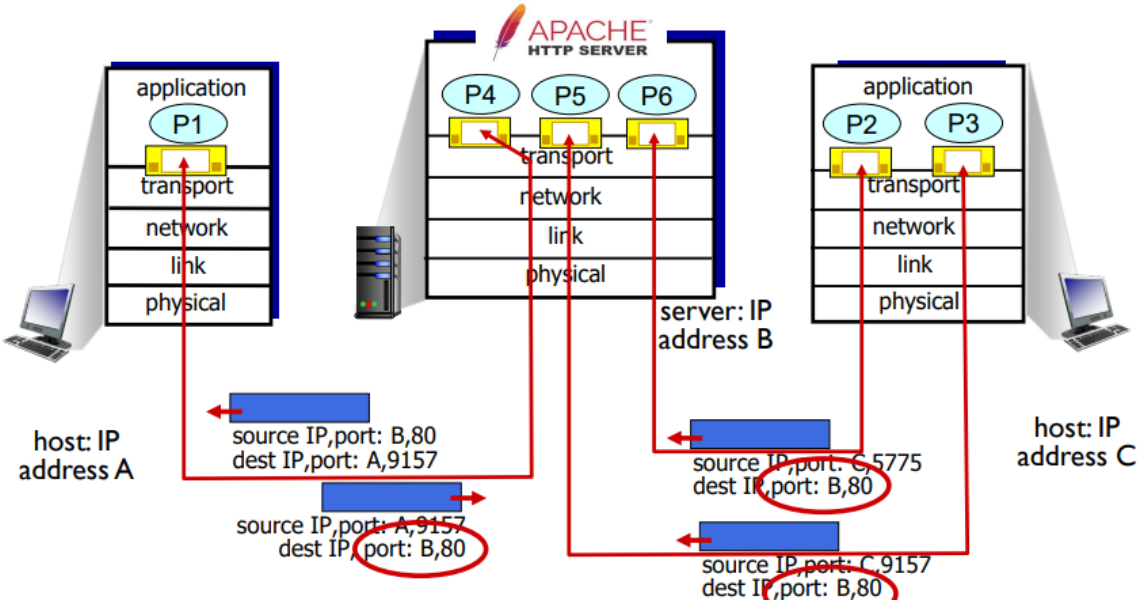
→ UDP segment를 port number에 해당하는 pocket에 보내버림

~ IP datagram이 있는데 같은 port number, 서로 다른 IP address / source port number면 같은 주소로 갈 것



Connection-oriented demultiplexing

- TCP socket은 4개의 정보로 identify됨
  - source IP address
  - source port number
  - destination IP address
  - destination port number
- demultiplexing → receiver가 4개의 모든 value 값을 이용해서 적절한 socket에 segment를 보냄
- server는 동시에 여러 개의 TCP socket을 support할 수 있음
- 웹 서버는 가각의 client에 대해 각기 다른 socket을 가지고 있음



- server는 공유기 안에서만 사용되는 private IP address 사용
- 외부 네트워크로 갈 때에는 공유기에 할당된 public IP address로 바뀌어서 나감

UDP (User Datagram Protocol)

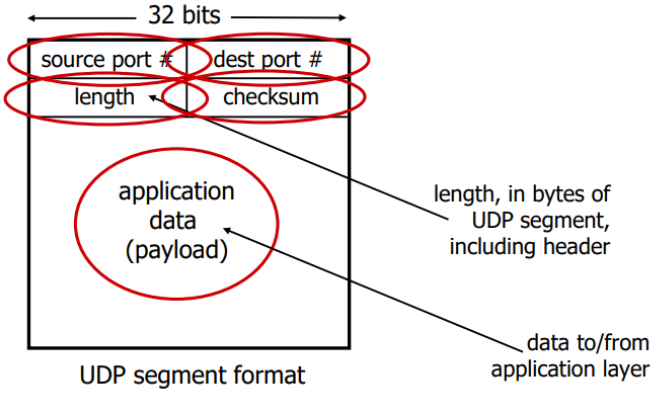
⇒ “best effort”

UDP를 통해서 보내면 순서가 바뀌어서 갈 수도 있고 중간에 잃어버릴 수도 있음

- connectionless → handshaking 없이 그냥 서버에서 보냄, 각 UDP segment가 독립적으로 handle됨

  ⇒ streaming multimedia app에서 많이 사용됨, DNS, SMP
- UDP를 사용할 경우 application layer에서 reliability에 대한 것을 추가해야 함

UDP : sengment header



UDP가 왜 필요한가 → application 특성 때문

- 추가 delay가 발생할 수 있는 connection establishment 없음
- send와 receiver 간단, header size가 작음, congestion control이 없음

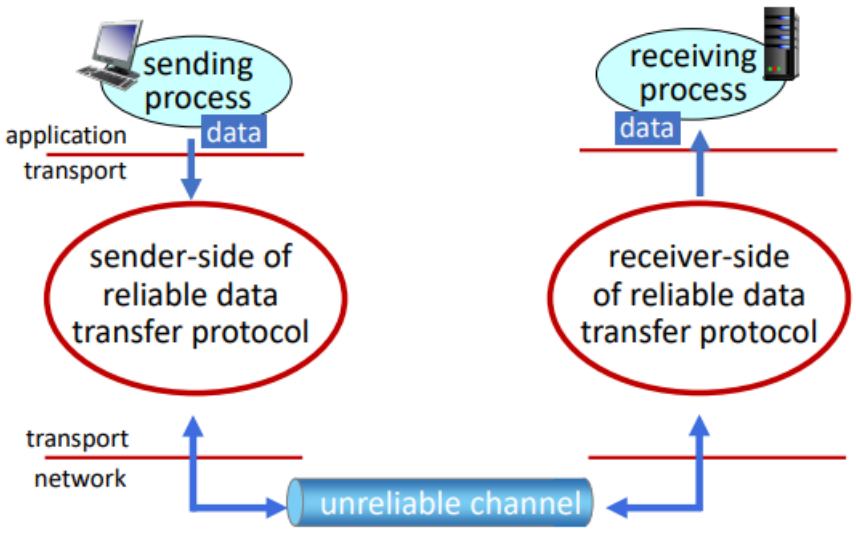
UDP checksum

→ segment에서 error를 detect하기 위해

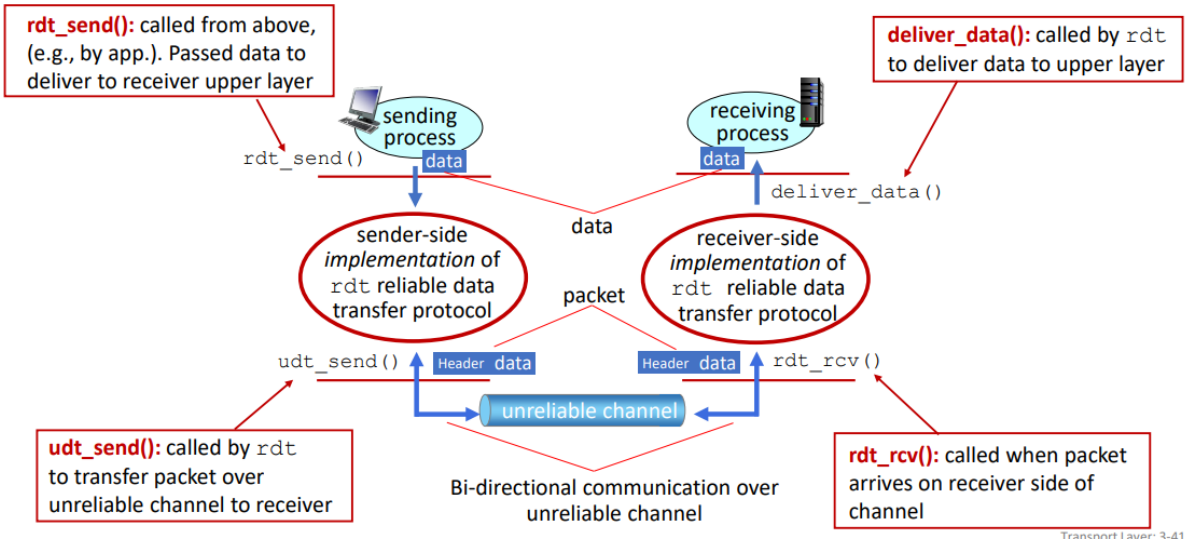
- sender
  - segment를 16 bit 단위로 끊어진 string으로 봄
  - checksum→ segment content를 addition함 → 16 bit 끊어진 것 다 더함
  - sender는 check sum value를 UDP checksum field에 적음
- receiver
  - 받은 segment의 checksum을 계산함
  - checksum이 checksum field에 있는 값과 같은지 비교 → 다르면 error detected

Principles of reliable data transfer

→ application, transport, link layer에서 중요



⇒ unreliable channel을 건널 때 TCP가 reliable data transfer protocol을 이용해서 위에서는 마치 reliable channel 서비스를 받고있는 것처럼 느껴지게 함



- application에서 send라는 함수를 이용해 데이터를 밑으로 보냄
- receiver의 reliable data tranfer protocol에는 packet에 에러가 있거나 중복된 packet이거나 이러한 이벤트 발생시 다시 보내달라고 요청하는 등의 함수들이 들어있음
-

Reliable data transfer

→ sender와 receiver가 고정되어 있다고 가정

finite state machine (FSM)을 사용해 sender와 recevier가 할 일을 정의

rdt 1.0 : reliable transfer over a reliable channel

→ 밑에 channel들이 완벽함 → bit error도 없고 packer loss도 없다고 **가정**

sender와 receiver에 대한 FSM을 디자인

- sender

  위쪽에서 데이터 내려오기를 기다림

  → rdt_send(data) 이벤트가 발생하면 packet을 만들고 보냄

  ⇒ 무조건 잘 전다로디기에 이후에 할 일이 없음

- receiver

  packet이 도착하기를 기다림

  → rdt_receive(packet) 이벤트 발생하면 packet에서 data 호출

rdt 2.0 : channel with bit errors

→ bit error가 발생할 수 있는 channel ⇒ detect 할 수 있음 ~> 다시 보내달라고 요청할 수 있다는 것을 가정

- Acknowlegements(ACK) : 잘 받았다는 표시를 하기 위해 receiver가 sender에게 보냄
- Negative Acknowledgements(NAK) : packet에 에러가 있다는 표시를 하기 위해 receiver가 sender에게 보냄

  → sender는 packet을 다시 보냄

⇒ 1.0에 비해 2.0은 error detection하고 receiver가 sender에게 feedback을 보냄

- sender

  위에서 데이터가 내려오기를 기다림

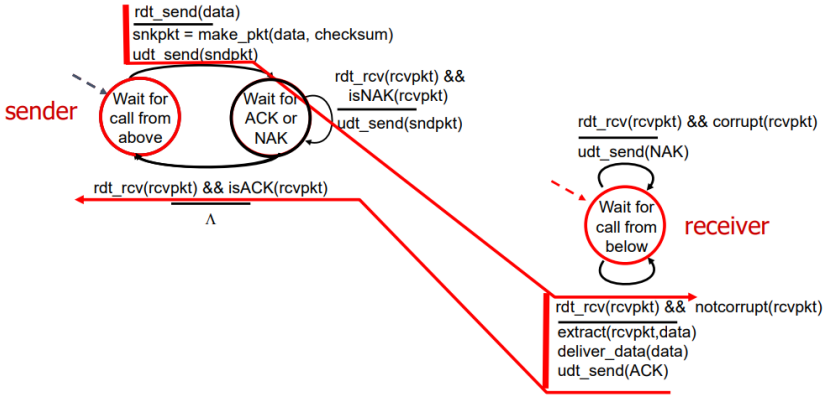
  → 내려오면 data와 checksum field로 packet을 만들고 보냄

  → ACK 또는 NAK을 기다리는 상태가 됨

- receiver

  packet이 오기를 기다림

  → 받고 packet에 에러가 있으면 NAK을 보냄 / 에러가 없으면 data를 추출해서 보냄 + ACK 보냄



rt 2.0 has a fatal flow!

packet에 에러가 있냐 없냐만 가정함 → ACK과 NAK에도 에러가 생기면?

→ sender는 ACK/NAK을 받았는데 에러 때문에 이게 ACK인지 NAK인지 알지 못함 + 그렇다고 무작정 packet 재전송할 수 없음

⇒ sender가 ACK 또는 NAK에 에러가 발생하면 무조건 재전송

- 각 packet에 sequence number를 붙임
- receiver는 중복된 packet이 오면 버림

기본 방식 : stop and wait ⇒ 보내고 기다림 (또 보내지 않음)

rdt 2.1 : sender, handles garbled ACK/NAKs

→ 에러가 생긴 ACK, NAK까지도 handle할 수 있음

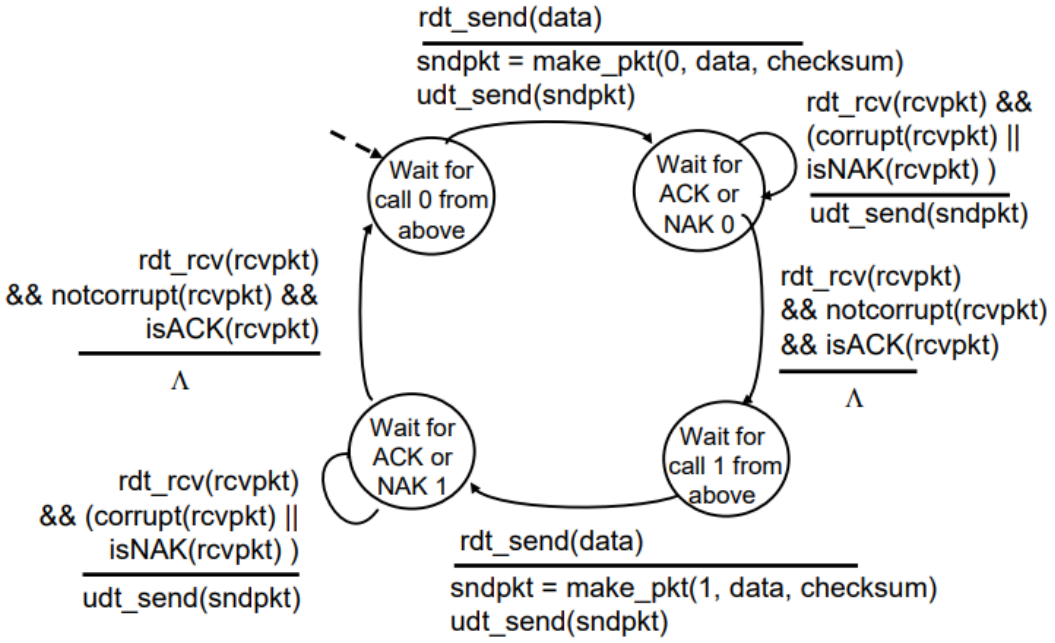
- sender

  위쪽에서 데이터 오면 sequence number, data, checksum field 이용해서 packet을 만들고 보냄

  → 해당 packet에 대한 ACK 또는 NAK을 기다림

  → packet에 에러가 있거나 NAK이 오면 다시 전송

  → 받았는데 packet에 에러가 없고 ACK이면 달라진 sequence number로 packet 보냄



- receiver

  밑에서 데이터 오기를 기다림

  → packet 받고 에러가 없으면 data를 추출해서 위로 보냄 + ACK 보냄

  → packet 1을 기다리는 상황

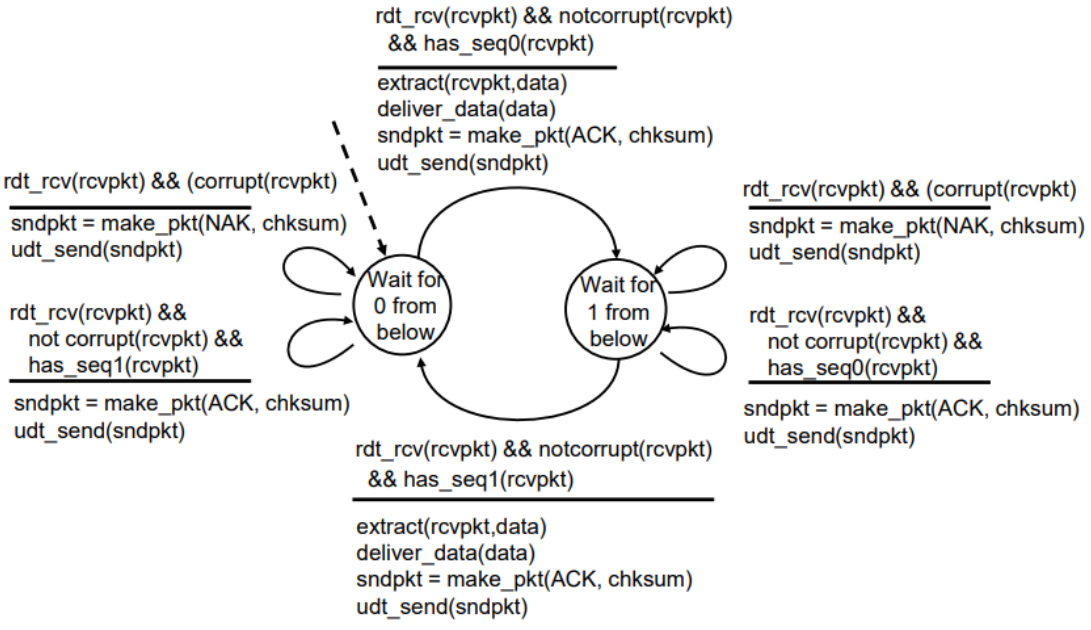
  또는

  → packet을 받았는데 에러가 있으면 NAK을 보냄

  또는

  → 에러가 없는데 기다리는 packet이 아닌 다른 packet이 오면 ACK을 다시 보냄

  ~> 이전에 ACK을 보냈는데 sender에서 에러가 난 채로 도착해서 이전 것을 다시 보낼 때



rdt 2.1 : discussion

- sender
  - sender가 sequence number는 packet에 적음 ⇒ sequence number가 0 또는 1이면 충분함 → stop and wait이기 때문에
  - 받은 ACK이나 NAK이 에러가 발생했는지 반드시 체크
  - state는 내가 기대하고 있는 packet의 sequence number가 0인지 1인지 정확히 기억해야 함 → state 두개 필요
- receiver
  - packet이 중복인지 확인해야 함
  - receiver가 마지막으로 보낸 ACK과 NAK이 sender에서 잘 받아졌는지 알 수 없음

rdt 2.2 : a NAK-free protocol

→ NAK이 없고 ACK만으로 packet 잘 받았는지 확인

receiver가 마지막으로 제대로 받은 packet에 대해서 ACK을 보냄

→ 중복 packet이 오면 NAK과 같은 액션을 취함

- sender

  0번을 기다림

  → sequence number 0과 data, checksum field로 packet을 만들어서 보냄

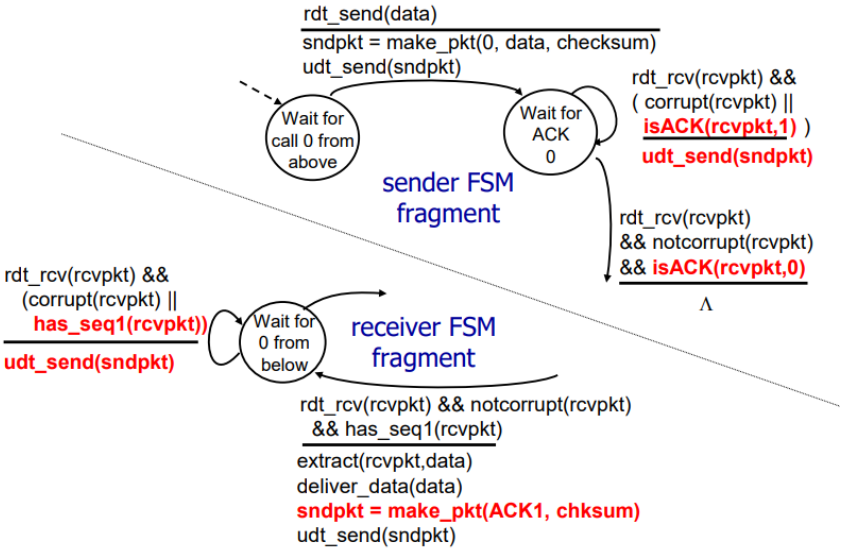
  → 0번에 대한 ACK을 기다리는 상태

  → 1번에 대한 ACK이 오거나 packet에 에러가 있으면 다시 전송

- receiver

  0번 packet 기다림

  → 에러가 발생하거나 1번 packet이 오면 ACK 0 packet 보냄



rdt 3.0 : channels with error and loss

→ packet loss가 일어날 수 있다고 가정 → receiver는 아무런 액션을 취하지 않음

⇒ sender는 무한정 ACK을 기다리게 됨 ~ 타이머를 둬서 어느정도 시간이 지나면 packet loss가 일어났다고 판단

- 만약 packet loss가 아니라 ACK이 지연이 되어서 왔다면 sequence number로 handling

  → 다시 보낸 다음 해당 ACK이 도착 → 중복 → receiver가 중복된 것 버림
- receiver는 어떤 패킷에 대해 받은 sequence number인지를 알려줘야 함 + ACK을 보내고 타이머를 돌림

- sender

  data 오기를 기다림

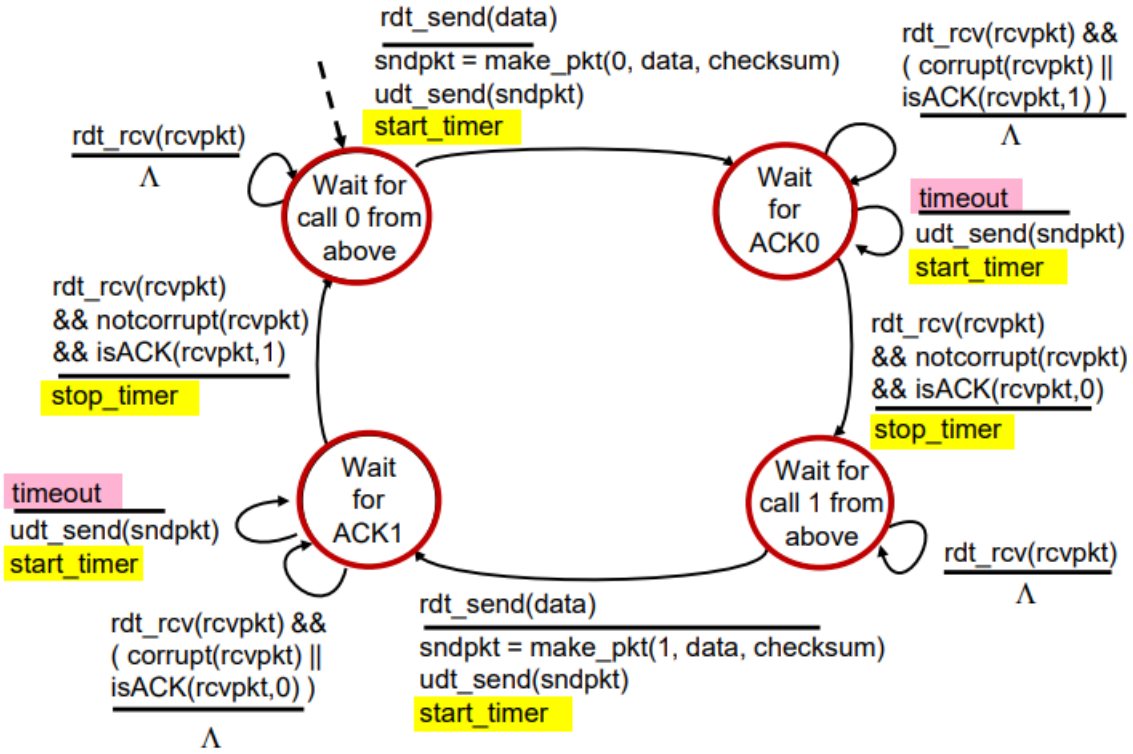
  → sequence number, data, checksum field로 packet을 만들고 timer 시작

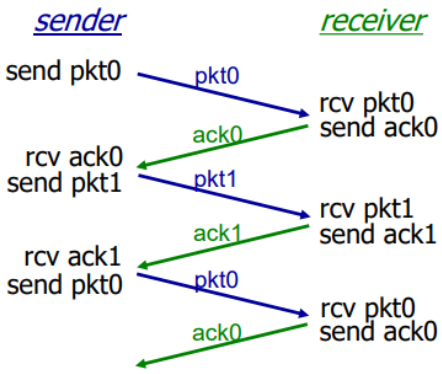
  → packet을 받았는데 에러가 났거나 다른 packet이면 아무일도 안함

  → timeout이면 packet다시 전송, 타이머 다시 시작

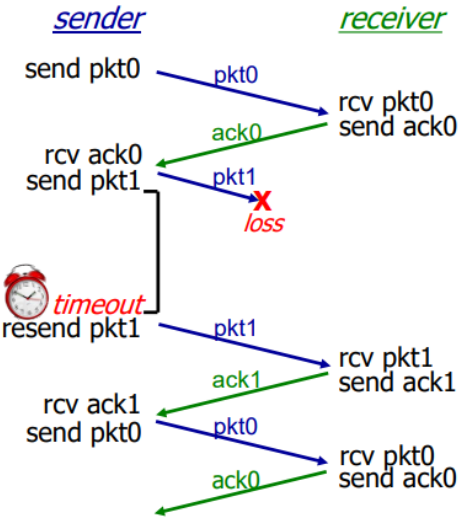
  → 제대로 된 ACK이 오면 타이머 중지하고 다음 데이터를 기다림

  → packet이 또 오면 아무일도 하지 않음 (중복 packet)

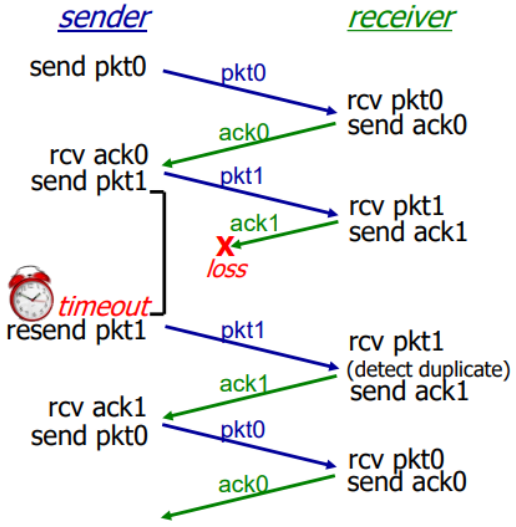




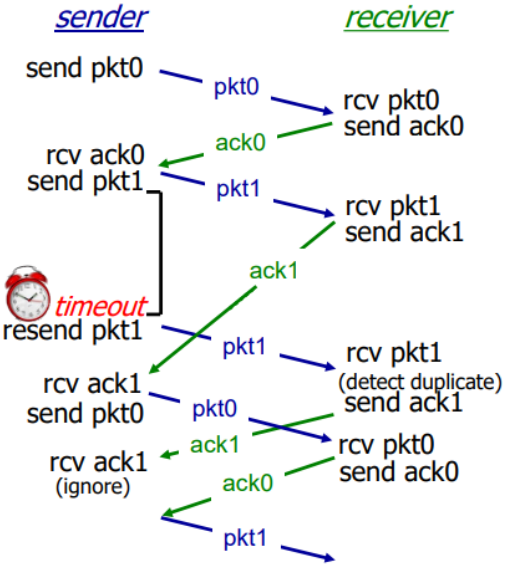
(a) no loss



(b) packet loss



(c) ACK loss



(d) premature timeout/ delayed ACK

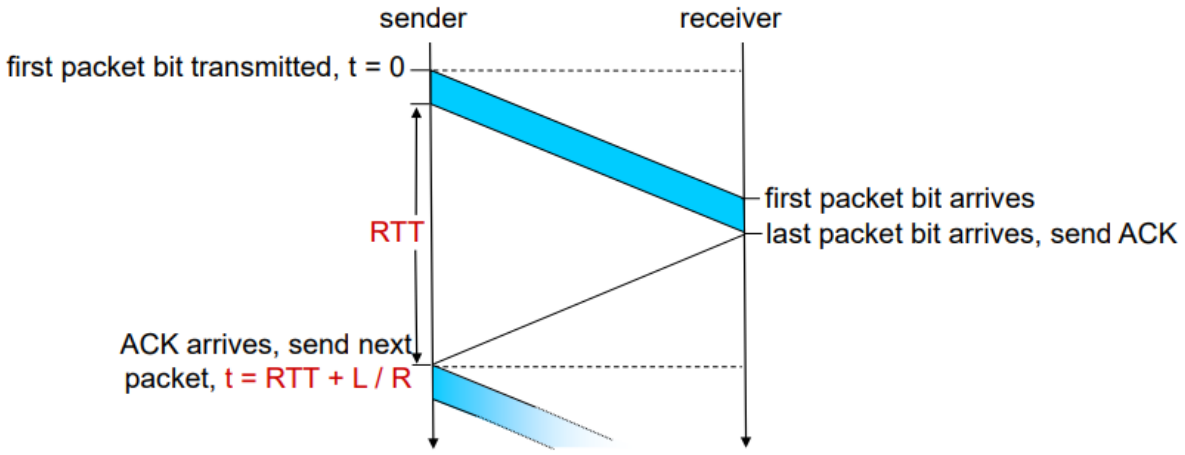
Performance of rdt 3.0

→ performance가 매우 안좋음

공식 : $D_{trans} = \frac{L}{R}$

D_{trans} : transmission delay, L : size, R : 속도

U_{sender} = utilization → server가 사용하는 비율



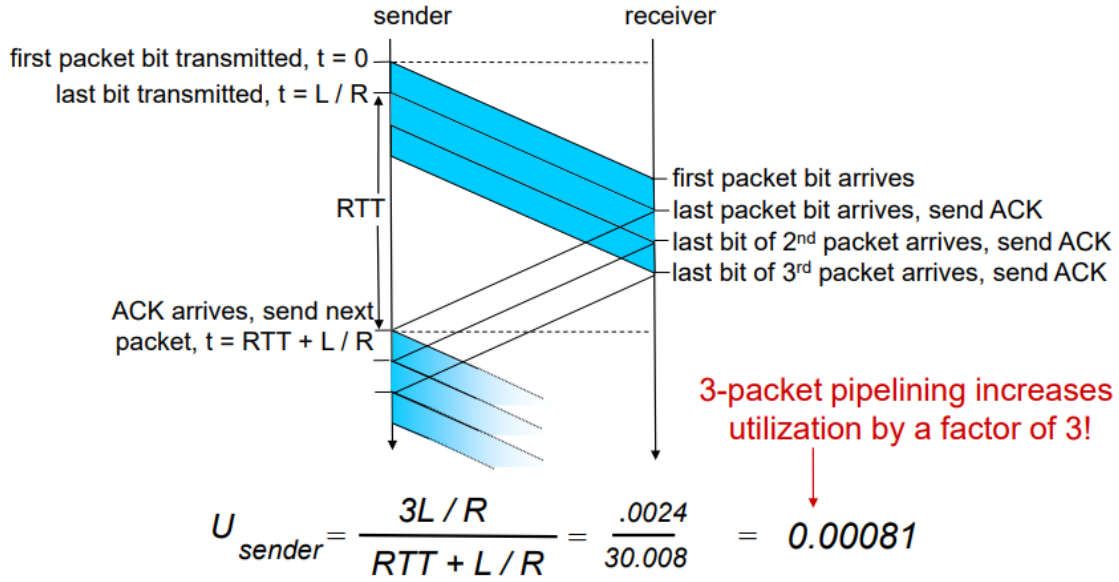
RTT → 링크가 노는 시간 ~ 이걸로 얼마나 효율적으로 사용되는가 봄

$U_{sender} = \frac{L/R}{RTT+L/R}$

Pipeline protocols

stop-and-wait에서 packet을 하나 보내고 ACK 기다리고 packet 보내고의 반복 → 비효율적

⇒ pipelining : 앞에 있는 일이 다 끝나기 전에 다른 일도 집어넣는 것



두 개의 pipelined protocol

- Go-Back-N : N개의 protocol을 ACK 없이 일단 보내놓음 → 각각에 대한 ACK이 오면 그 다음 packet을 보냄
  - sender가 N개의 unacked packet을 pipeline에 가지고 있을 수 있음 = 한 번에 최대 N개 밀어넣기 가능
  - receiver는 cummulative ACK만을 보냄 → 지금까지 잘 받아진 packet 순서대로 보냄

    - sender는 ACK을 못 받은 packet 중 가장 오래된 packet에 대한 타이머를 가지고 있음

      → 타임아웃되면 ACK 못 받은 packet 전부 다 보냄

- Selective Repeat

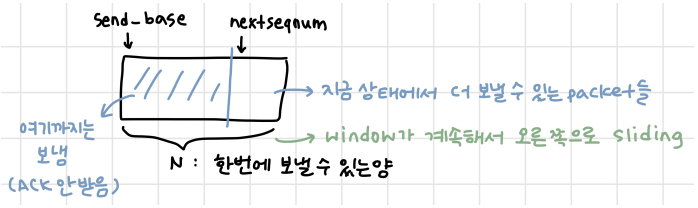
  → 각각의 packet에 대한 ACK을 보냄

    - N개의 packet을 한꺼번에 보낼 수 있음
    - sender는 ACK을 못 받은 모든 packet 각각의 타이머를 가지고 있음

      → 타임아웃되면 해당되는 packet만 재전송

Go-Back-N : sender

- packet header에 k-bit sequence number를 적음
- size N을 가진 window 존재

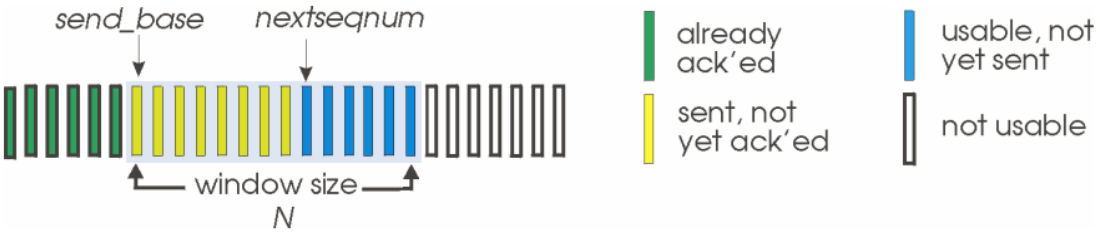


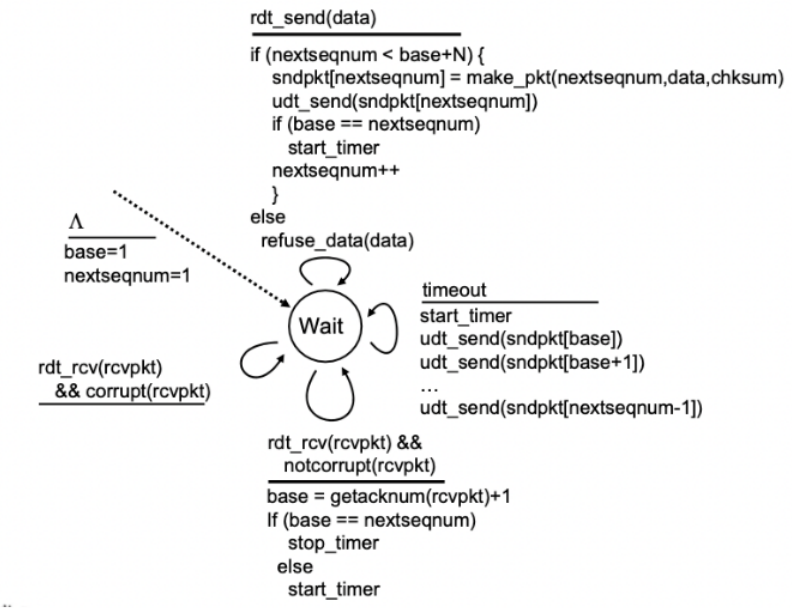
- ACK(n) : sequence number n까지 ACK을 제대로 받아졌다
- 가장 오래된 packet에 대한 타이머를 가지고 있음

  → 타임아웃되면 window 안에서 해당 sequence number보다 위쪽에 있는 packet 모두 재전송

- sender

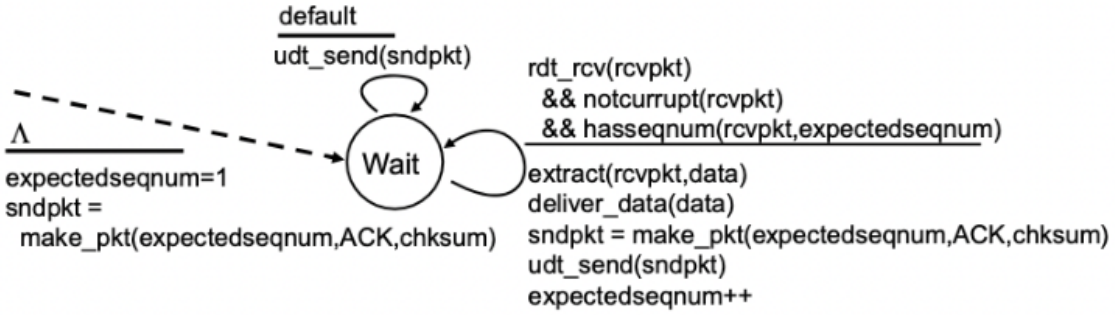
  base = 1, nextseqnum = 1로 초기화





- receiver

  expectedseqnum = 1로 초기화



  순서대로 받은 것 중 가장 최근 것에 대한 ACK을 보냄

  → 1번 packet 받고 2번 기다리는데 3번 packet 오면 1번에 대한 ACK을 보냄 → 중복 ACK 발생 가능

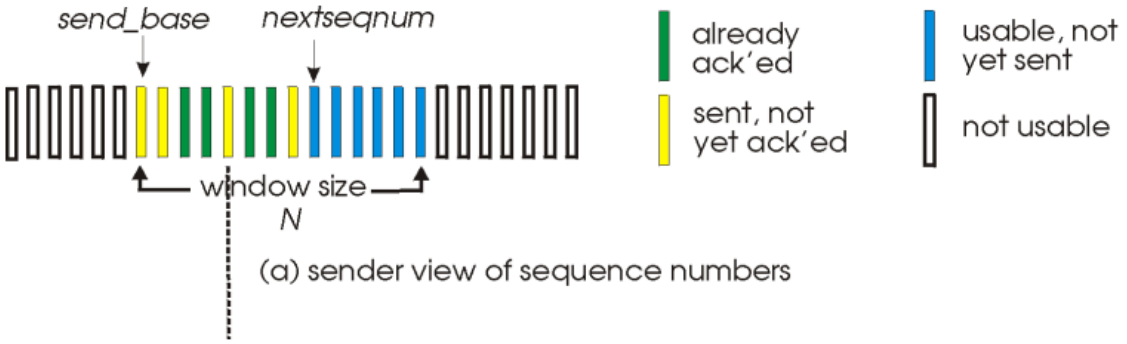
  receiver는 expectednum을 계속 기억해야 함

  순서가 바뀌어서 packet이 도착할 수 없음 → buffering을 하지 않음

Selected Repeat

→ packet 하나하나에 대해서 따로따로 ACK를 보냄 → receiver에서 packet buffering이 있음

- 순서대로 잘 받아진 것들만 위로 올림
- sender는 ACK을 못 받은 packet에 대해서만 재전송함
- 각각의 unacked packer에 대한 타이머를 가지고 있음
- sender window → N개의 연속된 sequence number를 가짐
- 이미 ACK를 받은 packet과 못받은 packet이 window안에 섞여 있음
- receiver sindow 보면 순서가 아닌데 이미 받아진 (out of order / buffered) packet이 있음



- sender

  위쪽에서 데이터 내려옴

  → nextseqnum이 window 안에 있으면 packet 보냄

  → n에 대한 timeout 발생하면 다시 packet 보내고 timer 시작

  → ACK(n) in [sendbase:sendbase + N] → packet receive라고 표시

  ⇒ n이 못받은 packet 중 가장 작은 숫자였으면 window base 다음 unacked로 이동
- receiver

  packet in [rcvbase:rcvbase+N-1] 받아지면 ACK(n) 보냄

  → out of order면 buffering 함, in order면 위쪽 layer로 올려보냄, window 이동

  packet in [rcvbase - N:rcvbase - 1] → ACK(n) 보냄

  ~ -N인 이유는 sender window를 보면 N개 전까지는 이미 지나가서

  → ACK에 대한 loss 발생으로 일어날 수 있는 이벤트