전송 계층

packet 에러를 해결하기 위한 4가지 메커니즘

Error detection: 에러 확인

feedback: 데이터에 대한 피드백

retransmisstion: 재전송

sequence: 데이터 넘버

packet loss 해결을 위한 메커니즘

Timeout

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L\ (\text{packet length in bits})}{R\ (\text{transmission rate, bps})} = \frac{8\text{kb/pkt}}{10^{**}9\ \text{b/sec}} = 8\ \text{microsec}$$

  - U_{sender}: utilization – fraction of time sender busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

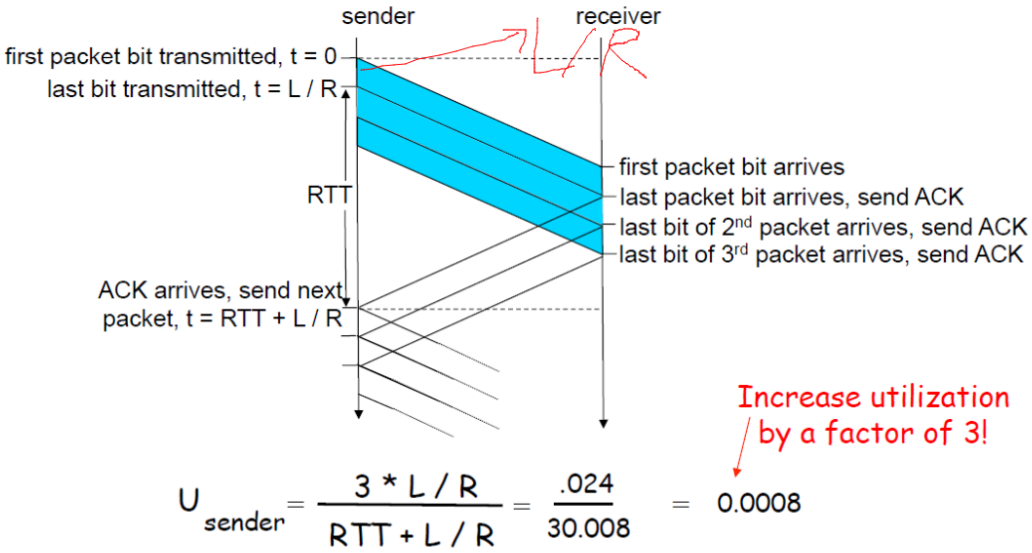
utilization: 전체 시간 중에서 sender가 네트워크를 사용하는 비율

이게 높으면 높을수록 좋다. 16차선을 깔아놓고 여러 차가 이용하는 게 좋기 때문에 (유후 되지 않고 사용되는 것이 의미있기 때문에)

L : 보내는 데이터의 길이

R: 빛의 속도?

RTT = round trip time



데이터 하나를 전송까지 끝마친 후 feedback을 받기까지 시간

L/R 은 데이터를 보내는 시간

데이터를 전송하는 시간 + feedback받기까지 시간 사이에 데이터를 세개나 보냈기 때문에 utilization이 증가했다.

한꺼번에 데이터를 많이 보낼수록 utilization이 증가한다.

---

pipelined의 프로토콜을 신뢰성 있게 동작하게 만드는 2가지 approach:

go-back-n, selective repeat

Go-Back-N

window size만큼은 feedback 받지 않고 한꺼번에 보낸다.

Go-Back-N

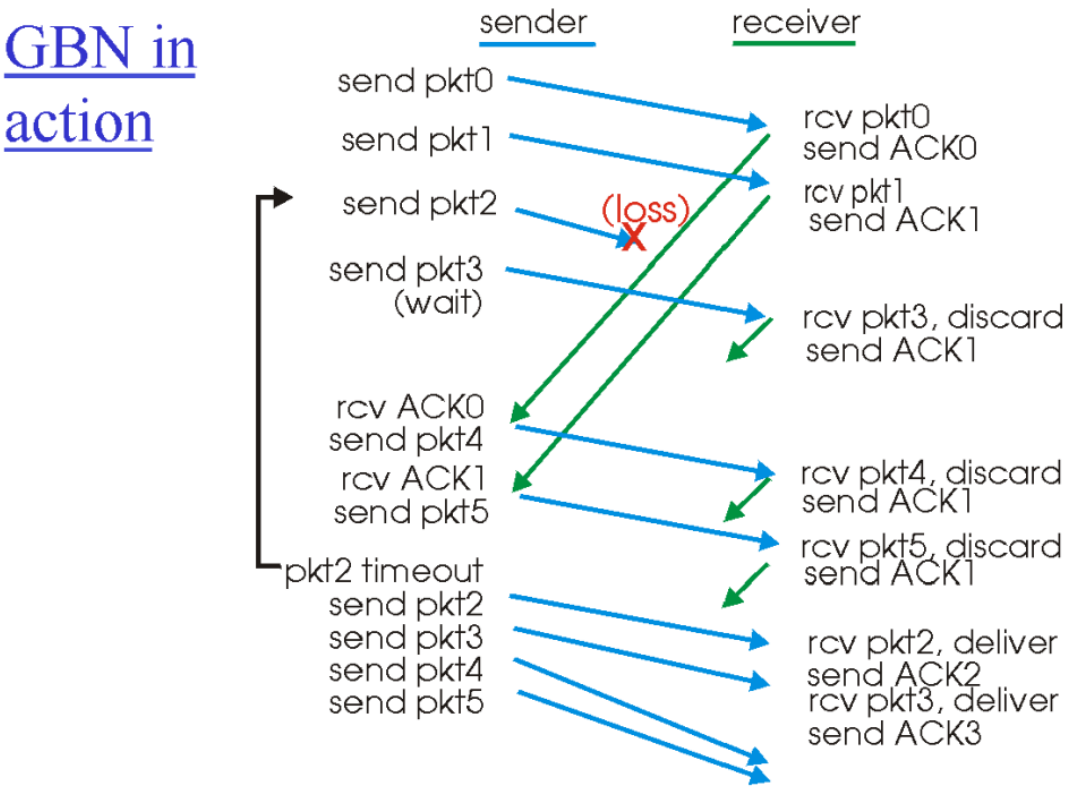
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n **and all higher seq # pkts in window**

만약 timeout이 생기면 타임아웃이 생긴 n과 그 상위 시퀀스를 다 재전송한다.

전송하는 데이터 각각에 대응되는 timeout을 켠다.



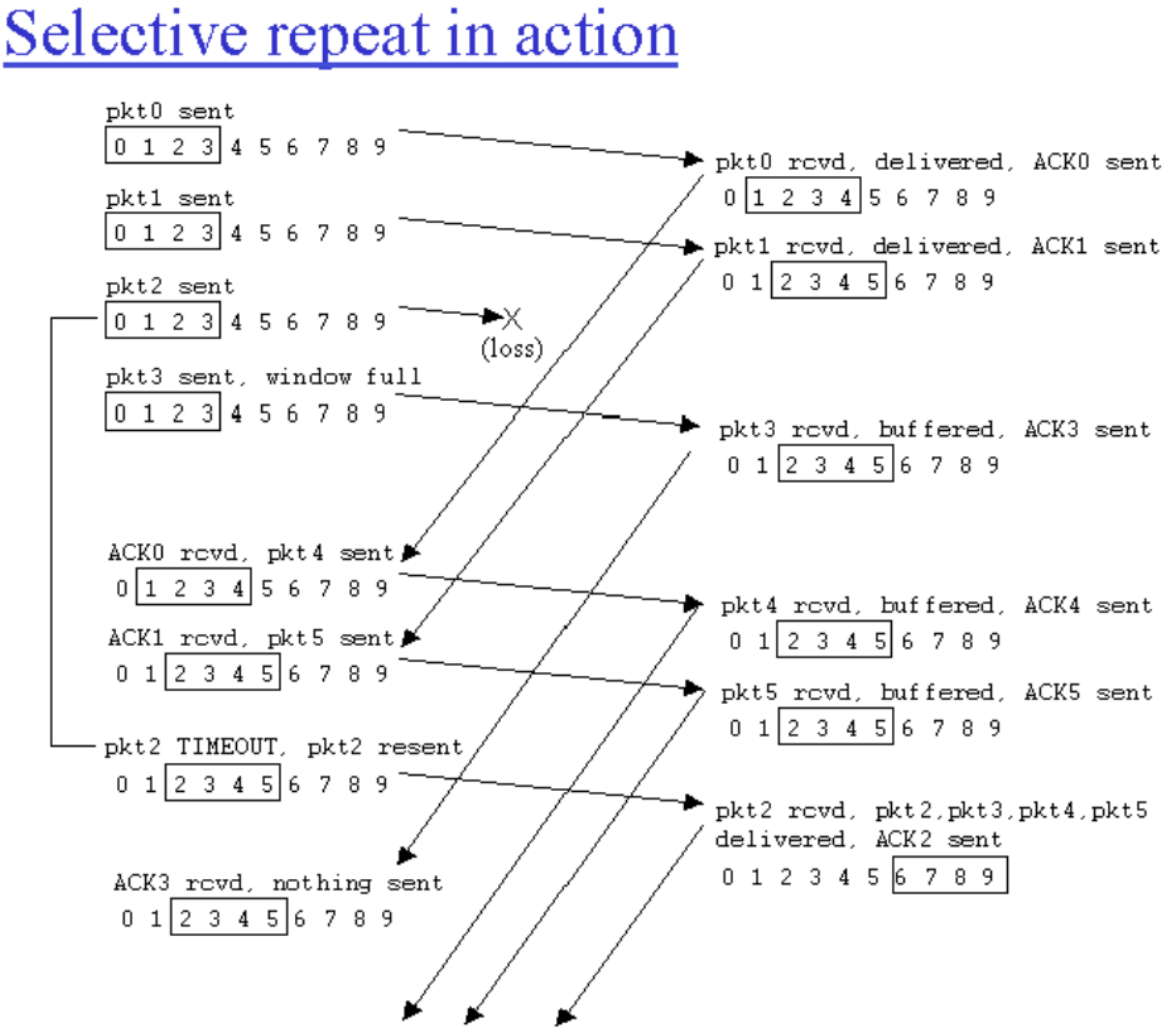
receiver는 시퀀스 순서대로 받기 때문에 하나가 유실되면 그 데이터가 다시 들어올 때까지 전송된 데이터를 받지 않는다. ack를 받을 때마다 윈도우를 한칸씩 이동시킨다.

feedback을 받은 시퀀스는 버퍼에 저장할 필요가 없고, 아직 윈도우에 들어 있는 애들은 재전송에 대비해서 버퍼에 저장해둬야 한다.

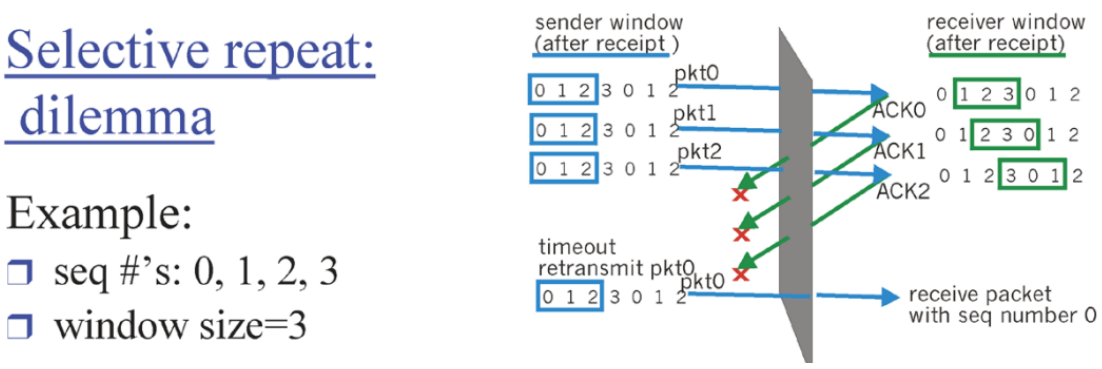
Selective Repeat

receiver는 순서에 맞지 않은 packet가 들어와도 저장을 한다.

sender도 ack 을 받지 않은 packet만 다시 전송한다. (유실된 packet만 재전송)



최소의 시퀀스 범위를 정해야 하는데, 이는 윈도우 size와 관련이 있다.



ex seq = window size)

receiver는 0, 1, 2 를 다 잘 받아서 ack을 보냈지만, ack이 유실되어서 sender는 다시 0을 보낸다. 하지만 receiver 측에서는 재전송된 0이 아니라 새로운 0으로 간주하는 문제점이 생

긴다.

duplicate 데이타를 구별할 수 있는 최소의 seq 을 가져가야 한다.

전송계층 2

TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

Transport Layer 3-52

point to point : 소켓이 서로 1개씩만 연결된 상황

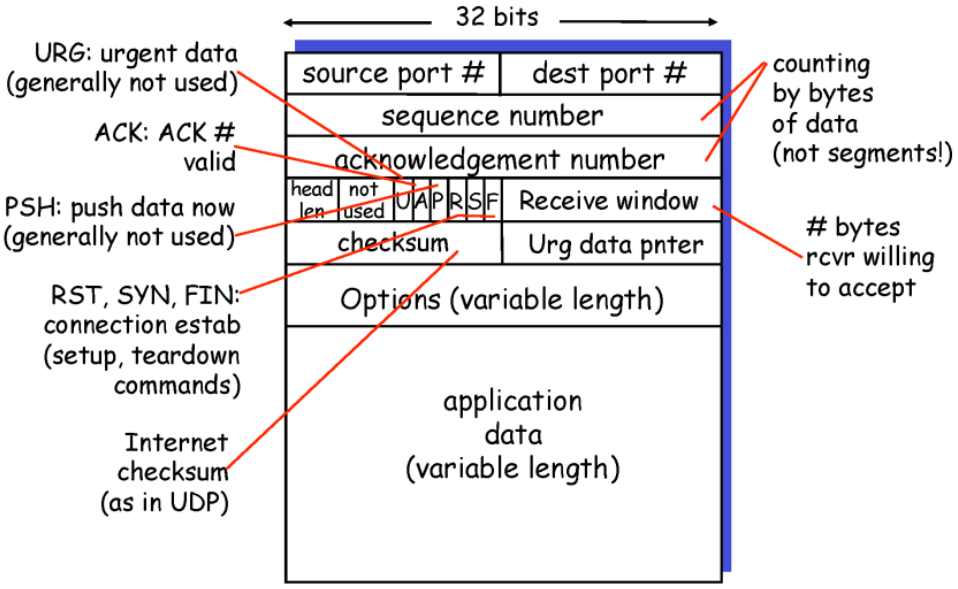
pipelined : packet을 한꺼번에 윈도우 방식으로 보낸다.

full duplex data : 데이터가 양방향으로 보내진다. sender와 receiver 둘 다 메세지를 보내고 받는다.

connection oriented: 서로 데이터를 주고 받기 위한 커넥션이 진행된다.

각자 sender buffer 와 receiver buffer 를 가지고 있음

TCP segment structure



Transport Layer 3-53

port 는 각각 16비트씩. 따라서 2의 16-1승의 숫자를 가질 수 있다.

checksum은 이 segment에 error가 있었는지 확인하기 위한 field

receive window는 receiver가 얼만큼 더 받을 수 있는지 buffer의 크기

TCP seq. #'s and ACKs

Seq. #'s:

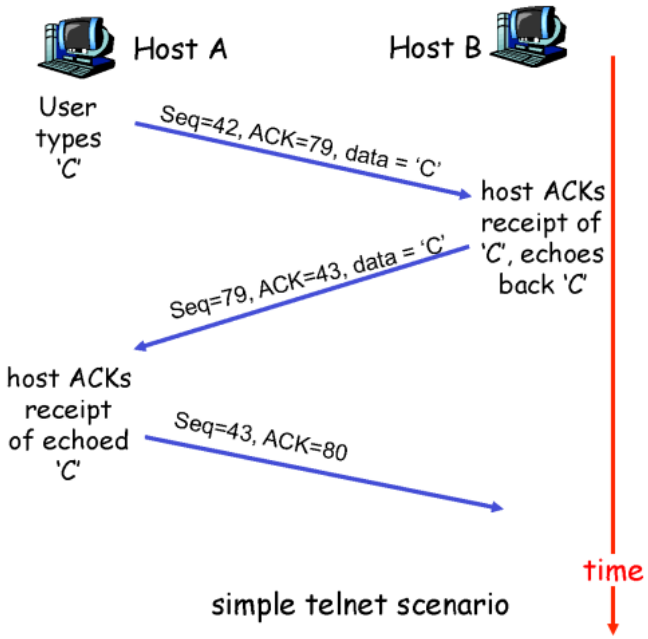
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



seq: 시퀀스 넘버는 segment 안의 데이터를 쪼개서 보낼 때 제일 첫번째 byte의 숫자

20byte의 크기일 때 10 크기로 쪼개면 첫번째 seq는 0, 두번째 seq는 10

ack: 다음 번에 받기로 예상되는 seq 넘버 (TCP에서의 ack는 cummulative ack이다. 따라서 10을 보내면 9번까지 다 잘 받았다는 내용을 담고 있다. 메시지가 100개 들어오면 ack를 일일이 보낼 필요없이 마지막에 101을 보내면 된다.)

각각 sendbuffer 와 receivebuffer를 갖고 있음

sendbuffer는 자기가 만들면서 번호를 트래킹하고, 이에 대응되는 상대방의 receivebuffer는 sender가 보낸 것을 저장하면서 트래킹 한다.

실제로는, 전송되는 정보에 header도 같이 보낸다.

위의 그림의 전개를 살펴보면,

HOST A는 시퀀스 42를 보내고, ack79를 받기를 기대한다.

HOST B는 42를 잘 받았기 때문에 ack 43 을 받기를 기대한다고 보내고, A가 요청한 seq 79를 보낸다.

TCP Round Trip Time and Timeout

$$EstimatedRTT = (1-\alpha)*EstimatedRTT + \alpha*SampleRTT$$

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

RTT 값은 큐잉 딜레이 또는 sender와 receiver의 위치에 따라 천차만별이다.

sampleRTT는 실제로 측정한 값

공식은 중요하지 않고, time out을 잡을 때 마진을 더 해준다는 것

TCP reliable data transfer

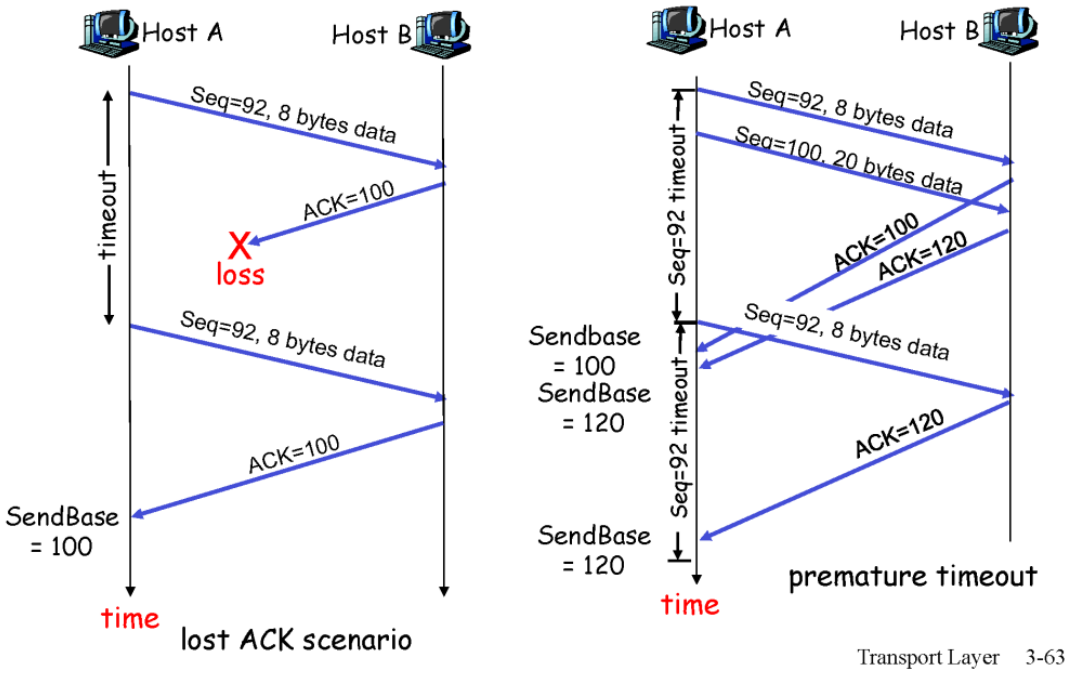
- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

한번에 때려 붓는 파이프라인 세그먼트

cumulative acks

타이머는 하나(go back end과 비슷하지만 다른 점은 timer가 터지면 그동안 보냈던 것도 다시 다 재전송 해야 하는데, TCP에서는 타이머가 터지면 그거에 해당하는 segment만 재전송한다.)

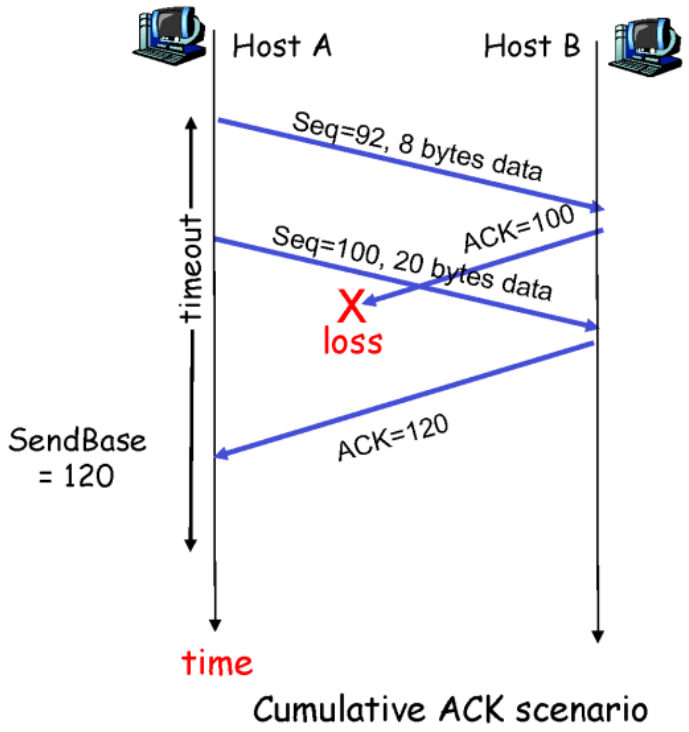
TCP: retransmission scenarios



첫번째, ack가 유실된 경우, 윈도우만큼 다시 보낸다.

두번째, 타이머가 터졌을 때 그에 대한 segment만 보내고, ack는 120번이 날라오니까 119번까지 잘 받았다는 얘기니까 sender는 120번 이후부터 다시 보낸다.

TCP retransmission scenarios (more)



ack = 100이 유실됐지만 다음에 120을 보내면서 119번까지 잘 받았다는 것을 알 수 있으니 sender는 120 이후부터 보내면 된다. (만약 receiver가 seq 100을 받지 못 했으면 계속 ack 100만 보냈을테니까)

fast retransmit = ack 가 3번 이상 반복돼서 오면, 타이머가 터지기 전에 유실된 것으로 판단하고 재전송을 보내도록 권고하고 있다. (타이머가 울릴 때까지 기다리기 전에)

그 전에 ack 요청과 3번 까지 합쳐서 총 4번

ack 10 (정상적 요청 = 10을 주세요), ack 10, ack 10, ack10 (비정상적 3번)