4
전송 계층 2

Selected Repeat dilemma

만약 sequence number 가 0, 1, 2, 3인데 window size가 3이라면 3까지 쓰면 0으로 다시 돌아옴

→ 같이 한 번에 전송하는데 ACK 못 받으면 이미 잘 받았는데 재전송됨

⇒ receiver는 새로운 packet인지 알고 올려보낼 수 있음

~> window size를 줄이거나 sequence number의 space를 늘림

window size만큼 전진했을 때 sequence number가 거기에 걸리면 안됨

→ N : window size, S : sequence number space

$2N-1 \leq S-1$

TCP

- point-to-point

  하나의 sender와 하나의 receiver를 연결

- reliable, in-order byte stream

  하나의 connection이 열려있으면 그 사이에서 왔다갔다하는 데이터들을 따로 따로 생각하지 않고 바이트들의 연속으로 생각함

- pipelined

  → Go-back-N이나 selected repeat에서의 principle과 동일

  window size를 제어해서 congestion control과 flow control

- full duplex data

  → 하나의 connection을 열어서 양쪽에 data를 왔다갔다하게 함 → bi-directional data flow

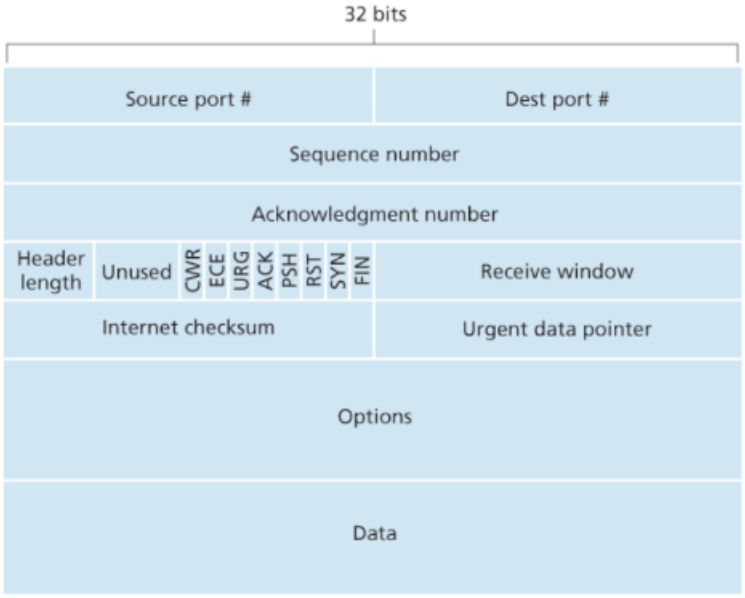
  MSS(Maximum Segment Size)

- connection-oriented

  데이터 보내기 전에 connection을 여는 과정 필요 (hand-shaking)

- flow controller

TCP segment structure



- acknowledgement number

  어디까지 잘 받았는지, 다음 기대하는 바이트를 적어놓음

- receive window

  flow control을 위해 존재 → 버퍼에 남아있는 공간을 알려줌

TCP sequence number, ACK's

- sequence number

  → byte stream이라 segment 데이터의 첫번째 byte를 적음

- acknowledgements

  → 그 다음 바이트를 적어서 보냄

  ex) MSS가 8일 때, 첫번째 패킷은 0부터 시작해서 ACK 0보내고 그 다음 기대하는 건 8byte 부터니까 ACK field에는 8이 적힘

데이터와 ACK을 같이 보냄 → piggybacking

→ 데이터와 컨트롤시그널을 같이 보냄

TCP round trip time, RTT

→ packet loss 때문에 timer를 설정

timeout 값은 RTT 보다는 크게 잡아야 함

→ RTT보다 작으면 쓸데없는 timeout이 많이 발생함

→ RTT보다 너무 크면 segment loss에 대한 reaction이 너무 느리게 일어남

RTT 추정

SampleRTT : segment transmission으로부터 ACK을 받기까지의 시간을 측정

→ 변할 수 있음 - sampleRTT 그대로 사용해서 timeout값을 정하면 계속 바뀔 수 있음

⇒ sampleRTT smoothing → sampleRTT를 평균 냄

ExpectedRTT = $(1-\alpha)EstimatedRTT + \alpha sampleRTT$

→ Exponential weighted moving average

timeout interval : ExtimatedRTT + "safety margin" → 표준편차를 더함

→ $DevRTT = (1-\beta)DevRTT + \beta|sampleRTT - EstimatedRTT|$

$timeoutInterval = EstimatedRtt + 4DevRTT$

TCP reliable data transfer

- pipelined segments
- cumulative acks
- single retransmission timer → Go-back-N과 비슷

retransmission → timeout event나 duplicate ack발생 시

TCP sender events

- data rcvd from app

  → application에서 데이터 내려오면 sequence number를 적어서 segment를 만듦

  (sequence number : byte stream number)

  → timer가 돌고있지 않으면 시작함

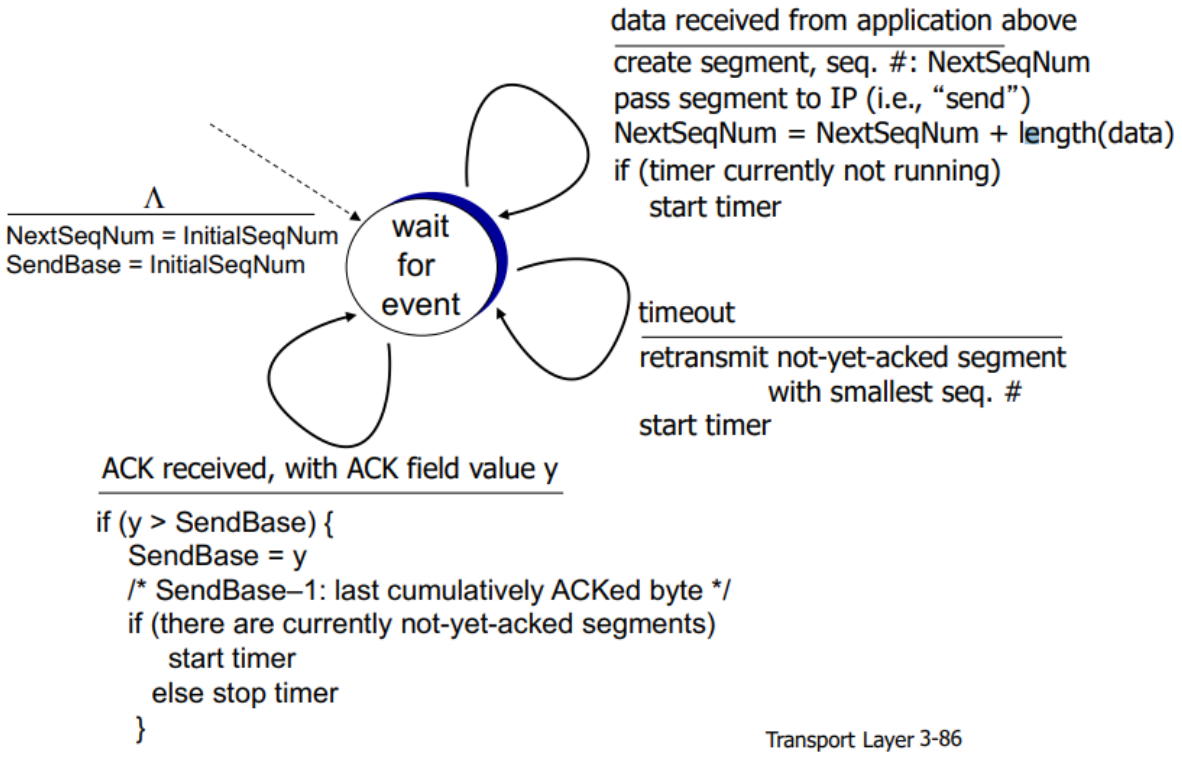
  ⇒ 가장 오래된 ACK 받지 않은 segment에 대해 타이머로 생각

  timeout interval이 넘어가면 타이머가 사라짐
- timeout

  → timeout 발생하면 segment retransmit하고 timer를 재시작
- ack rcvd

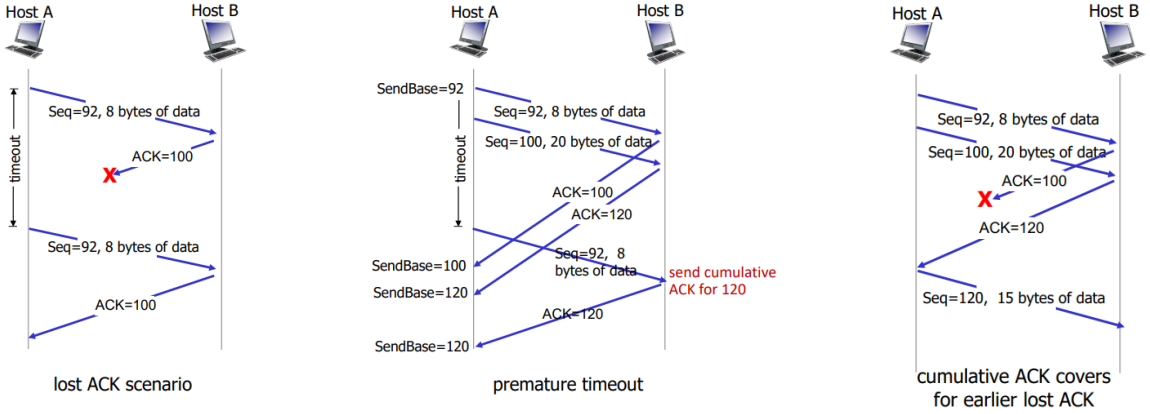
  → 현재까지 받은 segment update

  ack 받지 못한 segment가 있으면 timer를 재시작



timeout되면 아직 ACK 받지 못한 segment 중 가장 작은 sequence number를 가진 segment를 재전송

TCP: retransmission scenarios



→ 3번째는 다 잘 받아졌구나라고 생각

TCP ACK generation

- event at receiver
  1. expected sequence number를 가지고 in-order segment가 도착

→ 이미 ACK 받아진 모든 데이터를 보냄

TCP receiver action

⇒ delayed ACK, 다음 segment가 그때까지 없으면 ACK을 보냄

2. expected sequence number를 가지고 in-order segment가 도착

   → 하나의 segment가 ACK을 기다리고 있음

   TCP receiver action

   ⇒ 바로 cumulative ACK을 보냄

3. expected sequence number보다 큰 out-of-order segment가 도착

   TCP receiver action

   ⇒ 바로 duplicated ACK을 보냄

4. gap을 채우는 segment가 도착

   TCP receiver action

   ⇒ 바로 ACK을 보냄

TCP fast retransmit

이전까지는 timer가 expire되면 재전송

→ 중간에 packet loss가 일어나고 그 이후로는 계속해서 다 잘 도착함 → 기대하는 byte 적어서 보냄

⇒ congestion은 아닌데 해당 packet만 잃어버렸다고 판단

~ 잃어버렸으니까 해당 packet만 빠르게 다시 보냄 (timeout을 기다리지 않고)

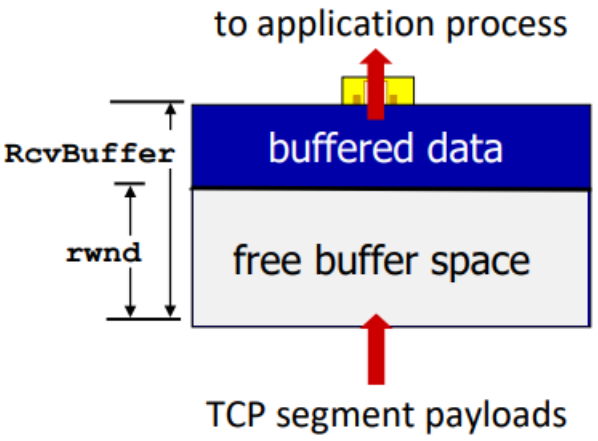
이전에 받은 ACK을 계속 받으면 fast retransmit을 진행

→ 같은 data에 대해서 중복 ACK받으면 가장 작은 sequence number를 가지면서 ACK 받지 않은 segment를 다시 보냄

TCP flow control

TCP에서 TCP socket buffer에 데이터 적으면 application layer에서 이걸 읽어감

→ flow control : receiver가 감당할 수 없을 정도로 너무 빠른 속도로 보내는 걸 방지



buffered data → 위로 올라가려고 기다리는 data

→ receive buffer의 size는 OS가 자동으로 관리

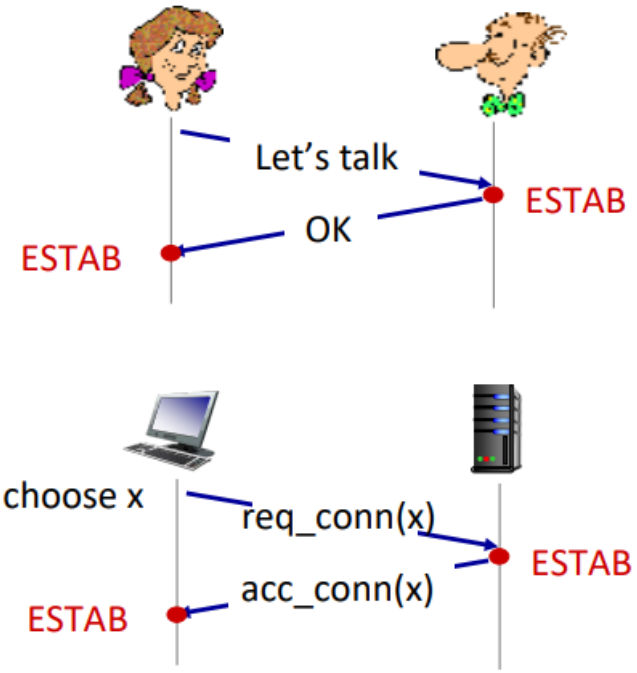
TCP header에 receive window를 적어서 보내는 field가 있음

→ 내 버퍼에 이만큼 공간이 있음 ⇒ 한 번에 이것보다 더 많이 보내지 말 것

congestion window update할 때 receive window 값을 넘지 않도록 설정

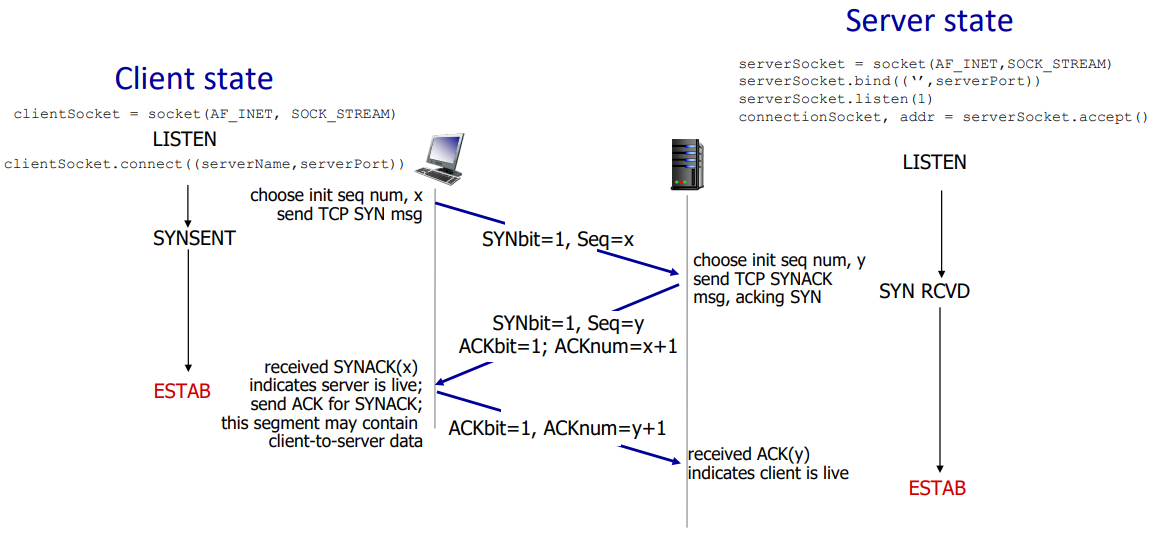
Connection management

- 2-way handshake



  server와 client가 sequence number를 synchronize하려면 3-way handshake가 필요함

- 3-way handshake

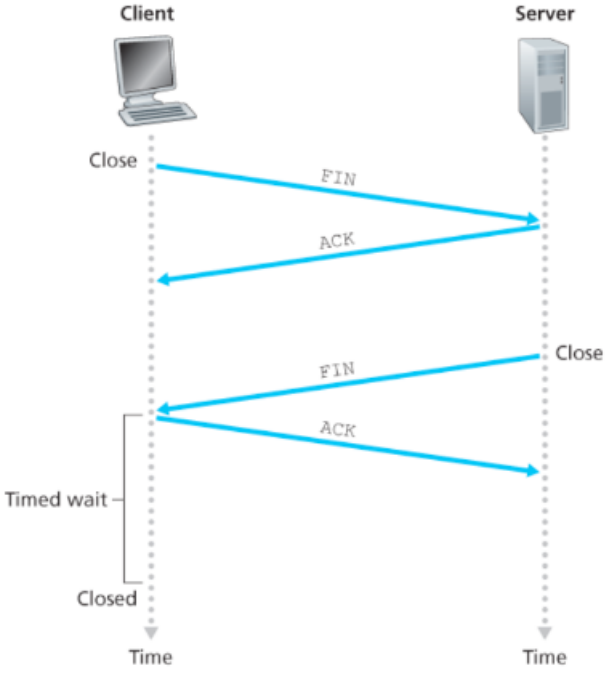


- SYN 설정하고 sequence number 적어서 보냄
- ACK로 그 다음 기대하는 segment 적어 보낸까 sequence number가 synchronize됨

TCP : closing a connection

client와 server가 connection을 끊으려고 할 때 FIN bit을 1로 설정

→ FIN을 받으면 이에 대한 ACK을 보냄 ~ 서로 FIN을 주고받아야 함

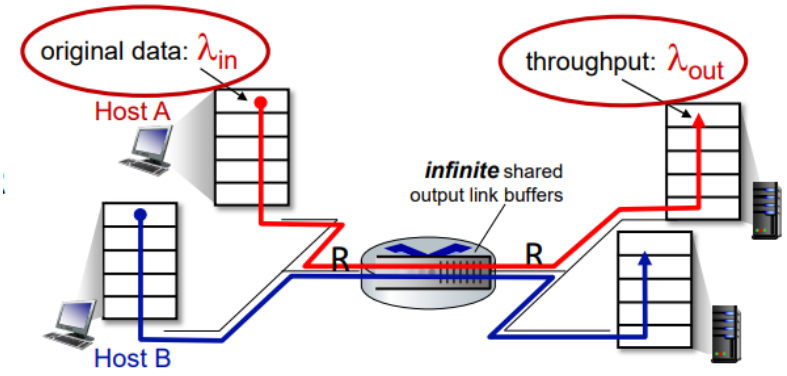


Principles of congestion control

flow control은 receiver가 감당할 수 없을 정도로 보내는 것을 제어

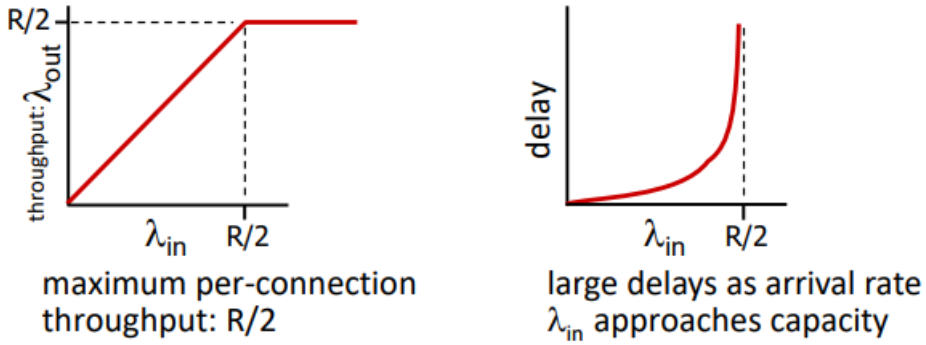
congestion control은 network이 감당할 수 없을 정도로 보내는 것을 제어

→ packet을 잃어버렸는지 확인 / delay가 발생했는지 확인 ⇒ 이걸 바탕으로 보내는 속도를 제어

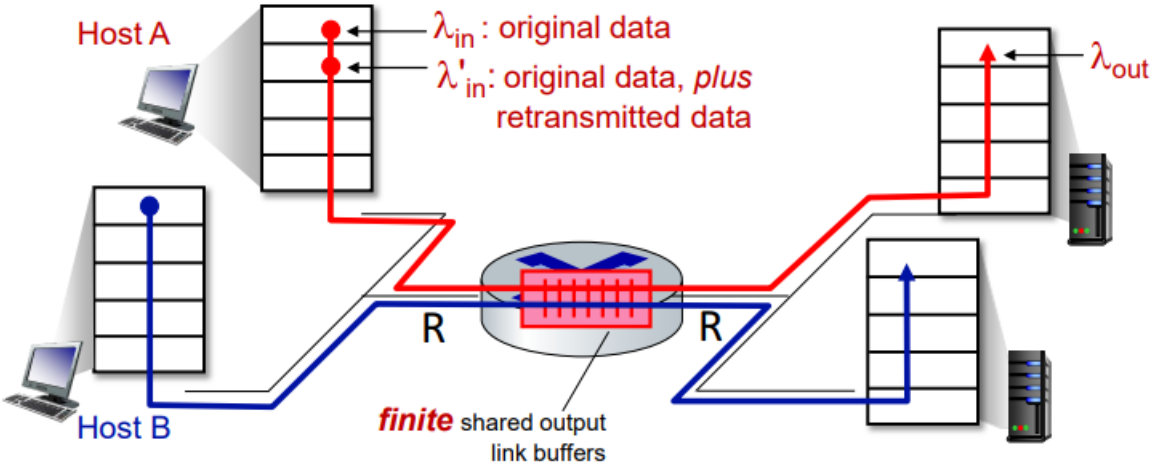


infinite buffer라서 packet loss는 일어나지 않을 것

두 링크에 대해 fair share한다고 사정



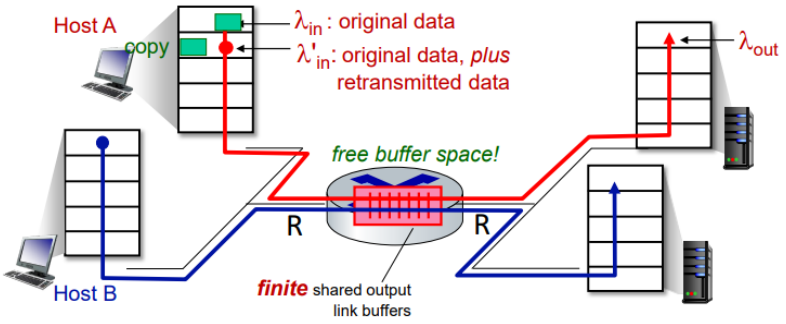
밀어넣는 속도가 R/2에 가까울수록 delay가 길어짐



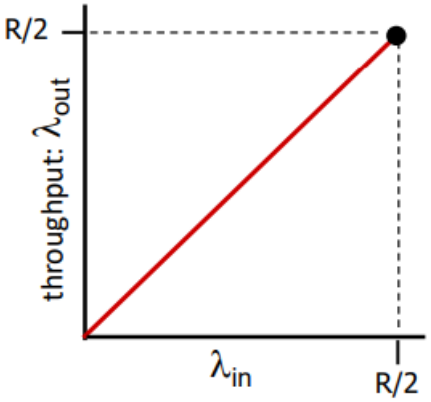
finite buffer → packet loss가 일어날 수 있음

sender는 timeout이 일어난 packet에 대해서만 retransmission

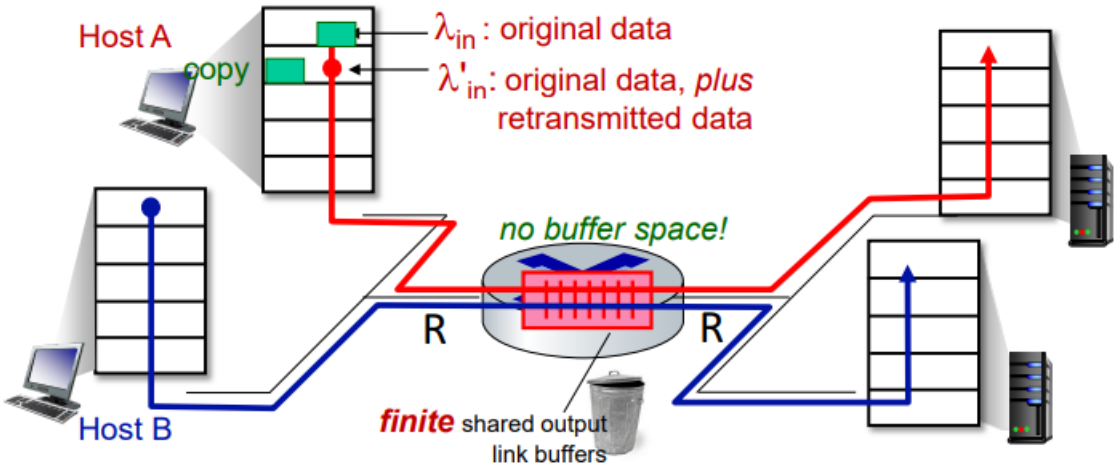
$I_{in} >= I_{out}$ → 재전송될 packet도 있으니 I_{out}은 재전송된 packet 또 갖는건 아님



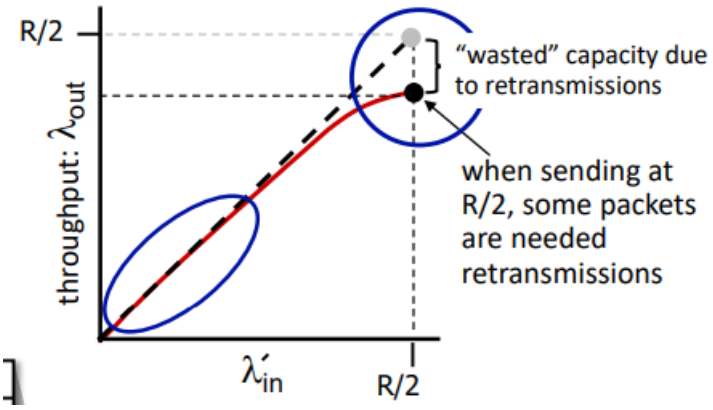
sender가 buffer가 언제 available한 지 정확하게 알 수 있다고 가정 → 버퍼에 공간이 있을 때만 보냄



보내는 속도가 R/2를 넘지는 않음 (공간이 있을 때만 보내기 때문에)



packet loss가 일어날 수 있는데 일어났다는 것을 알고 잃어버린 packet만 다시 보냄



R/2만큼 밀어넣을 때 throughput은 R/2가 안될 수 있음

→ packet loss로 재전송되는 packet이 있을 수도 있기 때문에

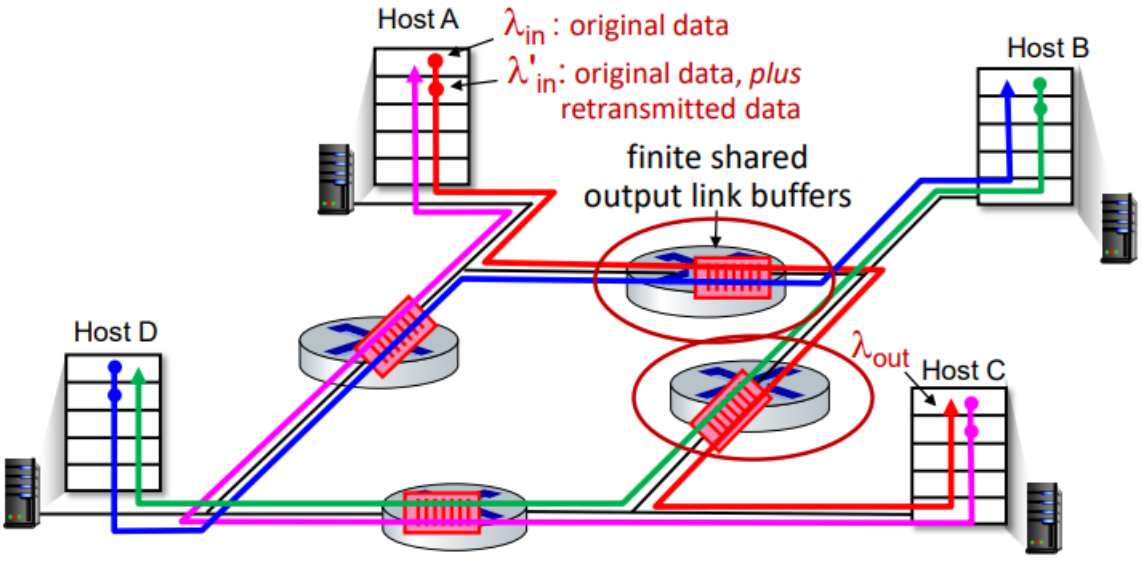
R/2보다 더 밀어넣는 경우에는 throughput이 R/2가 될 수 있음

→ 실제로는 duplicate packet을 보낼 수 밖에 없음

cost of congestion

→ 주어진 goodput에 대해 transmission이나 어떤 packet을 잃어버렸는지 등의 방법이 필요

→ 필요하지 않은 재전송이 발생할 수 있음



I_{in}과 I'_{in}이 계속 증가하면 맨 위 버퍼가 가득하게 됨 → loss가 일어남

⇒ 빨강, 파랑 있는 connection의 sender가 재전송

? 만약 빨강 connection이 빠르면 파랑보다 buffer에 빨리 넣어짐 → 파랑은 또 packet loss가 일어남

⇒ 속도를 제어하지 않고 계속 올리면 buffer를 어떤 connection이 독점해버리는 일이 생겨날 수 있음

→ throughput이 전부 다 0으로 떨어질 수도 있음 = congestion collapse

congestion control ⇒ window size 조절(크면 더 높은 속도로 밀어넣을 수 있음)

TCP congestion control

- additive increase,multiplicative decrease

sender는 transmission rate을 증가시킴 (window size 제어), 사용간으한 속도 (bandwidth) 조금씩 올려봄

(⇒ window size : ACK을 보내지 않고 보낼 수 있는 packet의 최대 개수)

loss가 일어나면 또 조금 줄임 → 사용할 bandwidth가 얼마나 되는지 알아봄

→ additive increase : congestion window를 loss가 감지되기 전까지는 매 RTT마다 1MSS증가

→ multiplicative decrease : loss가 감지되면 congestion window를 절반으로 줄임

- sender는 transmission을 제한 → LastByteSent - LastByteAcked ≤ congestion window
  congestion window는 고정적이지 않고 network congestion에 적응적으로 바꿈
- TCP sending rate → cwnd byte만큼 보내고 RTT 동안 ACK 기다리고 byte를 더 보냄
  $rate = \frac{cwnd}{RTT} bytes/sec$ → cwnd 키우면 속도가 높아짐

TCP slow start

→ 낮은 속도에서 출발

connection 시작하면 속도를 지수적으로 증가시킴

→ 1MSS : 매 RTT마다 cwnd가 두배가 됨 ⇒ 1ACK 받을 때마다 1MSS씩 증가

TCP: detecting, reacting to loss

- timeout에 의한 loss 발생
  → cwnd를 1MSS로 줄임 (처음 상태로 돌아감)
  ⇒ window는 일정 threshold까지 exponentially하게 증가하다가 linearly하게 증가
- duplicated ACK이 3개 이상 도착 → loss로 판단
  → TCP RENO 버전은 cwnd를 절반으로 줄임

TCP : switching from slow start to CA

slow start → 일정수준까지 exponential하게 증가

⇒ 넘어서면 linear → loss 발생하면 1로

~ threshold를 loss 일어나기 직전 cwnd 절반으로 바꿈

ex) cwnd = 4MSS → 4개의 segment를 보냄

매번 $\frac{1MSS}{cwnd}$만큼 증가

TCP throughput

loss가 나면 절반으로 줄어듦

average TCP throughput = $\frac{3}{4}\frac{W}{RTT}$ ⇒ $(W+\frac{W}{2})\frac{1}{2}$

TCP over “long, fat pipes”

→ RTT가 크면서 물리적으로 긴 링크(pipe)

TCP throughput = $\frac{1.22MSS}{RTT\sqrt{L}}$

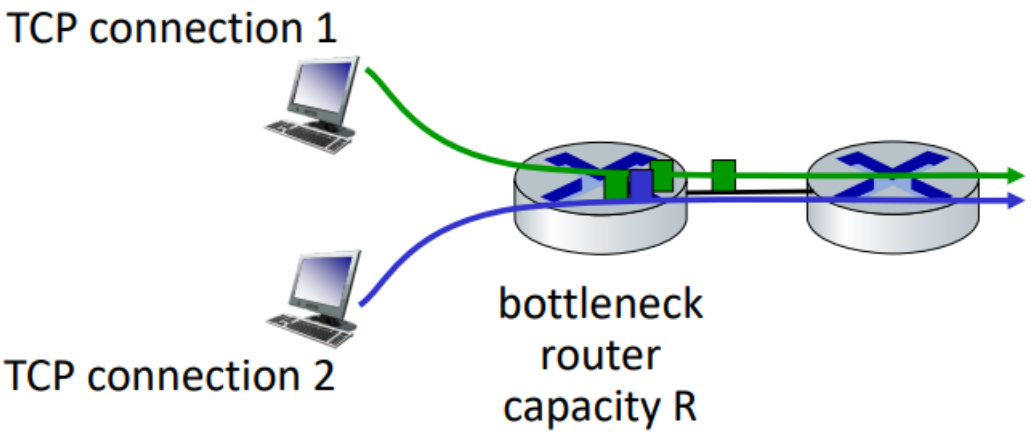
L : segment loss probability 낮을 수록 throughput 높음

→ segment loss가 자주 발생하면 window size가 커질 수가 없음

TCP fairness

fairness goal : k개의 TCP session이 같은 bottleneck 링크를 공유

각각의 average rate = $\frac{R}{K}$ ⇒ 공평하게 나눠가져야 함



fairness & UDP

→ UDP는 rate control를 하지 않음

parallel connection 열어서 둘 간의 연결이 더 빠르다고 느낄 수 있음

Explicit congestion notification(ECN)

network-assisted congestion control

→ IP header에 2bit field 추가 → network 안에서 congestion일어나면 field에 표시