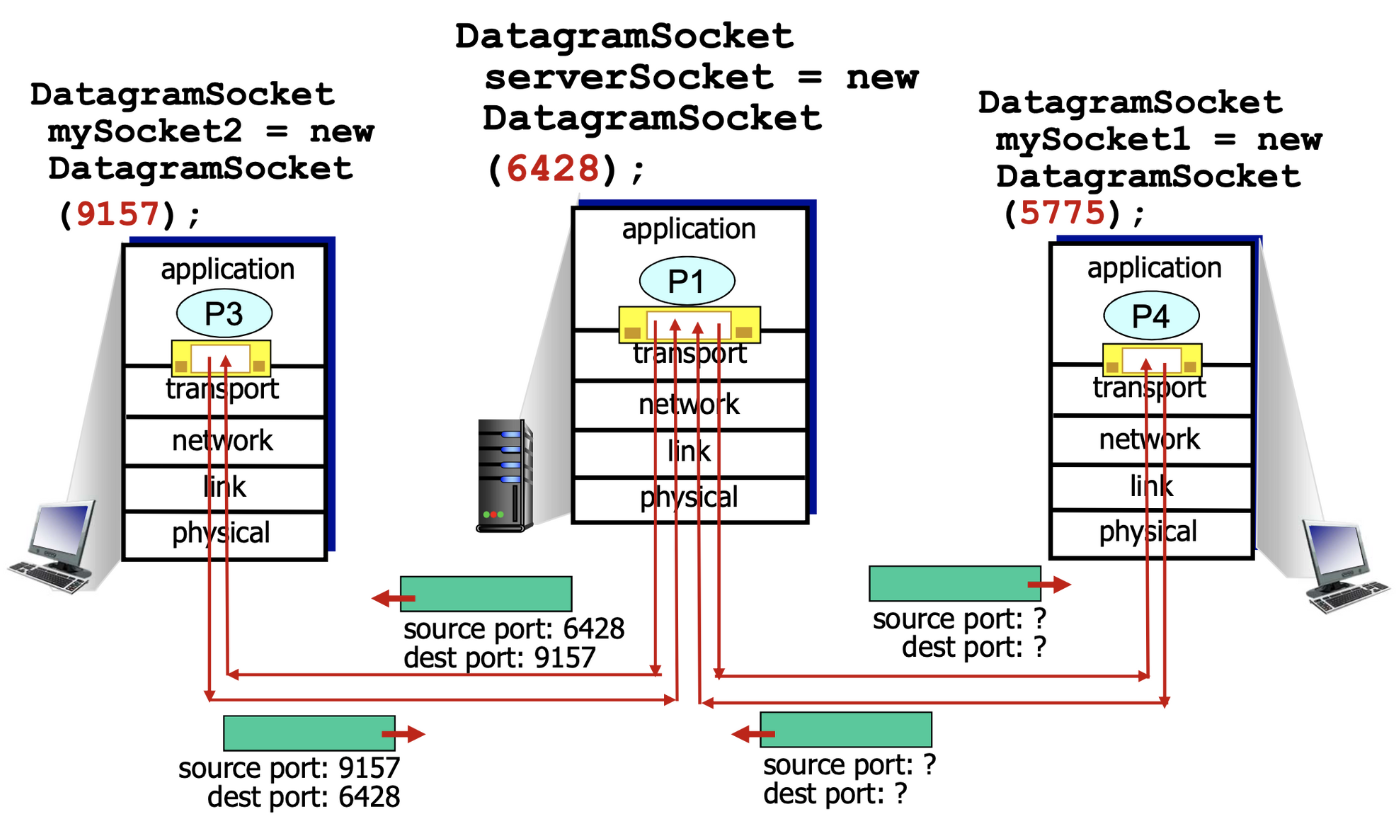
transport protocols run in end systems

network layer: end(host) to end routing using IP address

transport layer: process to process connection using port #

UDP에서는 여러 클라이언트가 서버의 같은 소켓으로 통신

TCP는 하나의 클라이언트가 하나의 소켓으로 통신



각 프로세스는 자신의 IP주소와 port #를 넣은 socket을 연다.

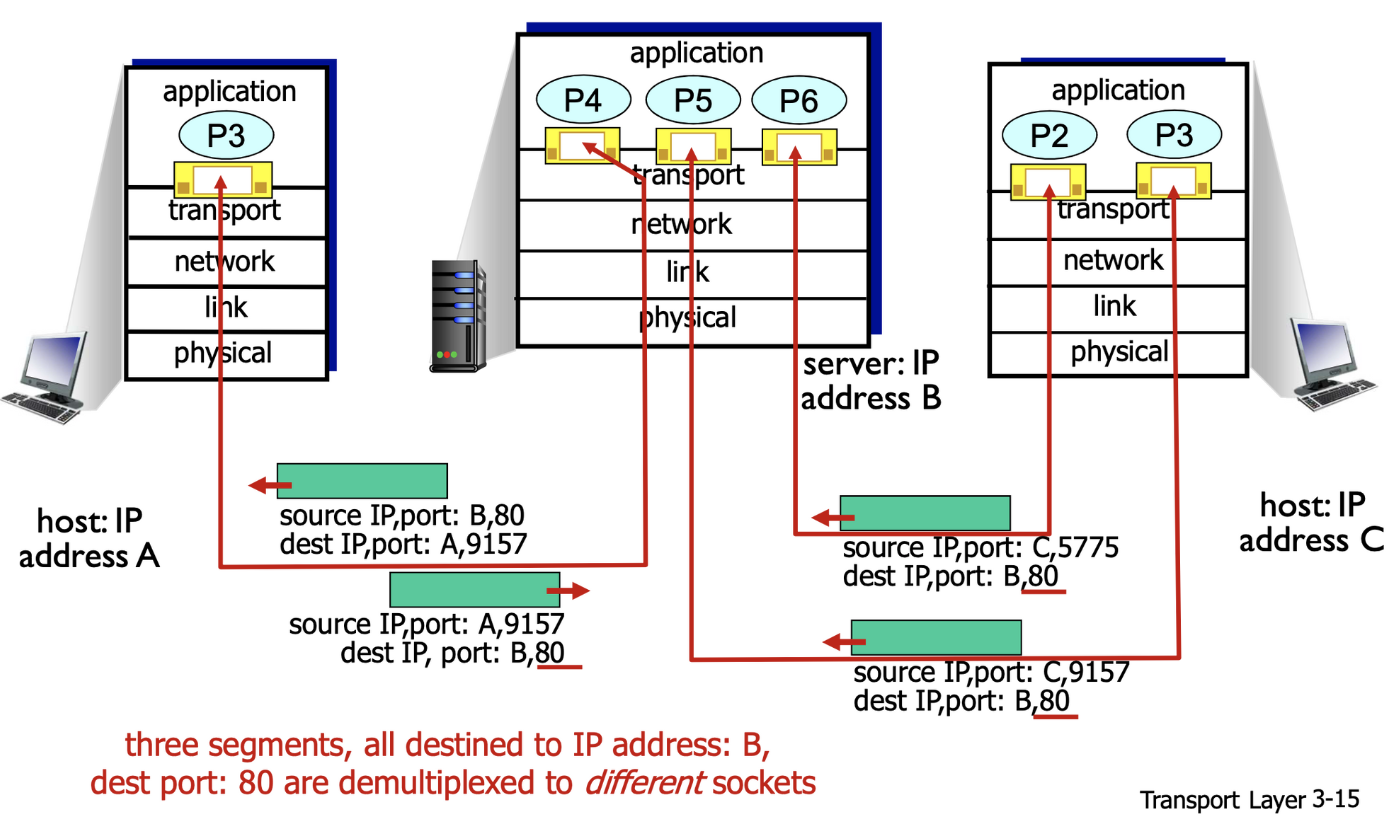
P3가 P1에게 보낸 데이터의 UDP Segment에서 source port는 9157, dest port는 6428.

6428 port를 연 P1 서버는 port #만으로 자신에게 온 메시지인지 확인할 수 없기 때문에 (같은 port를 연 다른 서버일 수 있음) IP주소를 transport 계층까지 올려 IP주소와 port #로 소켓을 찾는다. (Layering이 완벽히 이루어지지 않음)

IP주소는 network 계층만 보는 것이 아니라, 보고 한 계층 더 올려줌

= socket을 identify하는데 3계층 파라미터를 사용

P1 서버는 자신에게 온 데이터에서 source IP와 source port #를 보고 응답.



(가운데의 서버에는 항상 연결을 대기하는 welcome socket이 안 그려져 있음)

서버는 Client process들 당 전부 다른 소켓을 갖고 parallel하게 처리

Client process는 모두 같은 destIP, dest port #를 갖고 있을지라도, source 정보가 다르기 때문에 다른 소켓으로 처리 가능

timer: 안 오면 언제까지 기다려야 하나? 시스템의 오버헤드

timeout = 잃어버렸다고 판단 → retransmit

Timer의 비효율성

premature timeout → retransmit, duplicated data

ACK loss → retransmit, duplicated

Pipelining & Window

Pipelining: 한 번에 여러 개 보낼 수 있다.

TCP sliding window, cumulative ACK : ACK을 받지 않아도 w만큼 data 전송

cf. TCP는 full-duplex comm.를 support (보내면서 받기)

pipelined:

window size안에서 여러 개의 TCP segment가 ACK없이 전송 되는 것

69) reliable data transfer를 구현할 때 수신측에서 이미 정상적으로 받은 데이터를 다시 수신하게 되는 경우 두 가지를 설명하시오.

⇒

sender로 ACK이 도착하기 전 timeout이 걸린 경우

sender로의 ACK이 손실 된 경우

e) TCP는 application 계층에서 보낸 메세지 단위로 loss를 감지하고 재전송을 수행한다.

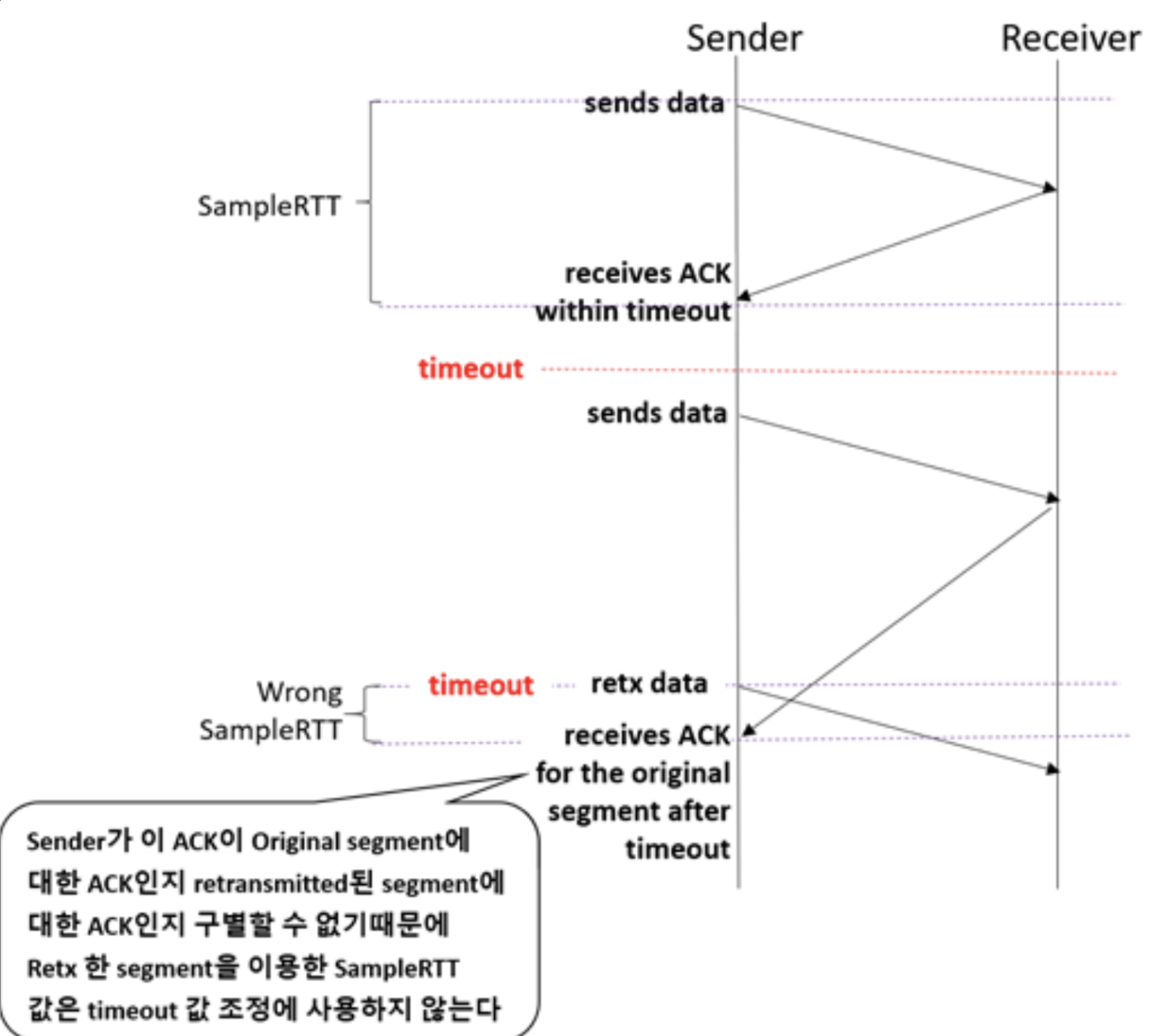
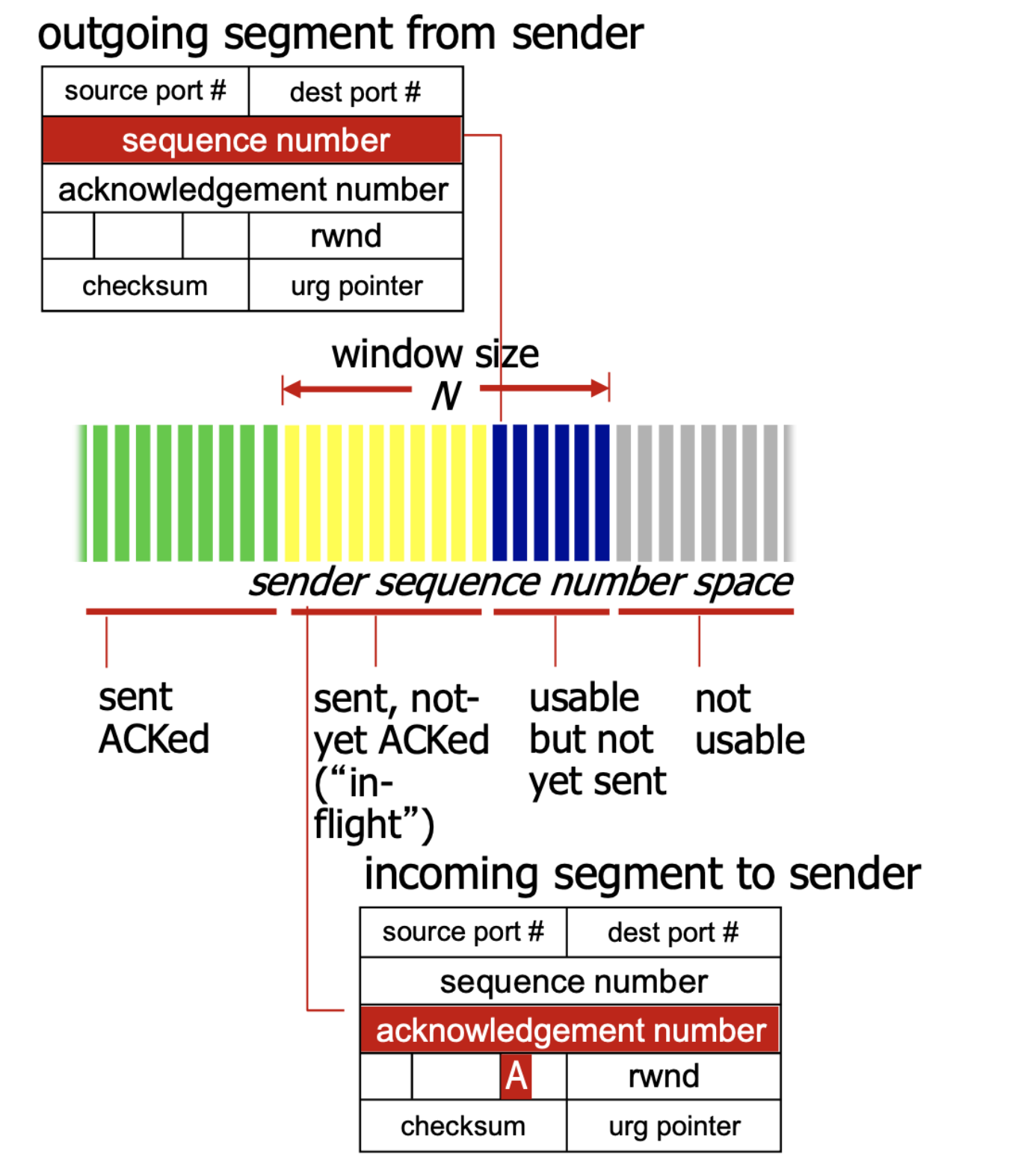
⇒ F. 메시지를 byte 단위로 numbering해 loss를 감지하고 재전송을 수행한다.

Ex. 호스트 A가 호스트 B로부터 0~535 바이트를 포함하는 세그먼트와 900~1000 바이트를 포함하는 또 다른 세그먼트를 수신할 경우, B에 대한 A의 ACK 번호는 536을 가진다. TCP는 스트림에서 첫 번째 잃어버린 바이트까지의 바이트들까지만 확인 응답 ⇒ cumulative ack

순서가 틀리게 도착한 세그먼트(900~1000)는 어떻게 하는가?

TCP RFC는 여기에 어떤 규칙도 부여하지 않았고, 이는 개발자의 몫

1. 수신자가 순서가 틀린 세그먼트를 즉시 버리거나 (2) 보유하고, 빈 공간에 잃어버린 데이터를 채우기 위해 기다린다.



SampleRTT는 어떤 시점에서 전송되었지만 현재까지 ACK이 없는 세그먼트 중 하나에 대해서만 측정되며, 재전송한 세그먼트에 대해선 계산하지 않는다.

ender TCP가 retransmission을 결정하는 두 가지 방법

1.timeout events

2.triple duplicate acks (3 dup ACK)

(b) Sender TCP의 타이머는 가장 최근에 보낸 segment의 첫번째 바이트를 가리키고 있다.

⇒ F. 가장 마지막으로 ACK을 받은 segment의 다음 segment (첫번째 inflight segment)

(c) Sender TCP가 3 Dup ACK으로 loss를 판단한 경우가 timeout으로 loss를 판단한 경우 보다 네트워크 상황이 더 나쁘다.

⇒ F. 반대, timeout으로 loss를 판단한 경우가 네트워크 상황이 더 나쁘다.

(e) TCP 가 connection을 맺을 때마다 sequence number 는 항상 초기값 0으로 시작한다.

⇒ F.

(72) 다음 빈칸을 send 혹은 receive 중 하나를 넣으시오.

TCP가 사용하는 ( send ) buffer는 수신 TCP가 순서에 맞게 segment들을 받았는지 확인될 때까지 보낸 segment를 저장하고 있다가 필요시 재전송을 위해 필요하며, ( receive ) buffer 는 sender TCP가 data를 보내는 속도와 상위 어플리케이션이 그 data를 가져가는 속도의 차이를 보완하기 위해 사용된다.

(75) TCP sender가 timeout 값을 지나치게 작게 잡으면 (premature timeout) 어떤 현상이 일어나는가?

⇒ ACK이 도착하기 전 timeout이 발생해 retransmission을 하게 된다. --->loss로 오판하는것 (not loss)

(77) 슬라이드 37번에서 정의한 SendBase와 NextSeqNum를 사용한다고 가정한다. 현재 SendBase = 800이고 NextSeqNum가 1150 일 때, 수신한 TCP segment의 ACK#가 1050이고 5계층에서 200 바이트가 전송되기 위해 내려왔다고 가정하면 SendBase와 NextSeqNum는 어떤 값으로 update 되는가? (MSS는 100이라고 가정함)

⇒ SendBase: 1050, NextSeqNum: 1250

Sendbase가 800, NextSeqNum이 1150이므로 window size는 최소한 1150-800=350보다 클 것이다. ACK# 1050이 도착해 SendBase가 1050으로 이동하므로 1050+350=1400번까지는 전송이 가능하므로 pipelining을 통해 연속으로 두 segment를 보내면 NextSeqNum은 1150+200=1350

만약, 5계층에서 내려온 데이터가 110 바이트라면?

이때는 우선 100 바이트 segment를 전송하고 남은 10바이트는 (OS에 따라 구현이 다르나) 대부분의 TCP는 바로 전송하지 않고 버퍼에 저장해서 다음 내려오는 응용 메시지 (outgoing data)와 함께 MSS를 채워서 보냅니다. 그러나 언제까지나 기다릴 수는 없으므로 타이머를 사용해서 일정기간 동안 더 이상의 응용 데이터가 내려오지 않으면 버퍼에 남은 데이터를 전송하게 됩니다.

여기서 타이머는?

만일 SendBase=890이고 NextseqNum=1150일 때는 timer가 890byte에 걸려 있었는데, ACK#=1050을 받고 나서는 timer가 1050 byte를 가리키고 re-start하게 됩니다.

그 다음 110 바이트가 5계층에서 내려왔고, MSS가 100이므로 한 segment를 전송하게 되어, NextSeqNum는 1150+100=1250으로 바뀝니다. 이때 timer는 5계층에서 응용 데이터가 내려온 것 과는 아무런 상관없이 1050byte에서 re-start 한 후 계속 흘러가고 있습니다.

(타이머는 SendBase)

따라서 window size는 최소 한 MSS (<MTU)보단 커야 한다.

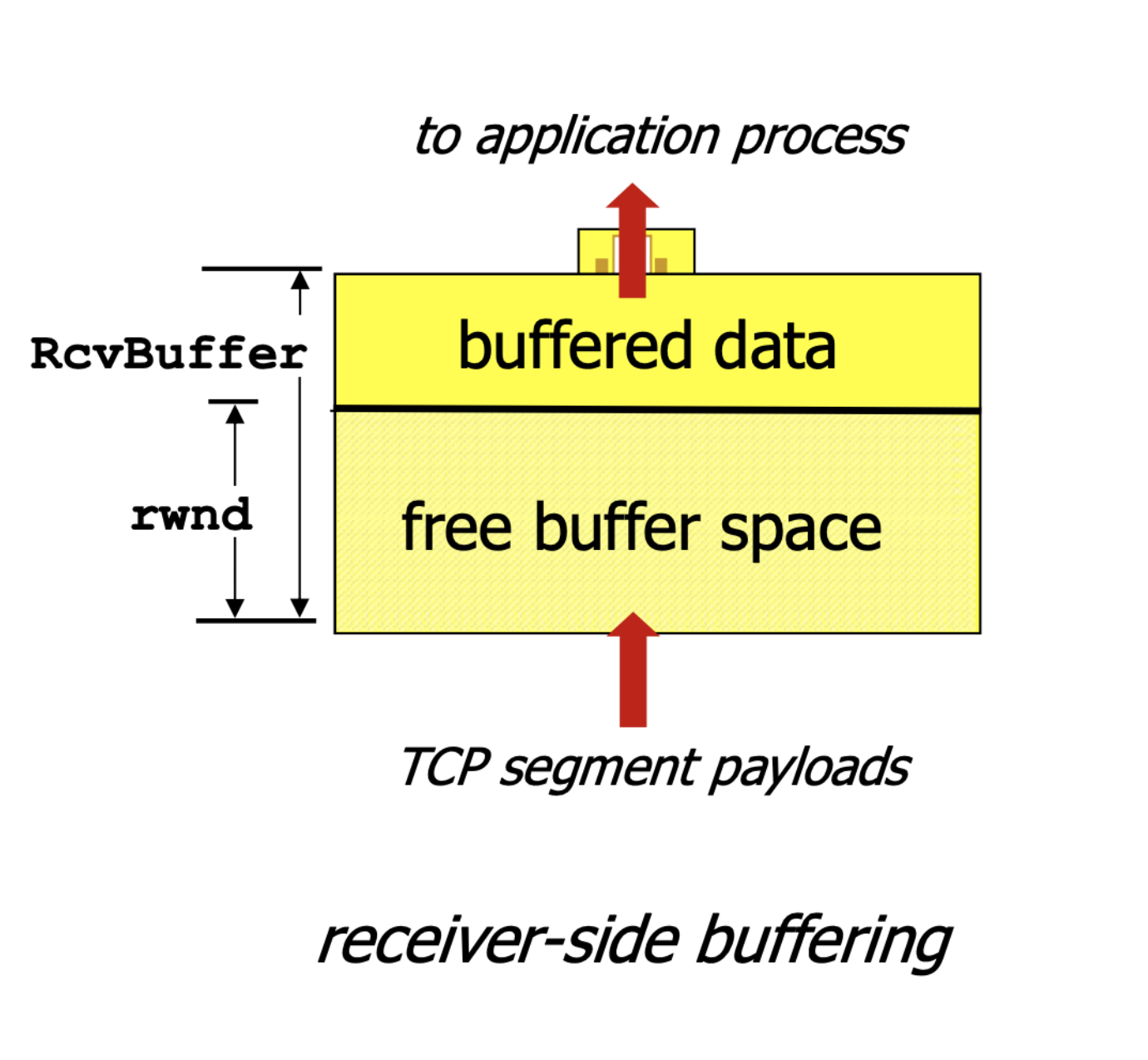
마지막 측정된 SampleRTT 가 1초였다면, update되는 TimeoutInterval값은 1초보다 크다.

⇒ F. 더 클 수도, 작을 수도 있다.

TimeoutInterval=EstimatedRTT+4\*DevRTT

CP는 in-order delivery, cumulative ACK이기 때문에 timer를 ACK이 안 된 세그먼트 중 가장 오래된 것(Oldest UnACKed Bute)에 붙이고 이것이 SendBase 가 됨. ( SendBase < NextSequenceNum 일 때만)

만약 SendBase =NextSequenceNum라면 inflight bytes가 없음 (노란색=전송 중인 UnAcked Bytes) → Timer 설정 X



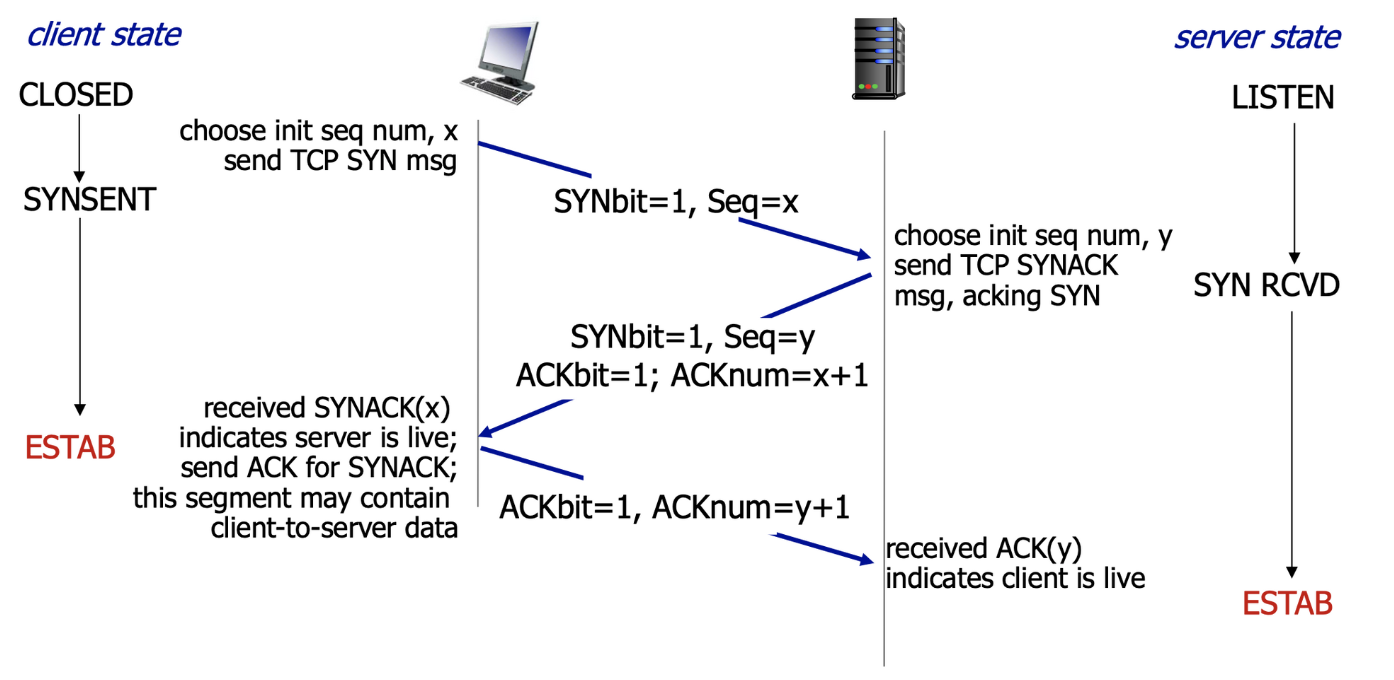
LastByteRead: 호스트 B의 애플리케이션 프로세스에 의해 버퍼로부터 읽힌 데이터 스트림의 마지막 바이트 수

LastByteRcvd: 호스트 B에서 네트워크로부터 도착하여 수신 버퍼에 저장된 데이터 스트림의 마지막 바이트 수

LastByteRcvd - LastByteRead ≤ RcvBuffer

rwnd 는 버퍼의 여유공간.

rwnd=RcvBuffer-(LastByteRcvd-LastByteRead)



클라이언트가 서버와의 연결을 초기화 하길 원할 때 서버가 클라이언트의 dedicated된 소켓을 여는 과정.

1단계: 먼저 클라이언트 TCP는 서버 TCP에게 TCP 세그먼트를 송신한다. 이 세그먼트는 애플리케이션 계층 데이터를 포함하지 않고, 세그먼트의 헤더에 SYN 이라는 하나의 플래그 비트를 가진다. 추가로 클라이언트는 최초의 SequenceNumber를 선택한다(client\_isn). 이것을 TCP SYN 세그먼트라고 한다. 이 세그먼트는 IP 데이터그램 안에서 캡슐화되고 서버로 송신된다.

SYNbit=1, Seq=x, ACKbit=0, ACK ACKnum=쓰레기값

2단계: TCP SYN 세그먼트를 포함하는 IP 데이터그램이 서버 호스트에 도착했을 때, 서버는 데이터그램으로부터 TCP SYN 세그먼트를 뽑아낸다. 그리고 연결에 TCP 버퍼와 변수들을 할당한다. 또한 클라이언트 TCP로 ACK 세그먼트를 송신한다. 이 ACK 세그먼트도 애플리케이션 계층 데이터를 포함하지 않는다.

그러나 3개의 중요한 정보를 포함하는데, 1) SYN 비트는 1로 설정된다. 2) TCP 세그먼트 헤더의 ACK 필드는 client\_isn+1 로 설정된다. 3) 서버는 자신의 최초의 SequenceNumber 를 설정하고(server\_isn), TCP 세그먼트 헤더의 SequenceNumber 필드에 이 값을 넣는다. 이 ACK 세그먼트는 "나는 당신의 최초 client\_isn를 가지고 연결을 시작하기 위해 당신의 SYN 패킷을 수신했다. 나는 이 연결 설정에 동의한다. 내 자신의 최초의 SequenceNumber 는 server\_isn 이다."라고 말하는 것이다. ACK 세그먼트는 SYNACK라고도 한다.

SYNbit=1, Seq=y, ACKbit=1, ACKnum=x+1

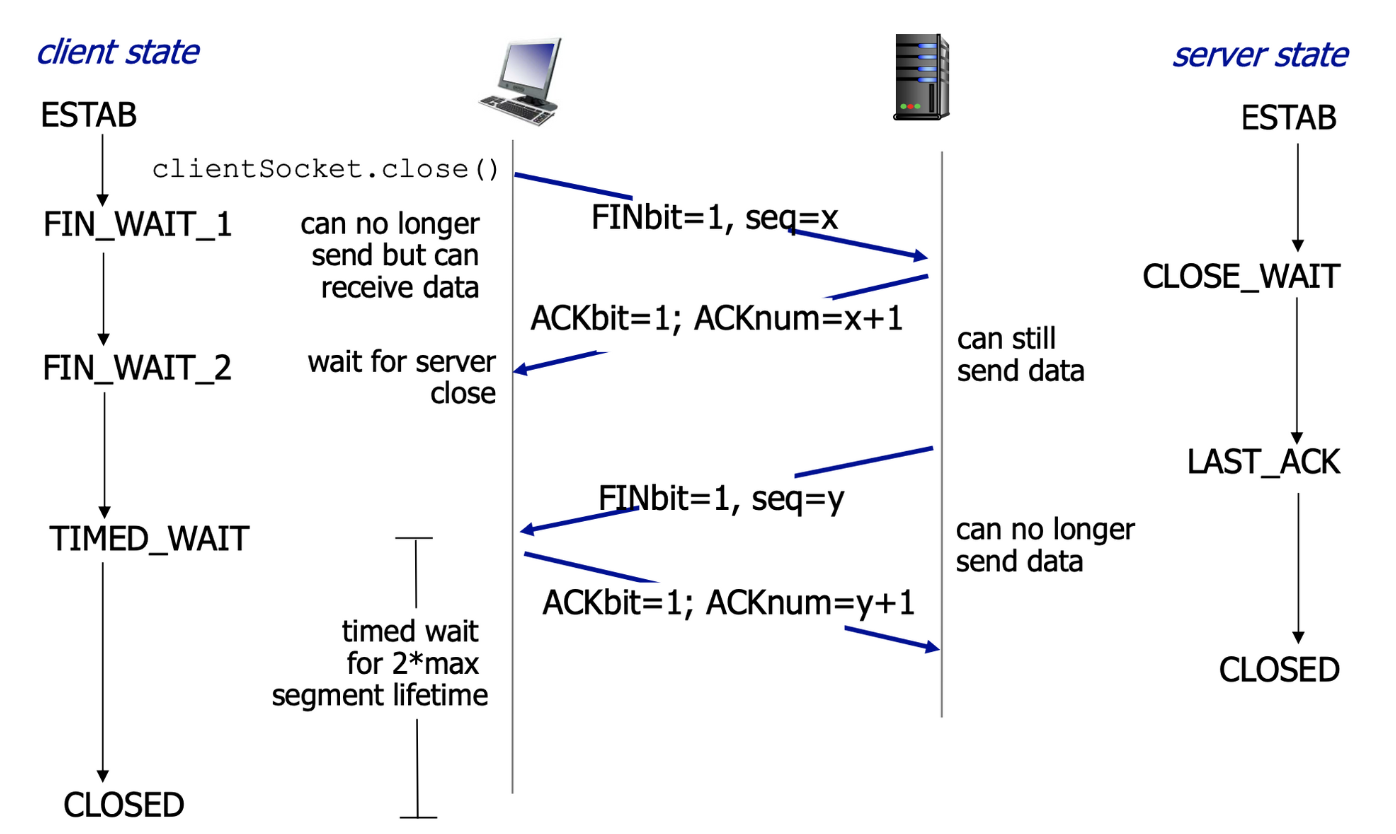
3단계: SYNACK 세그먼트를 수신하면, 클라이언트는 연결에 버퍼와 변수들을 할당한다. 그 다음 클라이언트 호스트는 서버로 또 다른 세그먼트를 송신한다. 클라이언트는 TCP 세그먼트 헤더의 ACK 필드 안에 server\_isn+1값을 넣고, 이 마지막 세그먼트가 서버의 ACK 세그먼트를 확인한다. 연결이 설정되었기 때문에 SYN 비트는 0으로 설정된다. 3-way handshake의 세 번째 단계는 클라이언트에서 서버로 세그먼트 페이로드에 데이터가 운반될 수 있다. (piggyback)

SYNbit=0, ACKbit=1, ACKnum=y+1

이 세 단계가 완료되면, 클라이언트와 서버 호스트들은 각각 서로에게 데이터를 포함하는 세그먼트를 보낼 수 있다. 이 다음의 세그먼트 들의 SYN 비트는 0이다.

random 한 SequenceNumber 를 설정하는 이유

이전의 connection으로 받는 데이터로 오인하지 않기 위해



TCP는 full-duplex 통신을 하기 때문에 Client의 send/recv buffer, Server의 send/recv buffer를 각각 닫아 줘야 함.

연결 종료를 결정할 땐, 클라이언트 애플리케이션 프로세스가 종료 명령을 내리고, 헤더의 FIN 플래그 비트가 1로 설정된 세그먼트를 보낸다.

⇒ 클라이언트의 send buffer 종료, 여전히 data를 받을 수는 있음

서버가 이 세그먼트를 수신하면, 서버는 클라이언트에게 확인 세그먼트를 보내고,

⇒ 서버는 여전히 data를 보내고 받을 수 있음

FIN=1로 설정된 자신의 종료 세그먼트를 다시 보낸다. (위와 함께 보낼 수 있음)

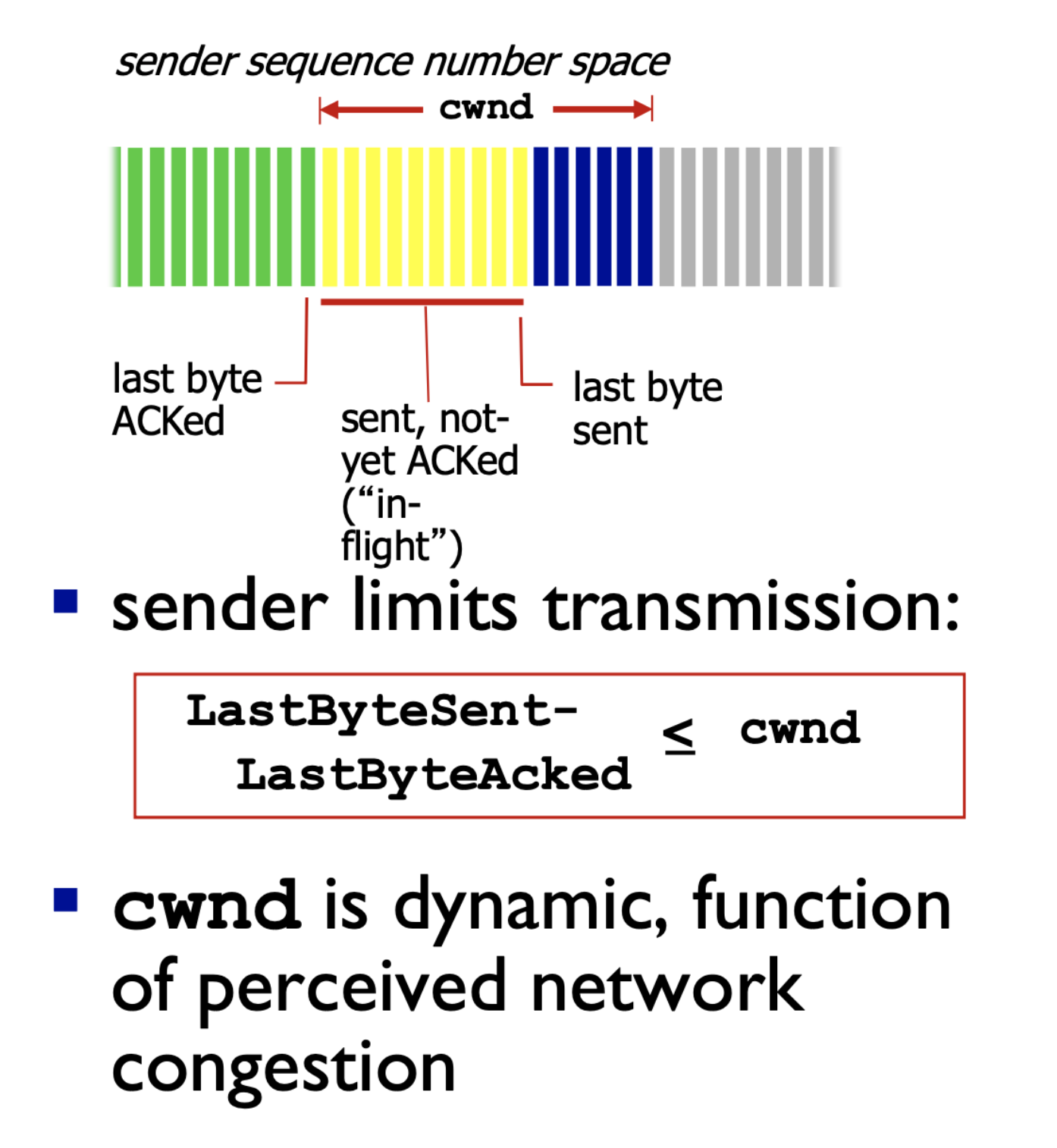
⇒ 서버의 send buffer 종료 요청.

마지막으로 클라이언트는 서버의 종료 세그먼트에 ACK을 한다.

⇒ 서버의 send buffer 종료, ACK의 loss를 대비한 retx를 위해 2분 정도 기다림.

클라이언트의 응답 없을 시 서버는 maximum number of retry 후 종료.

이 시점에서 두 호스트의 모든 자원들은 할당이 해제된다.



rwnd 무시하고 cwnd 를 Window size W로 봄. → 망 상황에 따라 실시간으로 cwnd 조절

TCP의 혼잡제어 방법은 네트워크 혼잡에 따라 연결에 트래픽을 보내는 sending rate을 sender가 제한하도록 하는 것이다.

만약, TCP sender가 자신과 dest 간의 경로에 혼잡이 없음을 감지하면 sending rate을 높이고, 혼잡을 감지하면 sending rate을 줄인다.

연결로 트래픽을 보내는 sending rate을 제한하는 방법

cwnd 로 TCP sender가 네트워크로 트래픽을 전송할 수 있는 비율을 제한한다. ACK이 안 된 데이터 양(inflight byte)은 cwnd 와 rwnd 의 최소값을 초과하지 않는다.

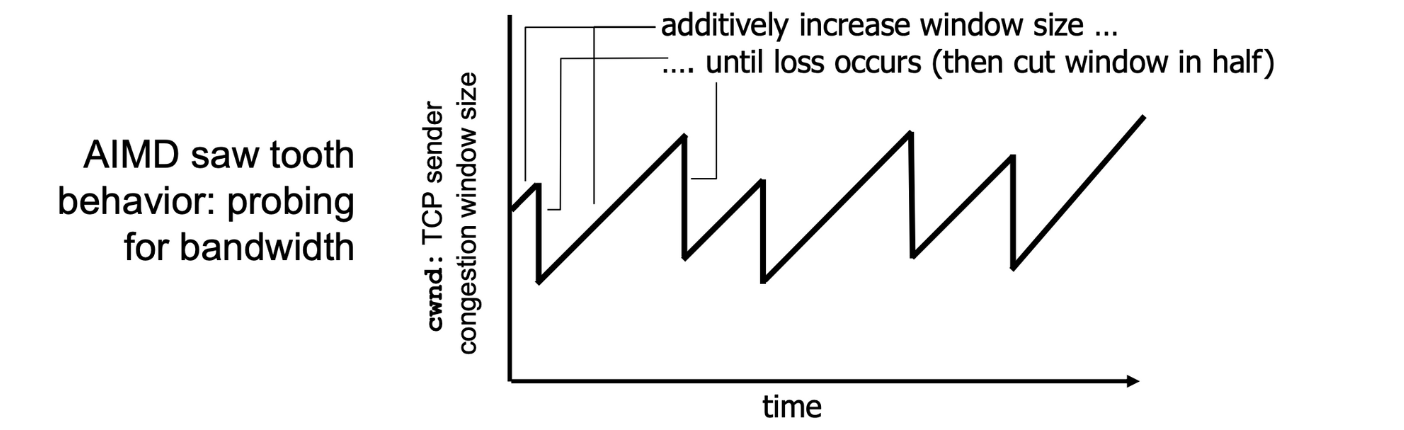
LastByteSent - LastByteAcked ≤ min {cwnd, rwnd}

매 RTT의 시작 때, sender는 cwnd 바이트 만큼의 데이터를 전송할 수 있고, RTT가 끝나는 시점에 데이터에 대한 ACK을 수신한다. → 한 RTT마다 cwnd에 따라 Window size를 조정

sender의 sending rate = cwnd/RTT (바이트/초)

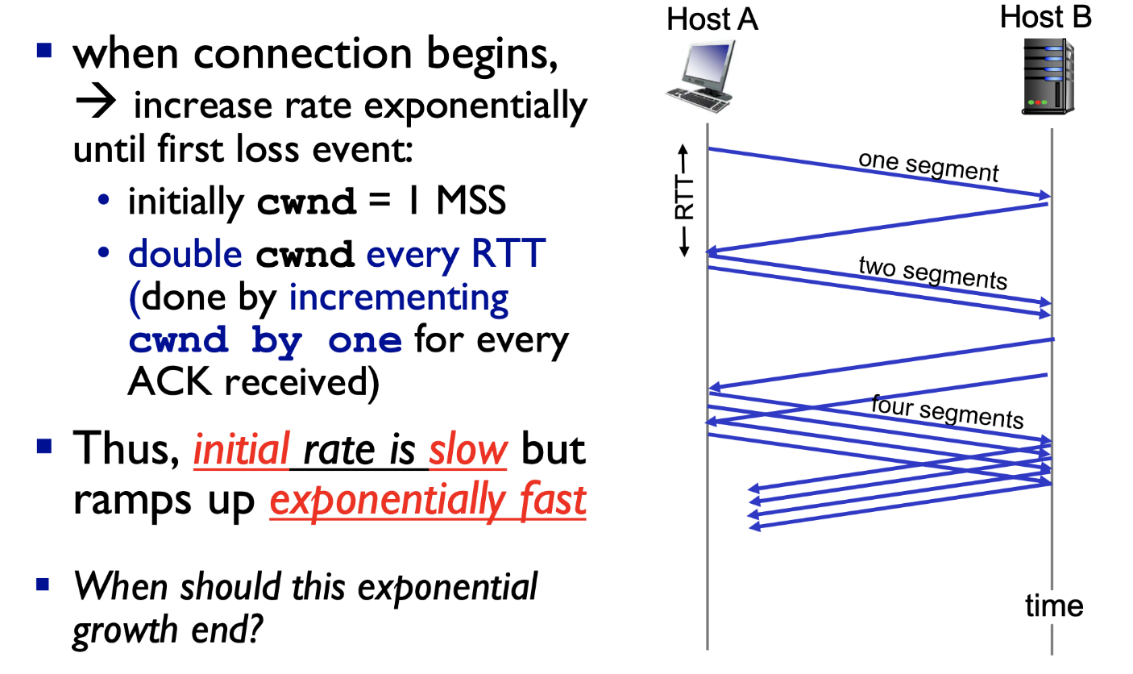
TCP sender가 자신과 dest 사이의 경로에 혼잡이 존재하는지 감지 하는 방법

Timeout 또는 3 dup ACK 수신이 발생하면, TCP sender에 loss가 발생한 것. → 혼잡 존재 감지



위 사진 TCP congestdion control:AIMD

sender는 loss가 발생할 때까지 transmission rate(Window size)를 1 MSS 단위로 증가시키다, loss 감지 시 cwnd를 절반으로 줄인다.

  
initial rate = 1 MSS (cwnd)로 시작해 (전송률: 1MSS/RTT) 한 세그먼트가 한 ACK을 받을 때마다 ACK당 1 MSS 씩 증가시킨다. (1→2→4→8→,,,) 언제까지? ssthresh에 도달할 때까지

loss가 생길 경우, ssthresh를 혼잡이 생긴 시점의 cwnd 값의 절반으로 정한다.

cwnd 값이 ssthresh 에 도달한다면 Slow Start를 종료하고 CA (혼잡회피모드)로 전환한다.

⇒ cwnd이 각 RTT당 1MSS씩 linear하게 증가한다.

if ( cwnd < ssthresh ) then

SS phase (double cwnd in every RTT)

else

CA phase (increment cwnd by 1 MSS in every RTT)

TCP RENO의 경우

Timeout으로 인한 loss ⇒ SS 시작 (추후 CA로 전환될 수 있음)

Timeout으로 인한 loss는 3 dup ACK보다 망 상황이 더 좋지 않다는 것이므로 SS를 시작

3 duplicate ACK이 온 loss ⇒ CA 시작

cwnd를 현재 cwnd값의 절반으로 줄인 후 1RTT당 1MSS씩 linear하게 증가시킨다.

TCP Tahoe

loss가 생길 경우, loss를 구별하지 않음 (3 dup ACK인지, Timeout인지)

(78) fast retx을 위해서 TCP sender는 duplicate ACK을 3개까지 기다려 본다. 1 혹은 2개가 아닌 3개까지 기다리는 이유는 무엇인가?

⇒

네트워크에서 drop되지는 않았으나, 늦게 도착하는 패킷 (delayed not lost packet)을 loss로 오판하여 불필요한 retx을 하는 것을 막기 위해서

(80) 만일 TCP sender가 timeout 값을 조정하지 않는다는 가정하에, 네트워크 라우터의 output buffer 크기를 증가시킨다면 throughput은 어떻게 (증가 혹은 감소) 변하고 그 이유는 무엇인지 설명하시오.

⇒ timeout 값을 조정하지 않고 네트워크 라우터의 output buffer 크기를 증가시키면 Queueing delay가 증가하고, 불필요한 retransmission으로 인해 결과적으로 throughput이 감소할 수 있다.

throughout 이 감소하는데 그 이유는 TCP sender 가 delayed packet 가 loss 된 것으로 오판하고 retx을 하게 되므로 이미 TCP receiver 가 받은 packet들이 중간 라우터의 output buffer를 차지하는 비율이 점점 증가하게 되기 때문이다.

(81) 호스트 A에서 일정 기간 동안 5계층 응용 프로토콜이 전송하는 메시지 량이 100Mbit였다고 가정할 때 UDP를 사용하는 경우보다 TCP를 사용하는 경우 라우터로 실제 전송된 데이터량은 100Mbit를 초과할 수 있다. 그 이유는 무엇인가?

⇒ TCP는 loss 발생 시 retransmission을 지원하므로, 라우터로 실제 전송된 데이터는 애플리케이션 계층에서 보내준 데이터와 재전송되는 데이터를 포함한다.

TCP는 send buffer를 두고 네트워크에서 loss 발생시 retx을 하므로 실제 5계층에서 응용 프로토콜이 전송하고자 하는 데이터 보다 더 많은 량의 데이터를 네트워크로 전송하게 된다.

82) 다음은 서술은 참인가? “TCP Sender가 네트워크로 전송한 데이터량이 증가하면 할 수록, TCP receiver의 throughput은 증가한다.”

⇒ F.

거짓, throughput 은 중복되어 전송된 bit들은 고려하지 않는다. 네트워크가 free한 경우에는 TCP sender가 전송한 만큼 receiver 측에서 throughput이 올라가나, 네트워크가 (라우터 output 버퍼가) 혼잡한 경우에는 네트워크로 전송한 데이터 중 불필요한(즉 TCP receiver 가 이미 받은) 데이터를 재전송하는 경우가 포함되어 있을 수 있기 때문에 TCP sender가 전송한 데이터량이 증가한다고 항상 throughput이 증가하지는 않는다.

Consider sending a large file from a host to another over a TCP connection that has no loss.

a. Suppose TCP uses AIMD for its congestion control without slow start. Assuming cwnd

increases by 1 MSS every time a batch of ACKs is received and assuming approximately

constant round-trip times, how long does it take for cwnd increase from 6 MSS to 12 MSS (assuming no loss events)?

⇒ 6RTT

b. What is the average throughput (in terms of MSS and RTT) for this connection up through time = 6 RTT?

⇒ (6+7+8+9+10+11)/6 = 51 MSS/6RTT = 8.5 MSS/RTT