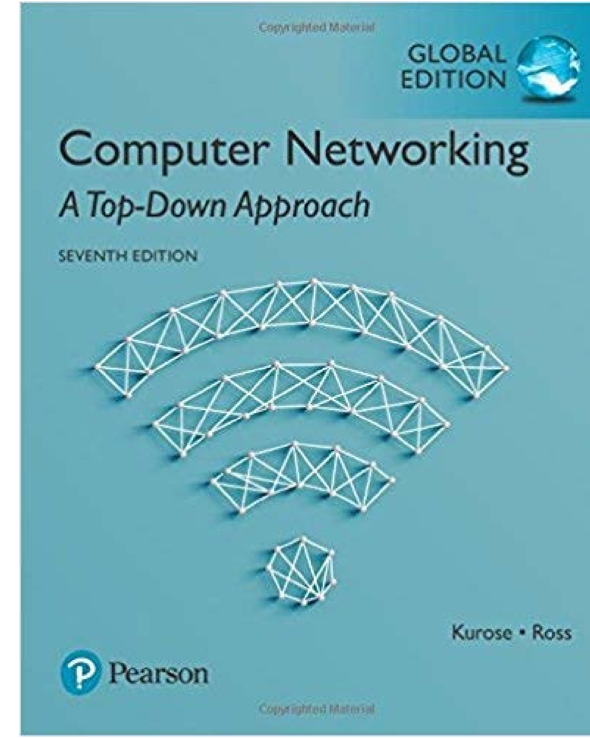
Chapter 3
Transport Layer part 3

School of Computing
Gachon Univ.
Joohyung Lee

Most of slides from J.F Kurose and K.W. Ross. And, some slides from Prof. Joon Yoo



Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson, 2017

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

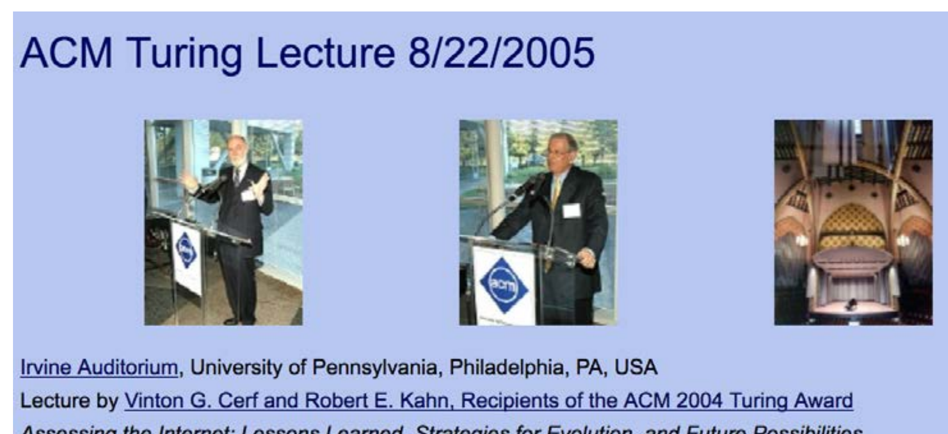
3.6 principles of congestion control

3.7 TCP congestion control

Ch.3.5.1 TCP/IP

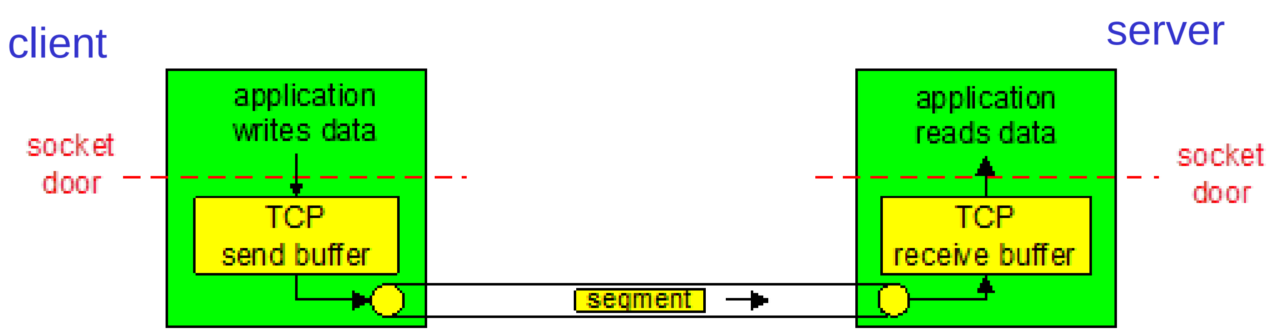
처음에는 TCP와 IP가 합쳐져 있었지만
나중에는 split된다.

- Vincent Cert, Robert Kahn and TCP/IP
  - Invented TCP/IP to interconnect networks
    - Initially a single protocol, later split into two parts: TCP and IP
  - Published a paper on IEEE Transactions on Communications Technology [1974]
  - TCP/IP protocol is the bread and butter of today's Internet
    - Devised before PCs, smartphone/tablets, Ethernet/WiFi, Web/Social media was developed
    - Provides support for applications, allows link-layer protocols to interoperate
  - ACM Turing Award in 2004



TCP: Overview

- Features from rdt protocols
  - error detection, retransmissions, cumulative acknowledgements, seq# and ack#, timers
- connection-oriented
  - 3-way handshake before data transmission
  - Initialize state variables (Ch. 3.7), resources



- full duplex data:
  - bi-directional data flow in same connection
- flow control, congestion control (Ch. 3.7)
- MSS: Maximum Segment Size
  - Maximum application-layer data in segment
  - MSS is determined by *link-layer* MTU (e.g. Ethernet 1500 bytes) (Ch. 4)

= 링크 속도

Ch. 3.5.2 TCP segment structure

TCP header

Will be discussed in detail

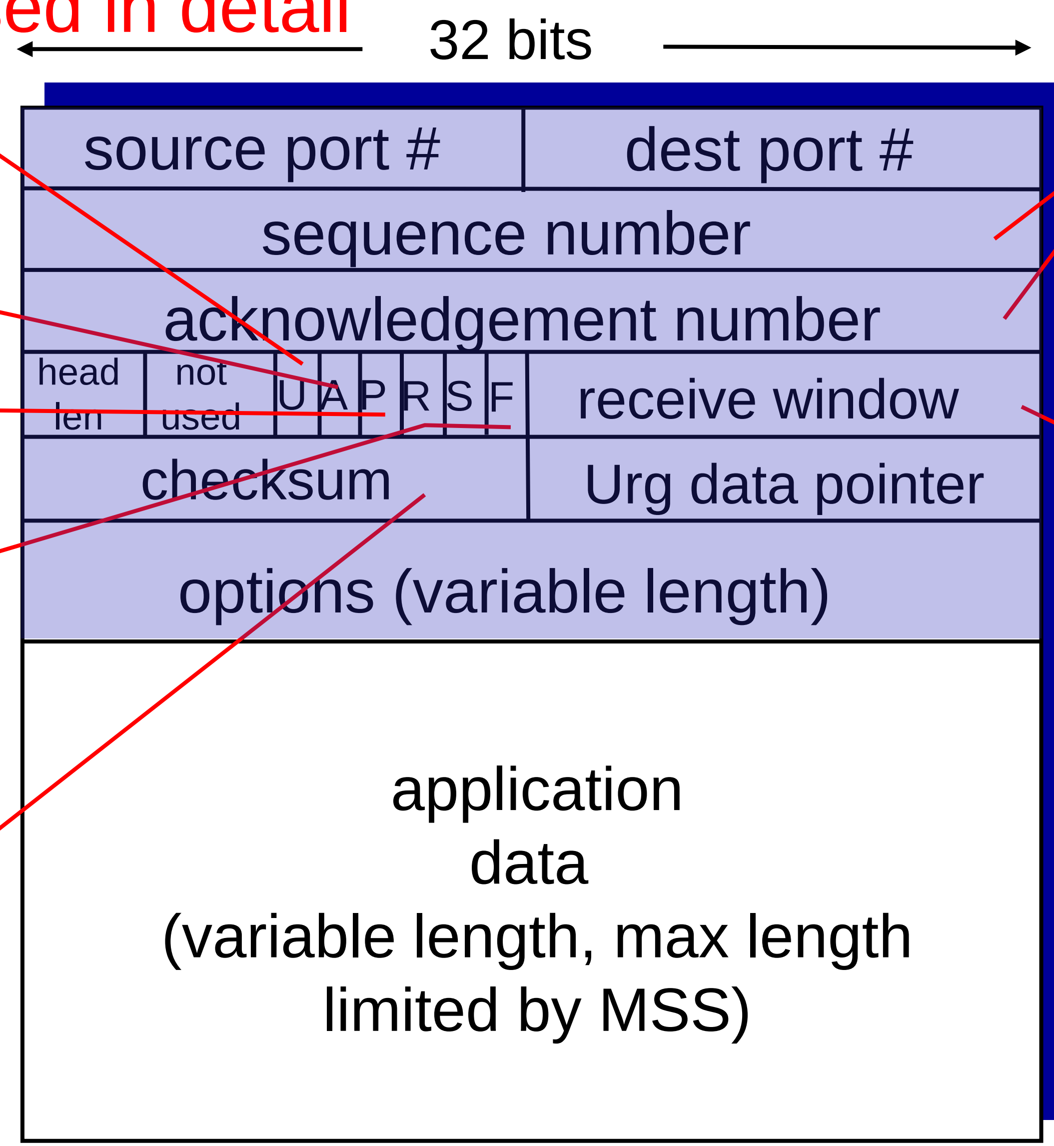
URG: urgent data
(generally not used)

ACK: ACK # valid

PSH: push data now
(generally not used)

RST, SYN, FIN:
connection establishment
(setup, teardown
commands)

Internet checksum
(as in UDP)



counting
by bytes
of data
(not segments!)

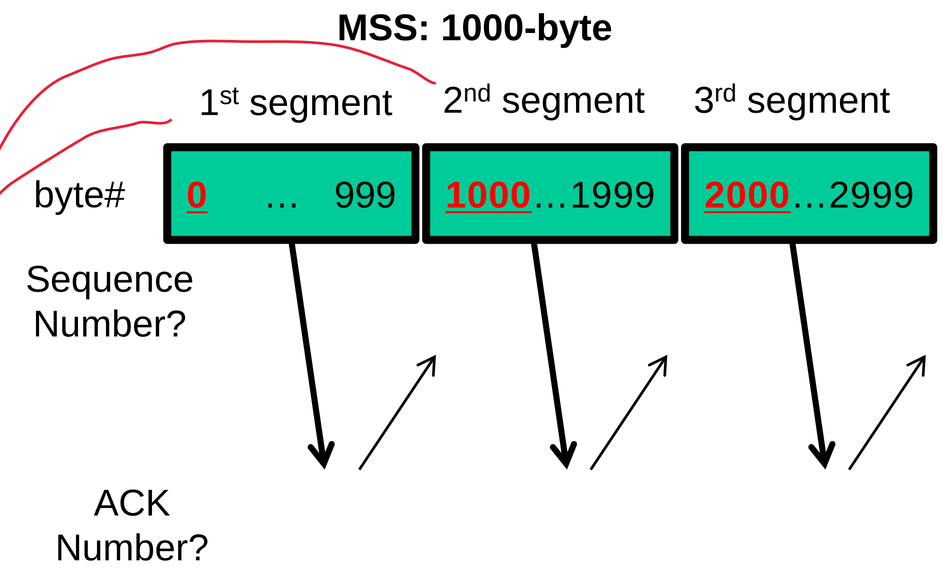
bytes
rcvr willing
to accept

연구자들은
기존 field를 바꿀 순 없으니
option 필드에 넣음.

TCP seq. numbers, ACKs

sequence numbers:

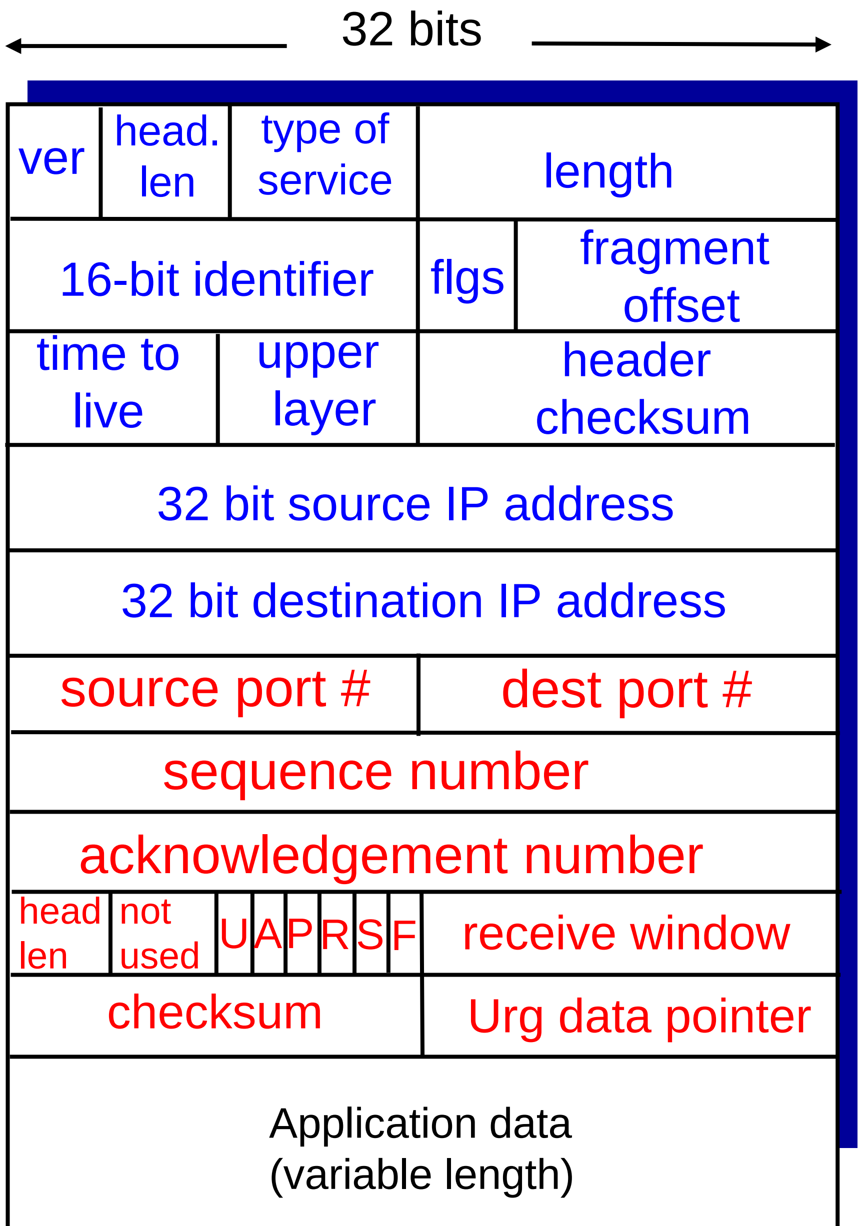
- ***byte stream*** "number" of first byte in segment's data
  sequence number0,
  sequence number1000,
  squence number 2000



acknowledgements:

- seq # of ***next*** byte expected from other side (sender)
  - cumulative ACK

source port #								dest port #
sequence number								
acknowledgement number								
head len	not used	U	A	P	R	S	F	receive window
checksum								Urg data pointer



So!!!

TCP/IP datagram format

application layer가 작으면 overhead?

how much overhead?

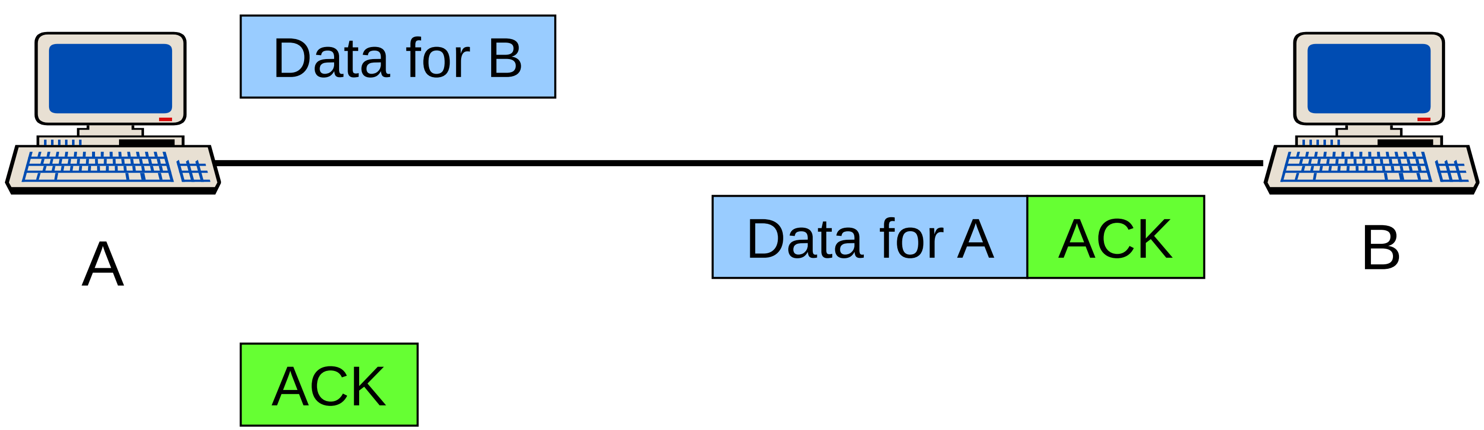
- 20 bytes of TCP
- 20 bytes of IP
- = 40 bytes + app layer overhead

ack이 많이 왔다가야 하니까 너무 많다.

Acknowledgements

- TCP receivers use acknowledgments (ACKs) to confirm the receipt of data to the sender
  - ACK information is included in the TCP header
- full duplex data: bi-directional data flow in same connection
  - Acknowledgment can be added ("piggybacked") to a data segment that carries data in the opposite direction
- Acknowledgements are very important!
  - also used for flow control, congestion control

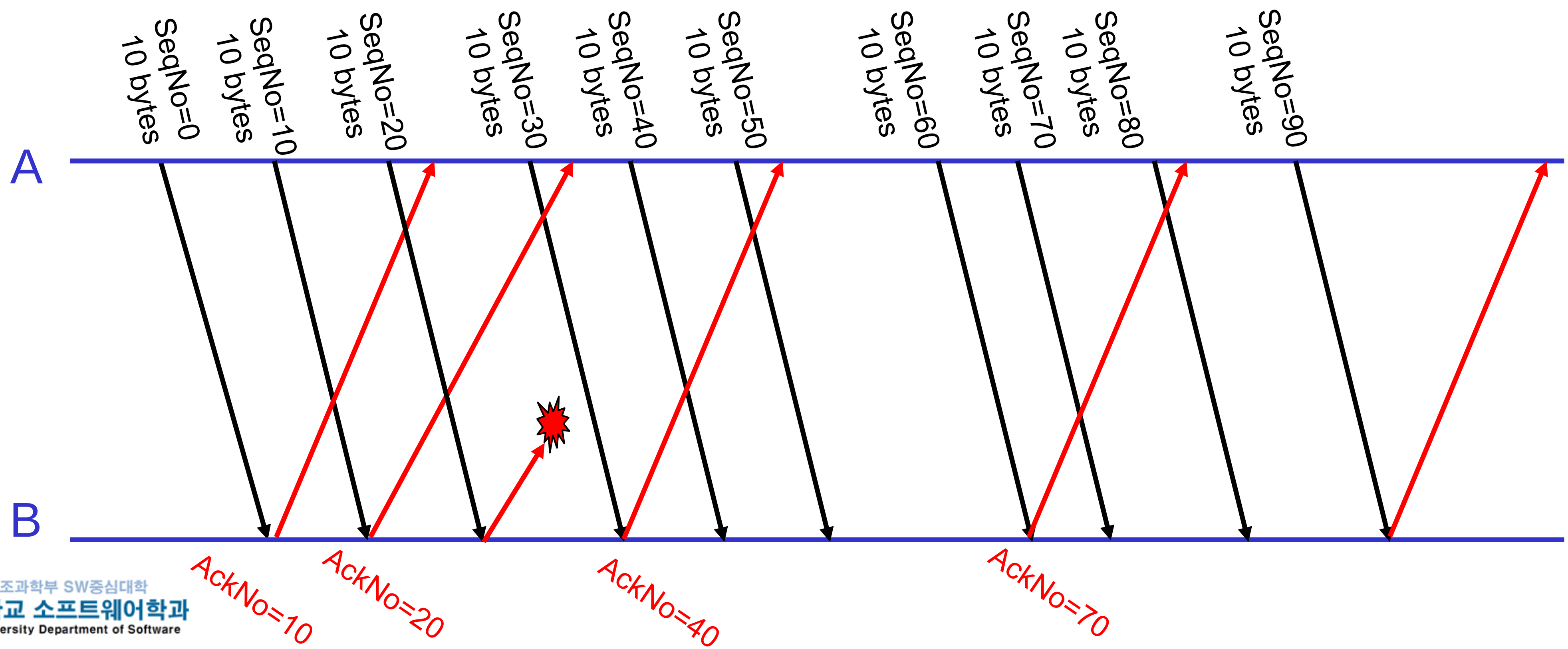
ack을 보내야 하는데 마침 b가 또 보내야 하는 data가 있어서 data for A에 실어서 보낸다.
만약 없다면 ack만 보낸다.



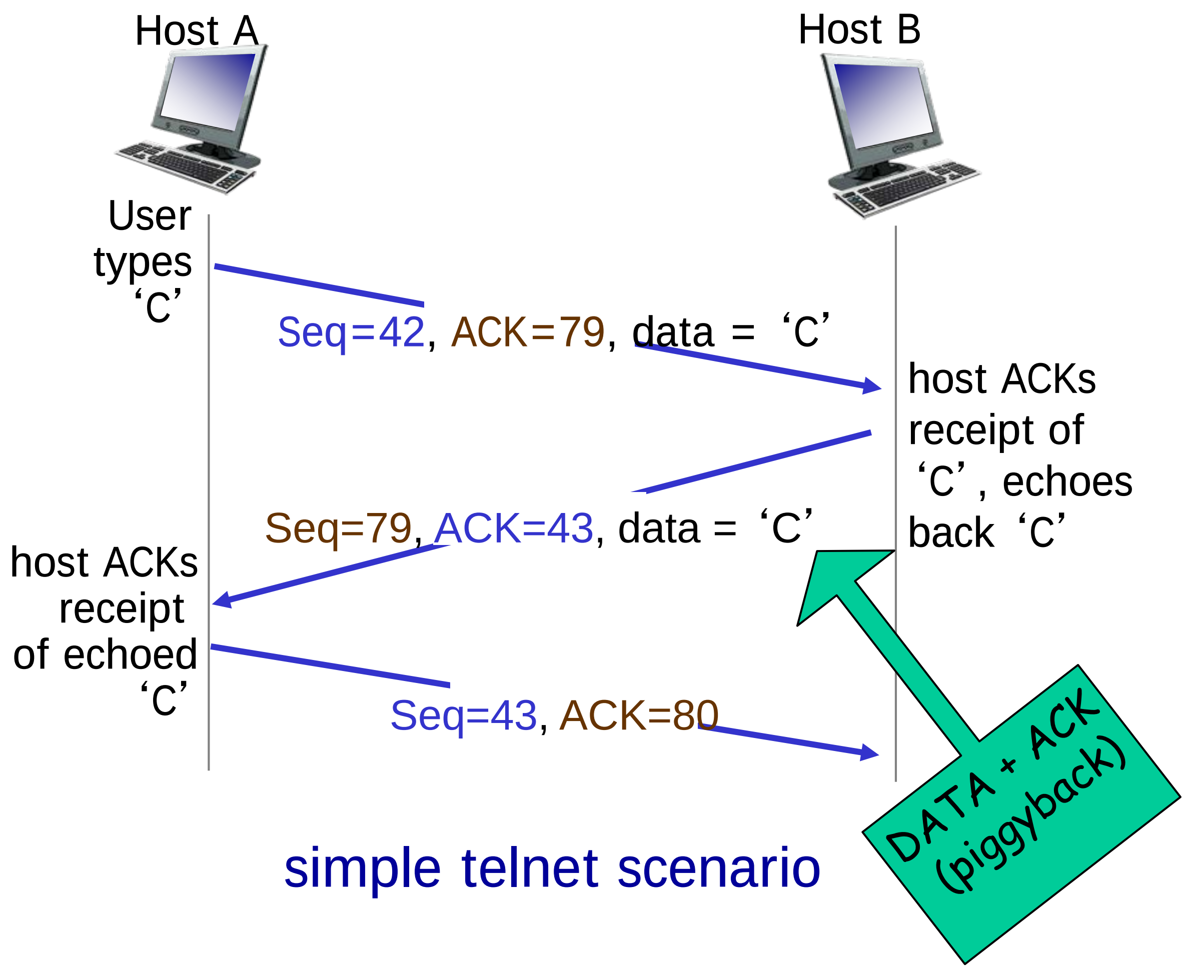
ack이 로스 나도 ackNo 40에 문제가 없다.
중간 ack loss를 커버할 수 있다.

Cumulative ACK

- An acknowledgment confirms receipt for all unacknowledged data that has a smaller sequence number than given in the ACK# field
  - Example: ACK#=40 confirms correct delivery until seq# 39.

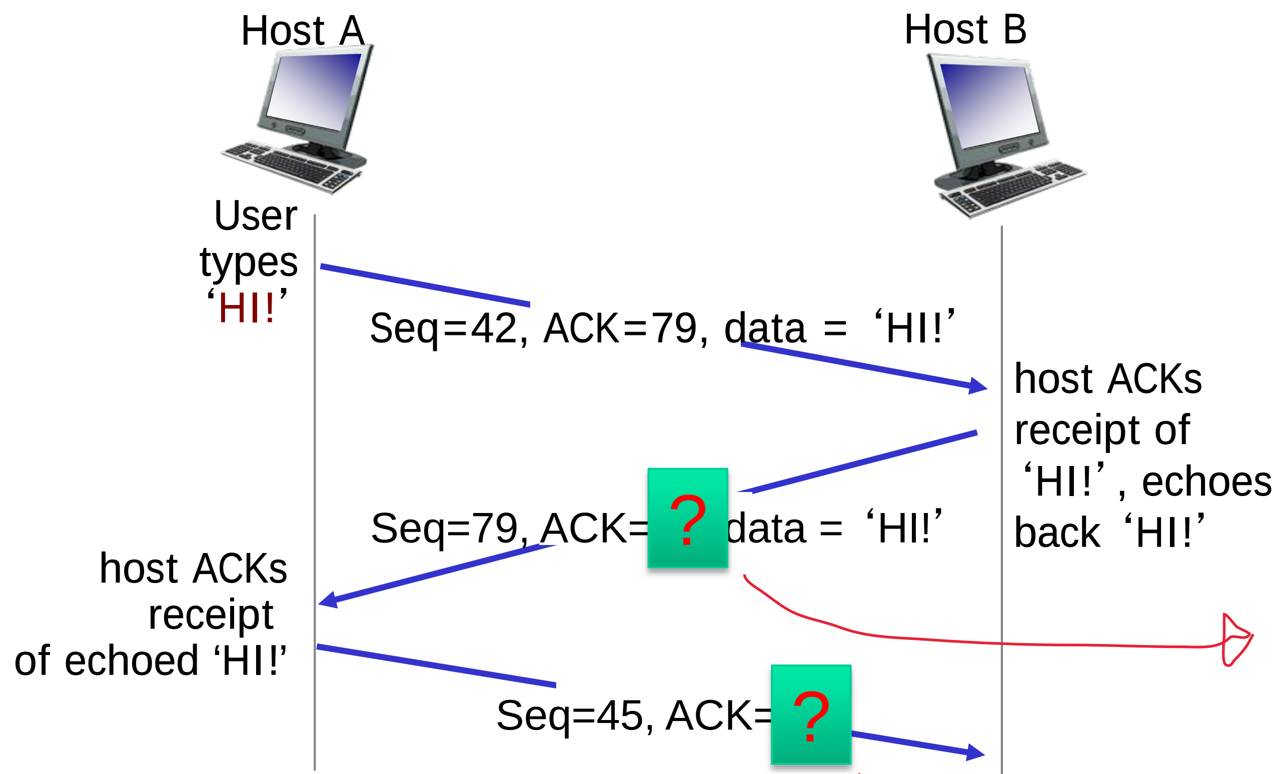


TCP seq. numbers, ACKs (1-byte data)



simple telnet scenario

TCP seq. numbers, ACKs (3-byte data)



45
3바이트니 42, 43, 44이다.
받기를 원하는 것은 45가 됨.

simple telnet scenario

ack = 81
because 원하는 넘버가 되기 때문에
79, 80, 81

Ch. 3.5.3 TCP RTT Estimation and Timeout

Q: how to set TCP timeout value? (i.e., How long should sender wait for ACK?)

- ① *too short:* premature timeout, unnecessary retransmissions
- ② *too long:* slow reaction to segment loss
- To prevent unnecessary timeouts: larger than RTT
  - but RTT varies

Q: how to estimate RTT?

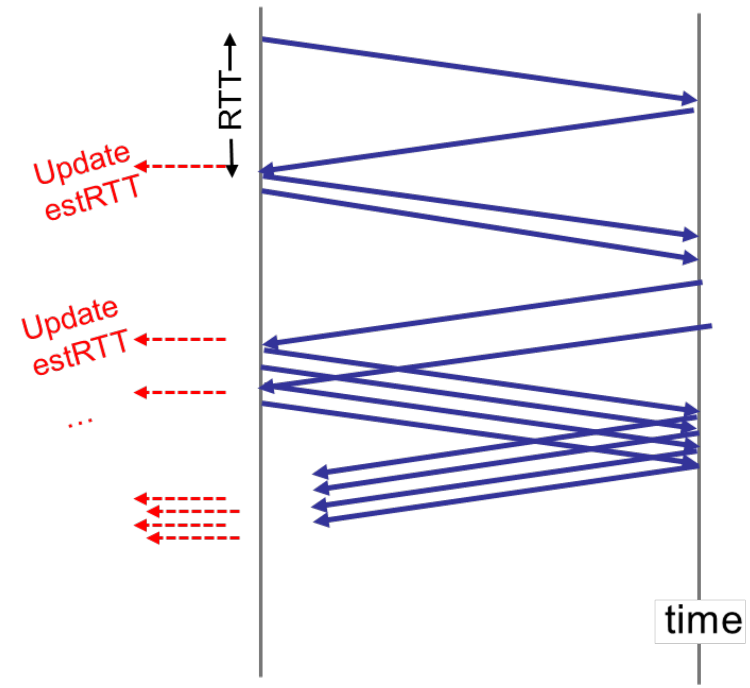
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "**smoother**"
  - average several *recent* measurements, not just current **`SampleRTT`**

Estimated RTT

Q: how to estimate RTT?

재전송 된것은 안 잰다.

- **SampleRTT**: measured time from segment transmission until **ACK receipt**
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "**smoother**"
  - average several *recent* measurements, not just current **SampleRTT**
- Average: **Estimated RTT**



$$\texttt{EstimatedRTT(n) = (1-}\ \alpha\texttt{)*EstimatedRTT(n-1) +}\ \alpha\texttt{*SampleRTT(n)}$$

재귀방식

과거의 값은 1-a만큼 반영

Last averaged RTT

This instantaneous RTT

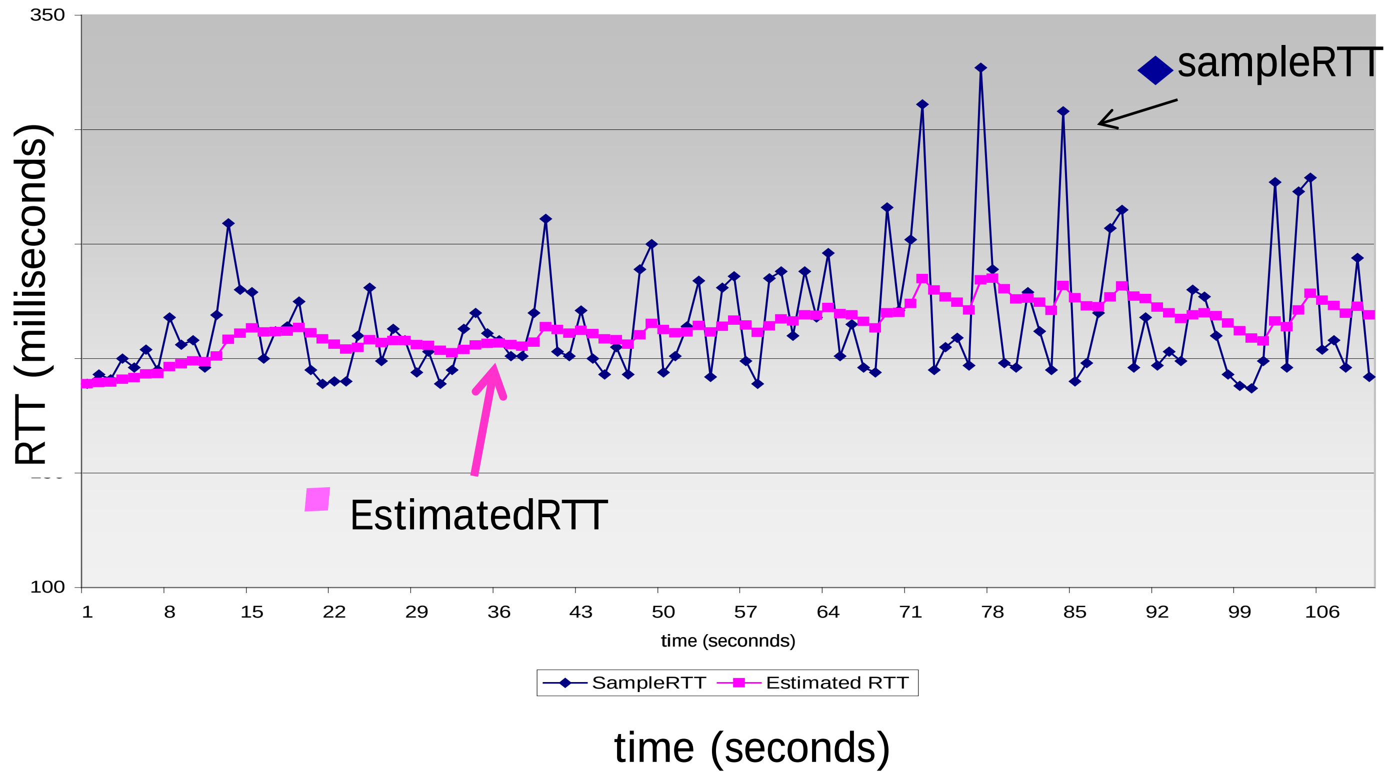
Estimated RTT

EstimatedRTT(n) = (1- α)*EstimatedRTT(n-1) + α*SampleRTT(n)

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125

보통

EstimatedRTT = 0.875 * EstimatedRTT + 0.125 * SampleRTT



RTT deviation

현재 추세를 잘 반영하는 구간도 있지만 그렇지 않은 구간도 있다.

- DevRTT: RTT variation (or deviation).
- estimate SampleRTT deviation from EstimatedRTT:

deviation

Last averaged DevRTT

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|
```

This instantaneous DevRTT

deviation

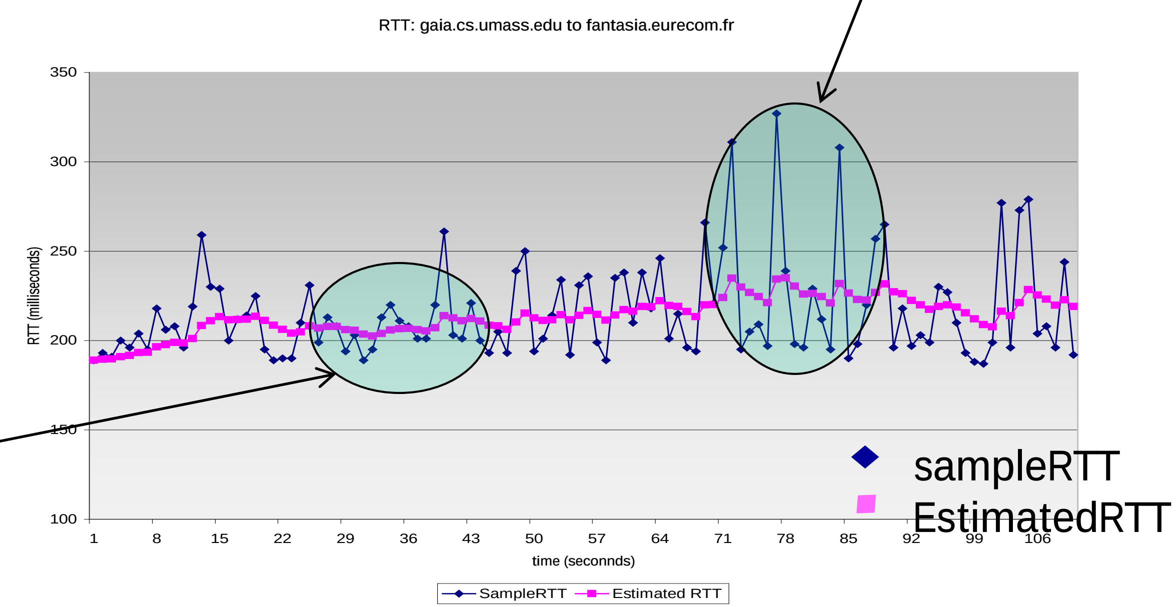
(typically, β = 0.25)

10ms, 20ms, 30ms
20ms에서 측정했을 때
0.875*10ms + 0.125*20ms

30ms
0.875 * estimatedRTT(20) + 0.125*30ms

high DevRTT

low DevRTT



TCP timeout

timeout interval: **`EstimatedRTT`** plus "safety margin"

large variation in **`EstimatedRTT ->`** larger safety margin

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

estimated RTT

"safety margin"

현재 추세를 잘 반영하는 구간도 있지만
그렇지 않은 구간을 반영하고 싶기 위해

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP reliable data transfer

- TCP creates **rdt** service on top of IP's unreliable service
- single retransmission timer
  - Conceptually, individual timer needed for each transmitted unacked packet
  - Timer management require considerable overhead
  - So TCP uses only **single retransmission timer**

let's initially consider ***simplified*** TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- create segment with seq #
  - seq # (Ch. 3.4.2) is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`** (Ch. 3.5.3)

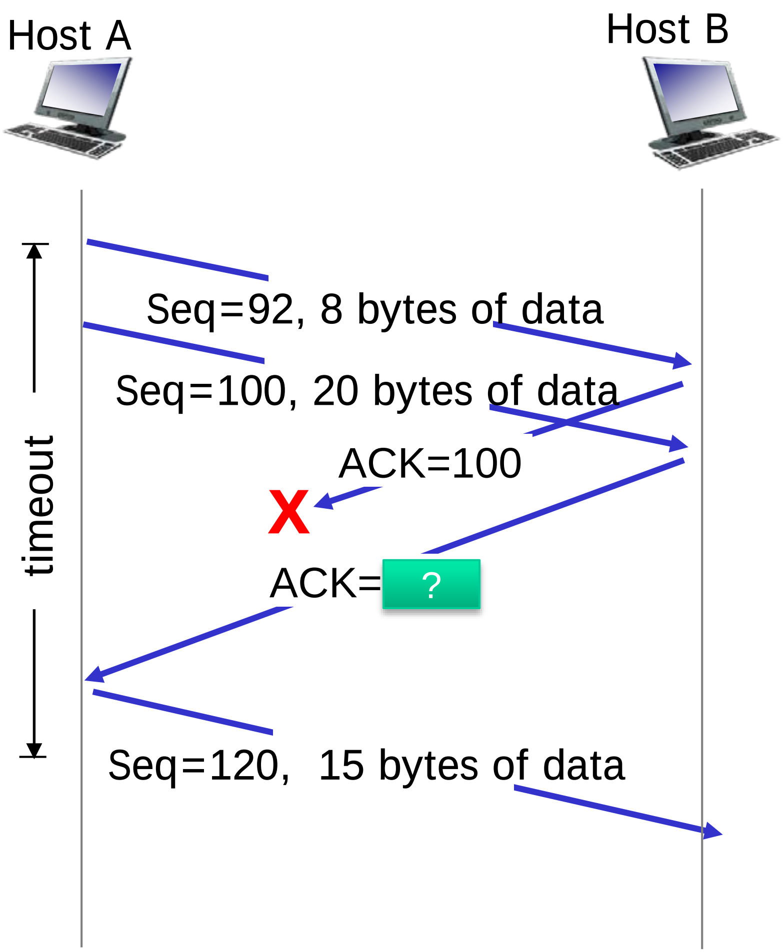
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

TCP: retransmission scenarios



cumulative ACK

Multiple consecutive timeouts

- Length of timeout?
  - `EstimatedRTT(n) = (1- α)*EstimatedRTT(n-1) + α*SampleRTT(n)`
  - `TimeoutInterval = EstimatedRTT + 4*DevRTT`
- Then what happens if another timeout occurs?
  - Option1: Again, use the above **`TimeoutInterval`**
  - Option2: Take a more conservative approach – **double** the timeout value
    - E.g., current timeout = 0.75s, another timeout occurs, then next timeout = 0.75 x 2 = 1.5s, next timeout = 1.5 x 2 = 3.0s, …
- TCP takes option2. why?
  - Consecutive timeouts – means network condition is good/bad?
  - TCP should be polite – retransmit after shorter/longer intervals

TCP ACK generation: Delayed ACK

- TCP delays transmission of ACKs for up to 500ms
  - Why?
- Avoid to send ACK packets that do not carry data.
  - The hope is that, within the delay, the receiver will have data ready to be sent to the receiver. Then, the ACK can be piggybacked with a data segment (reduce ACK-only response)
- Exceptions
  - ACK should be sent for every second (두개의) segment receptions
  - Delayed ACK is not used when packets arrive out of order

TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet (initially around 200-300ms)
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

3 dup = NAK

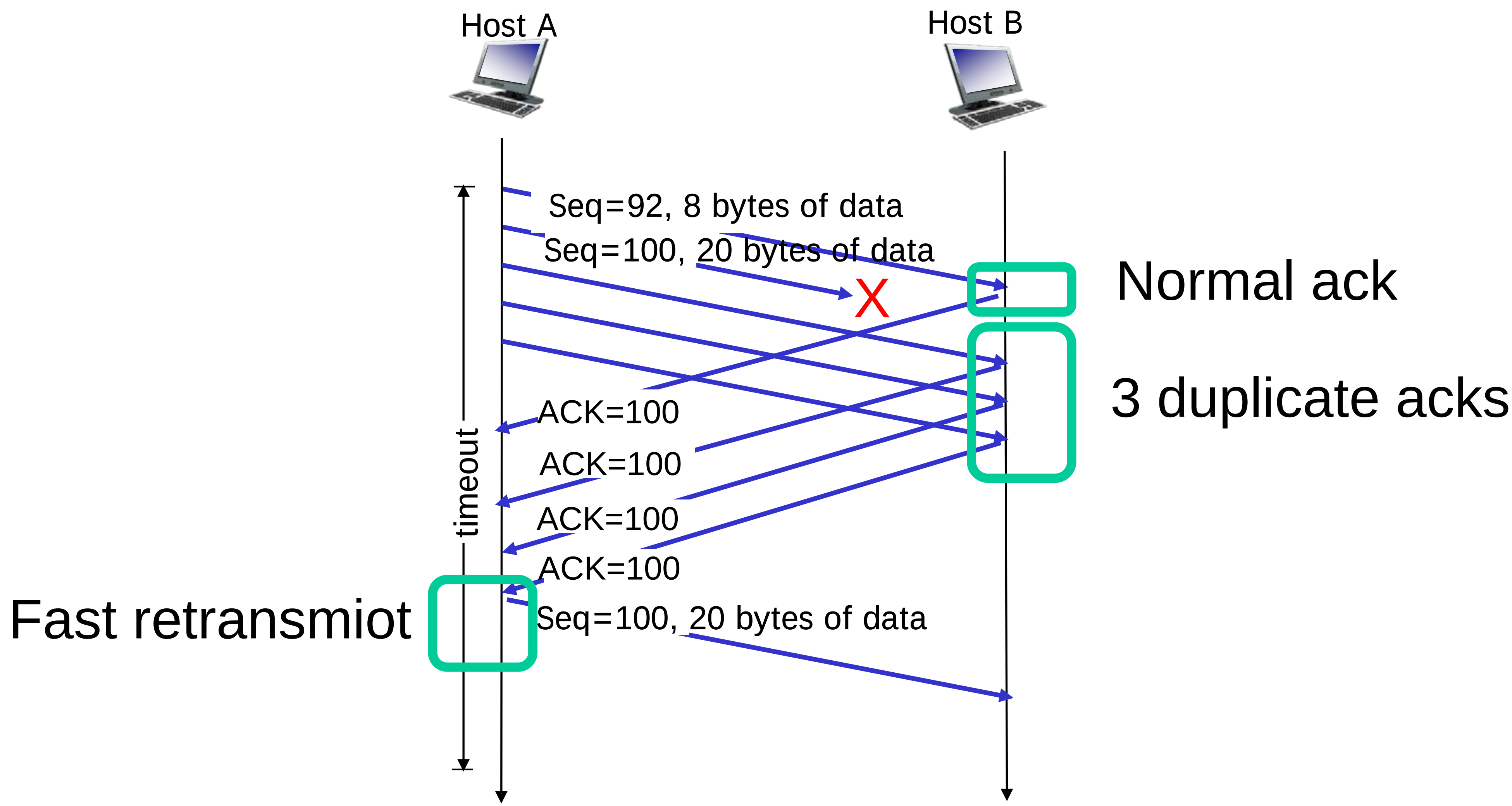
TCP fast retransmit

if sender receives **three** ACKs for same data

("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender
receipt of triple duplicate ACK

Is TCP rdt GBN or Selective Repeat?

- GBN?
  - TCP ACKs are cumulative
  - correctly received but out-of-order segments are not individually ACKed by receiver
  - consequently, TCP sender only maintain smallest sequence number of transmitted but unACKed byte (**`SendBase`**) and sequence number of next byte to be sent (**`NextSeqNum`**)
- SR?
  - Buffer correctly received by out-of-order segments
  - If segment n gets lost, TCP only retransmits segment n (GBN retransmitted n, n+1, n+2, …)
- Conclusion: TCP is a hybrid of GBN and SR

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Flow Control

1 Gbps

1 Gbps
Network

10Mbps

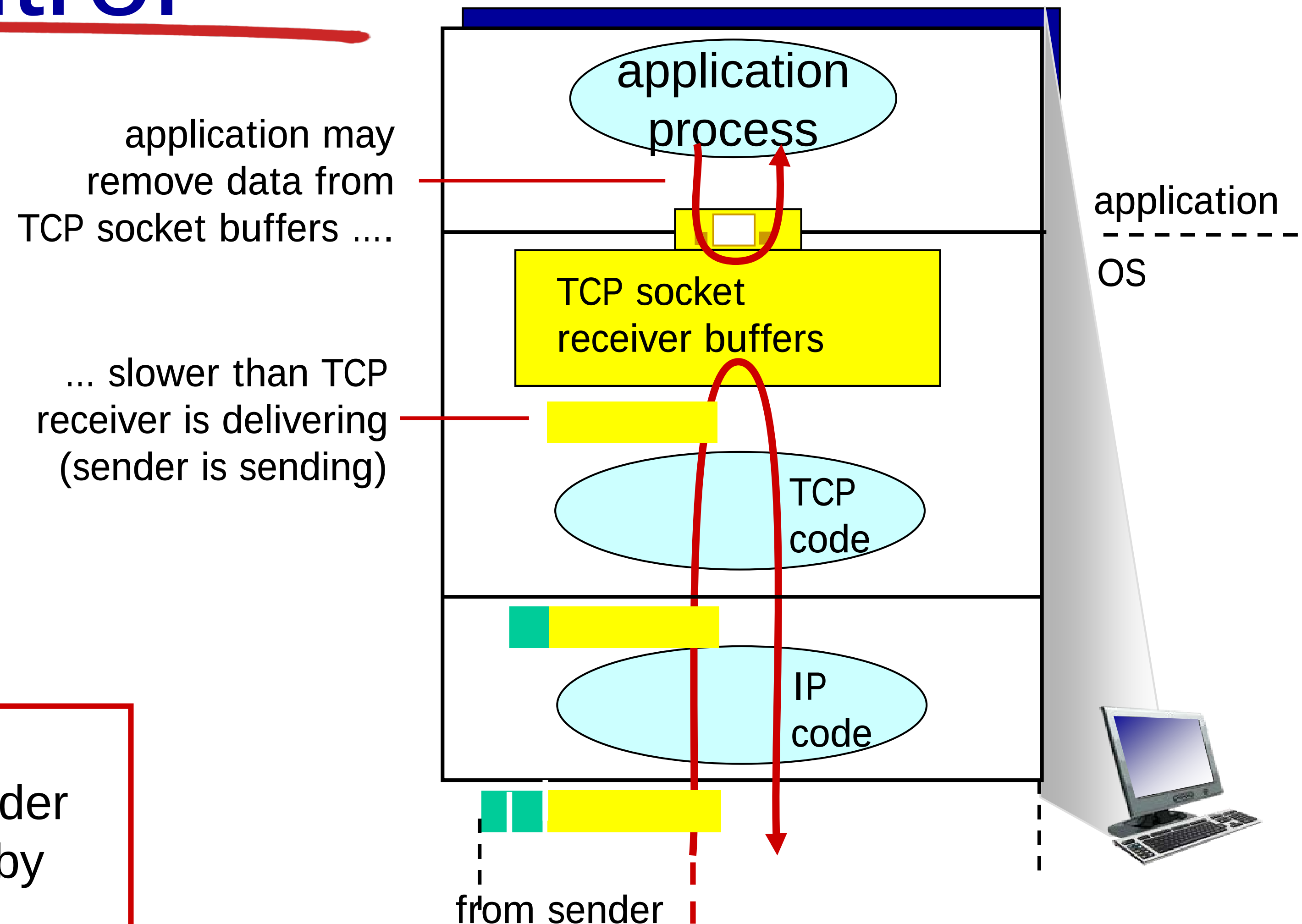
TCP Send
buffer (100MB)

TCP Receive
buffer (1MB)

What will happen if Sender keeps sending at 1Gbps? – buffer overflow at receiver

TCP flow control

source port #		dest port #	
sequence number			
acknowledgement number			
head len	not used	U A P R S F	receive window
checksum		Urg data pointer	



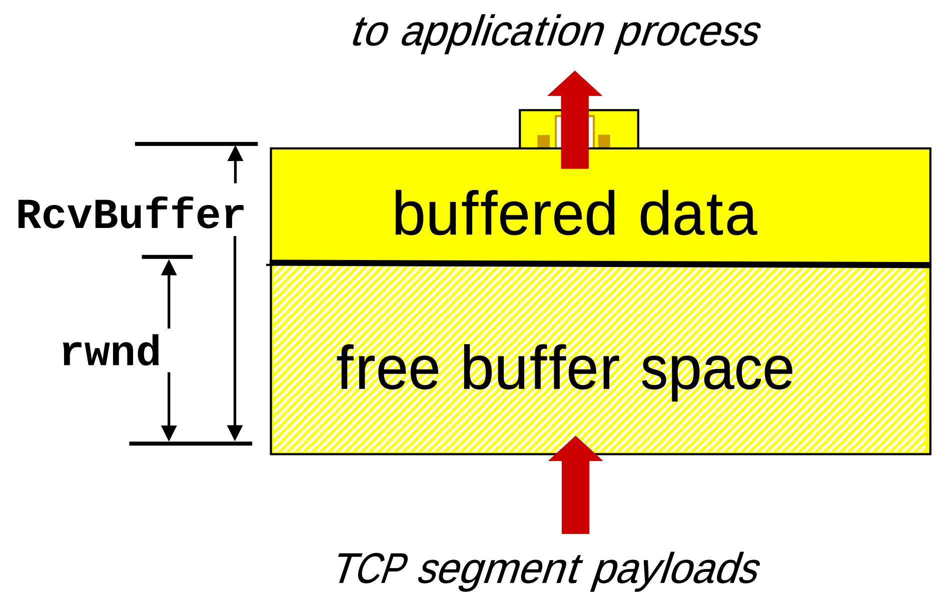
receiver protocol stack

flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow



receiver-side buffering

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

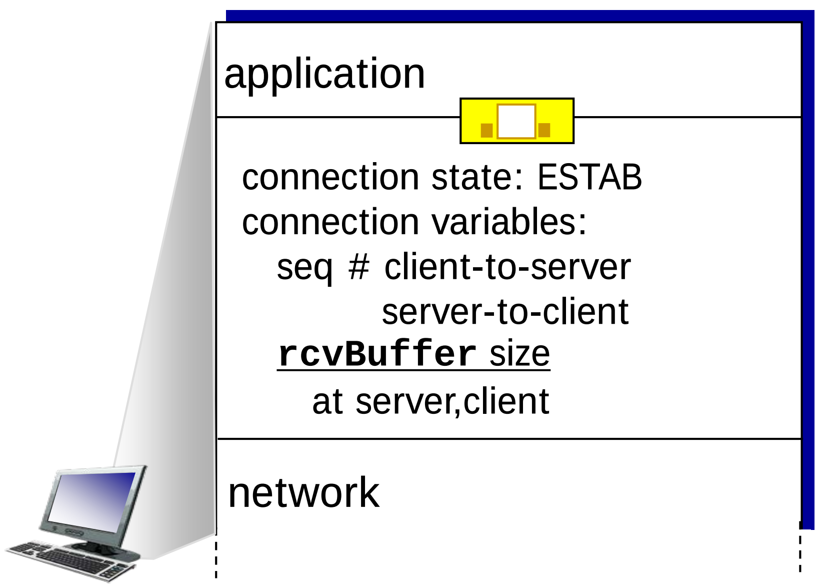
3.6 principles of congestion control

3.7 TCP congestion control

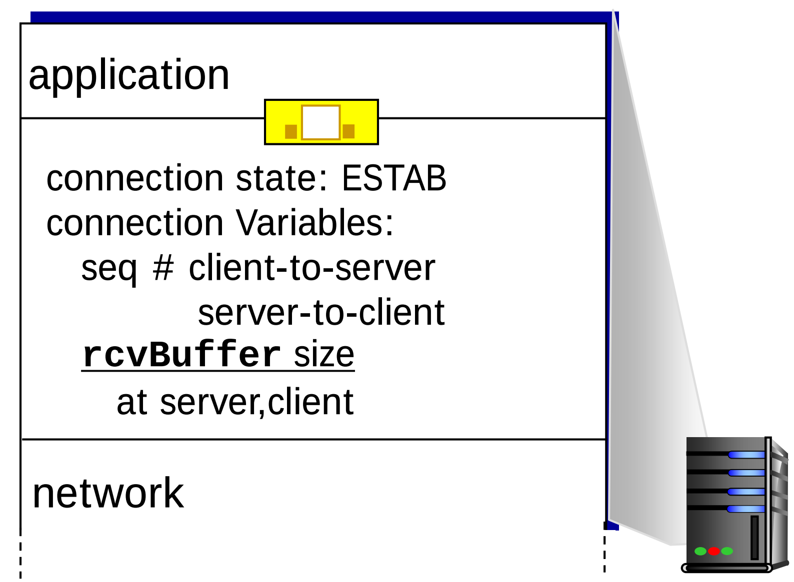
Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

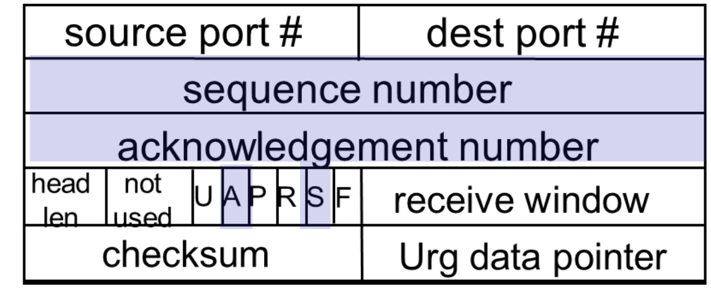


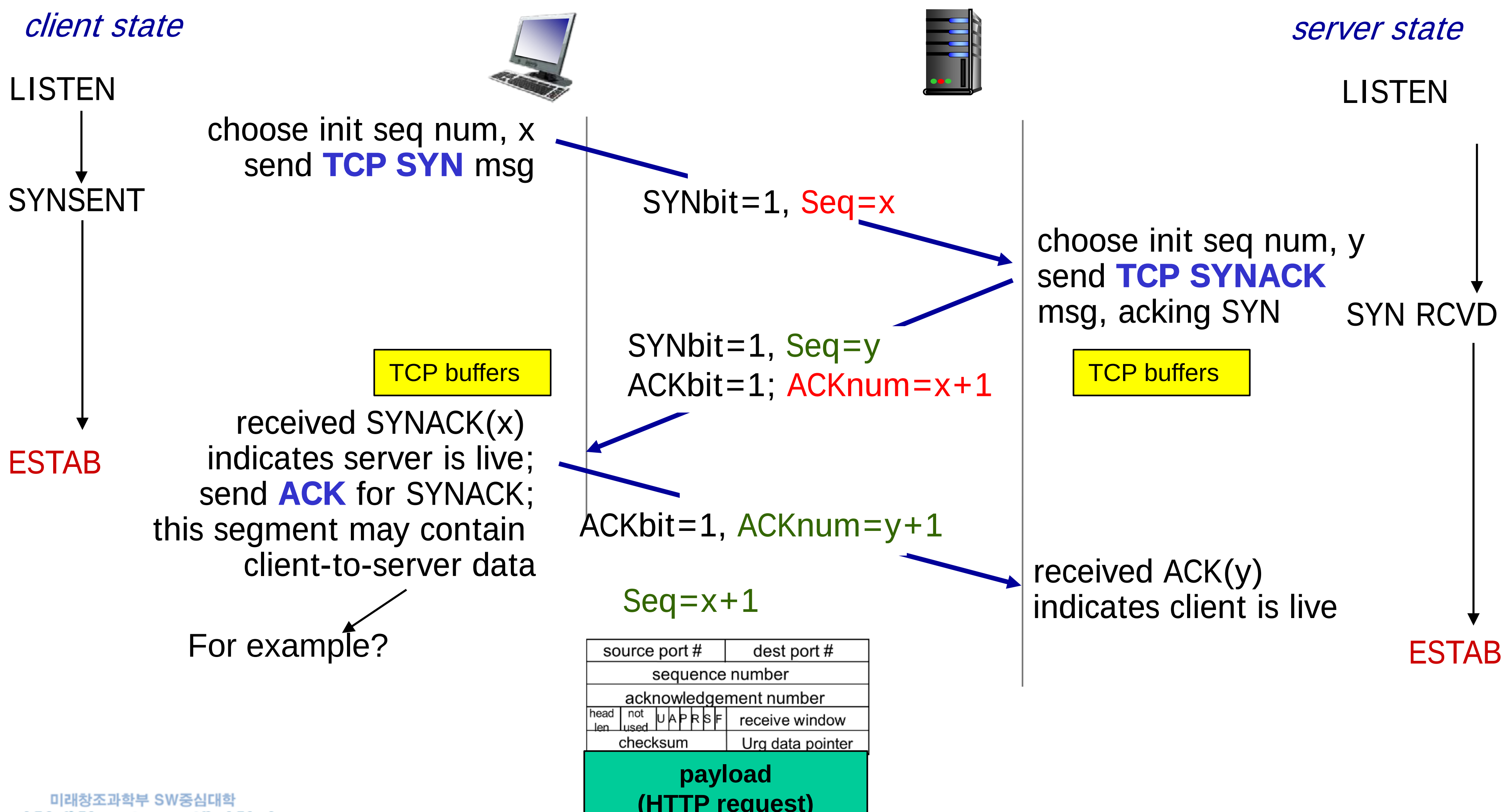

```
Socket clientSocket =
  newSocket("hostname","port number");
```




```
Socket connectionSocket =
  welcomeSocket.accept();
```

TCP 3-way handshake





TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection