Transport Layer: TCP 2

Oct 15, 2024

Min Suk Kang
Associate Professor
School of Computing/Graduate School of Information Security

NetS&P Research Lab @ KAIST

KAIST

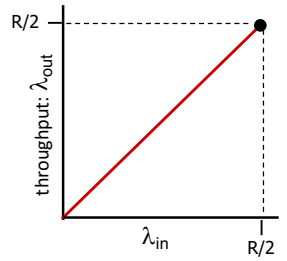
Chapter 3: roadmap

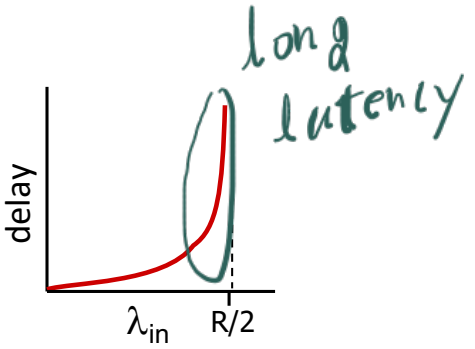
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

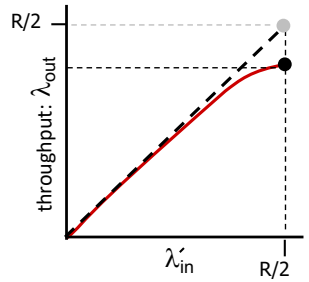
Causes/costs of congestion: insights

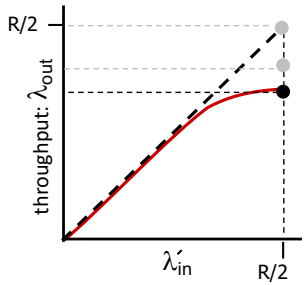
= priciple of congestion

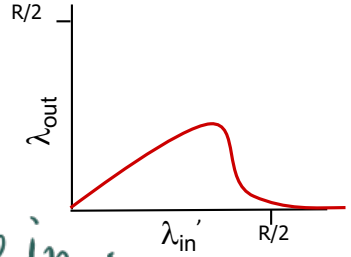
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
  - by timeout
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream
  - ↳ wasted all sources in upstream







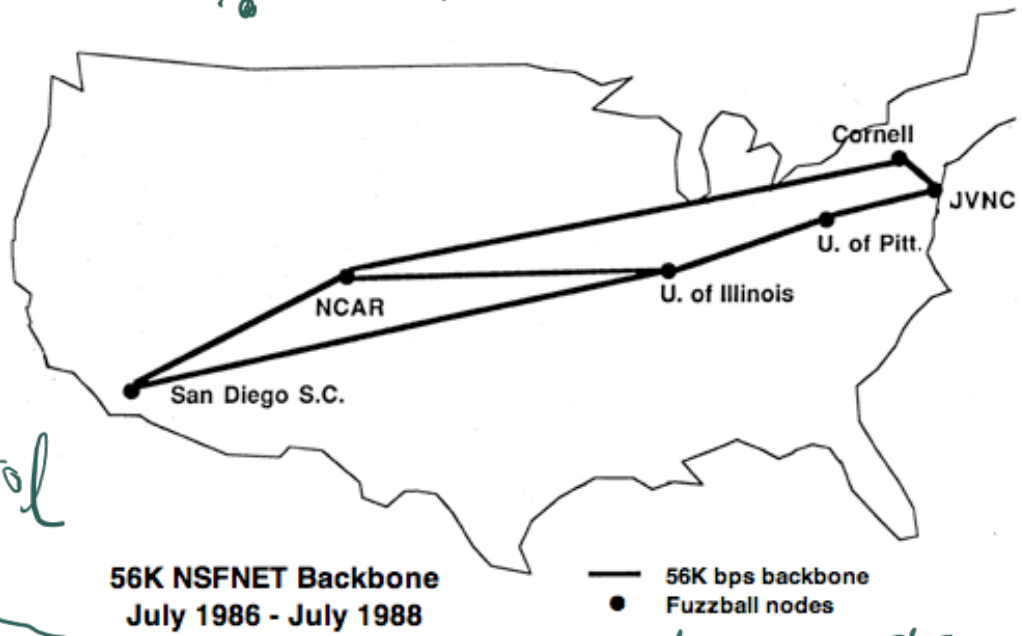




“Congestion Collapse”

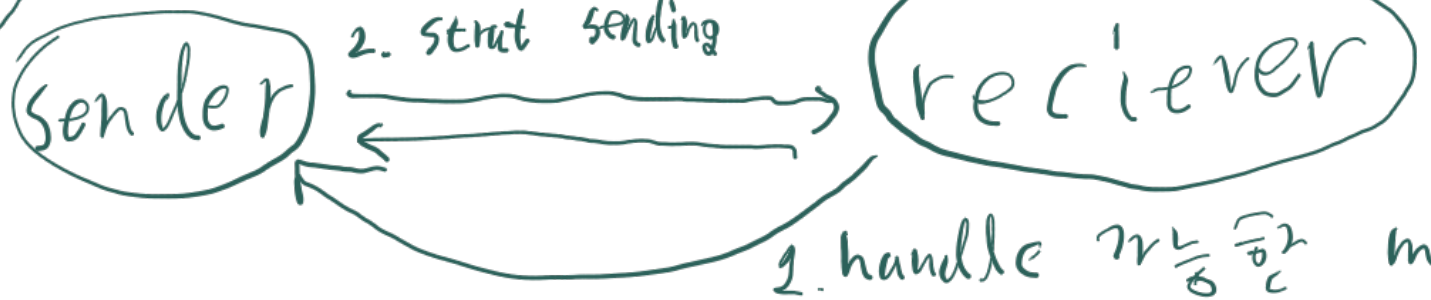
- In October 1986, NSFNET backbone capacity dropped from 32 kbps to 40 bps
- TCP congestion control invented between 1987-1988 by Van Jacobson and Sally Floyd

32 k bps → 40 bps



처음엔 no congestion control

window

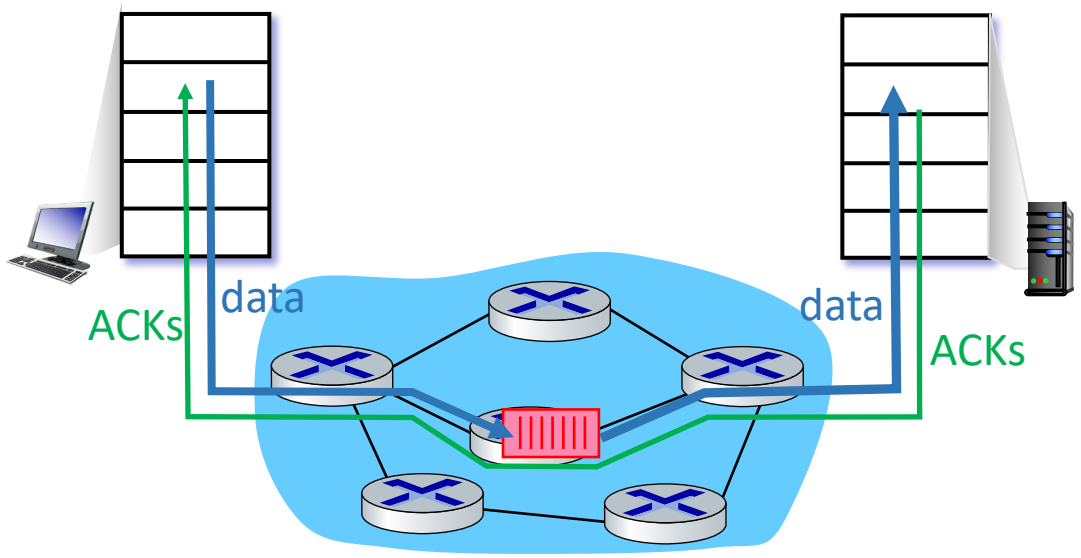


→ 그러면 sender의 입장은 congestion 발생가능 (loss / retransmit)

Approaches towards congestion control

End-end congestion control:

- no explicit feedback from network
- congestion *inferred* from observed loss, delay
  - approach taken by TCP



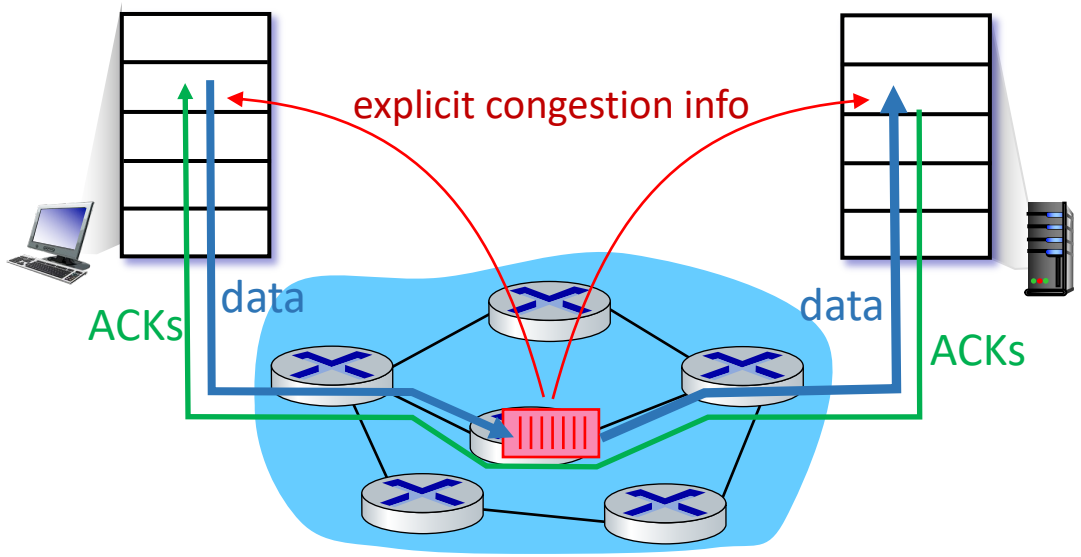
loss, delay ↑ ⇒ congestion ↑

Approaches towards congestion control

TCP expension 중 일부 해당

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



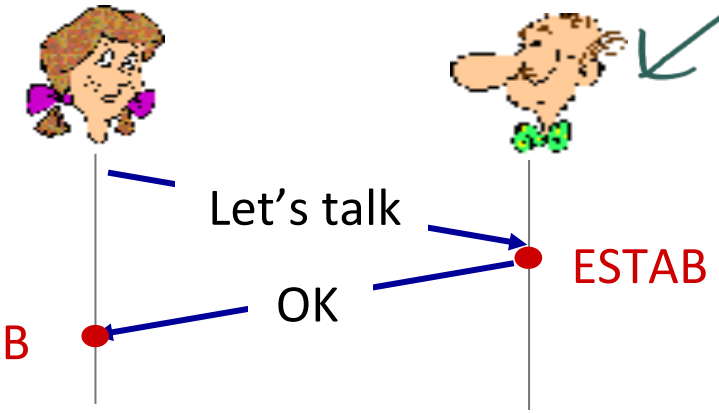
router가 congestion 정보를 직접 전달

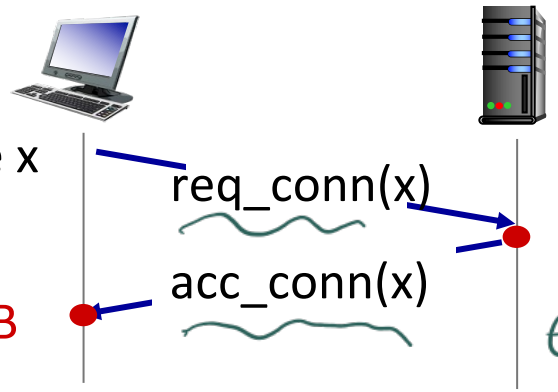
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

Agreeing to establish a connection

2-way handshake:





Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

TCP 3-way handshake

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName,serverPort))
```

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

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

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

ACKbit=1, ACKnum=y+1

ESTAB

Server state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

LISTEN

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYN RCVD

received ACK(y)
indicates client is live

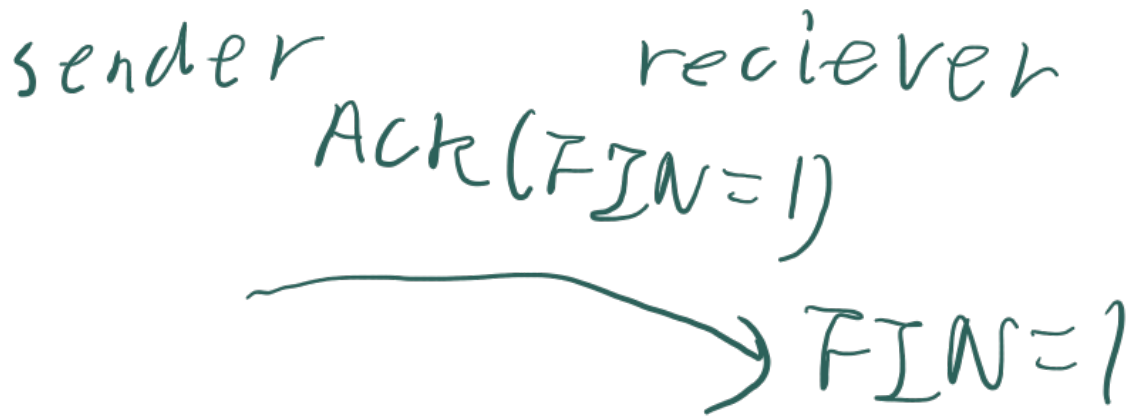
ESTAB

A human 3-way handshake protocol

Closing a TCP 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

동시에



Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Thought experiment: subway chair

n개의 좌석

사람이 없을

때는 n개를

전부 차지해도

OK

문제는 다른

사람과 resource를

공유해야 한다.

↓

극단적 상황을 예로

들어서, TCP는 자원

낭비도 하나만 차지

더 효율 높일 방법?

일단 빼고, 복잡하면 짱!

TCP congestion control: AIMD

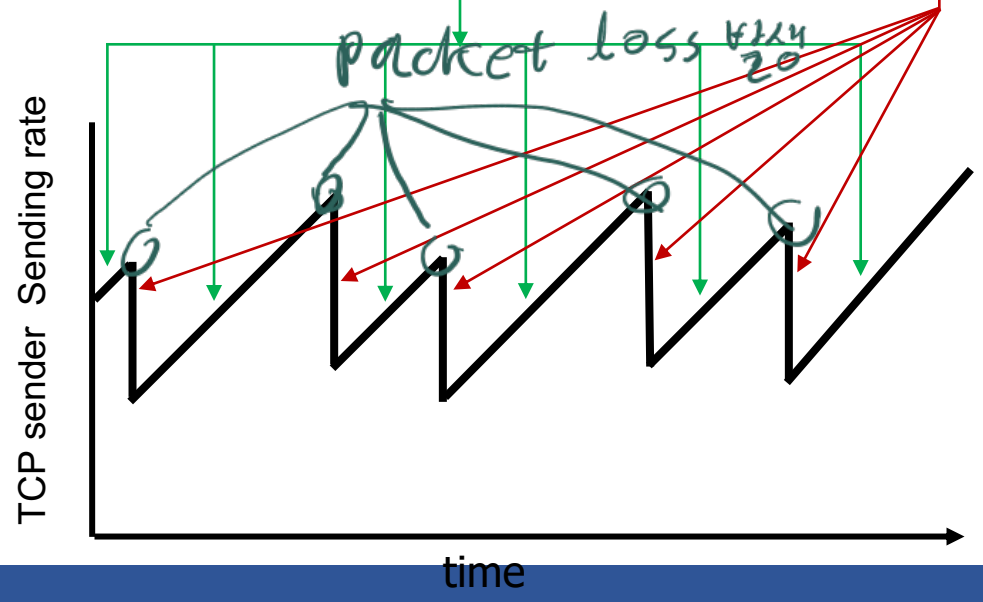
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

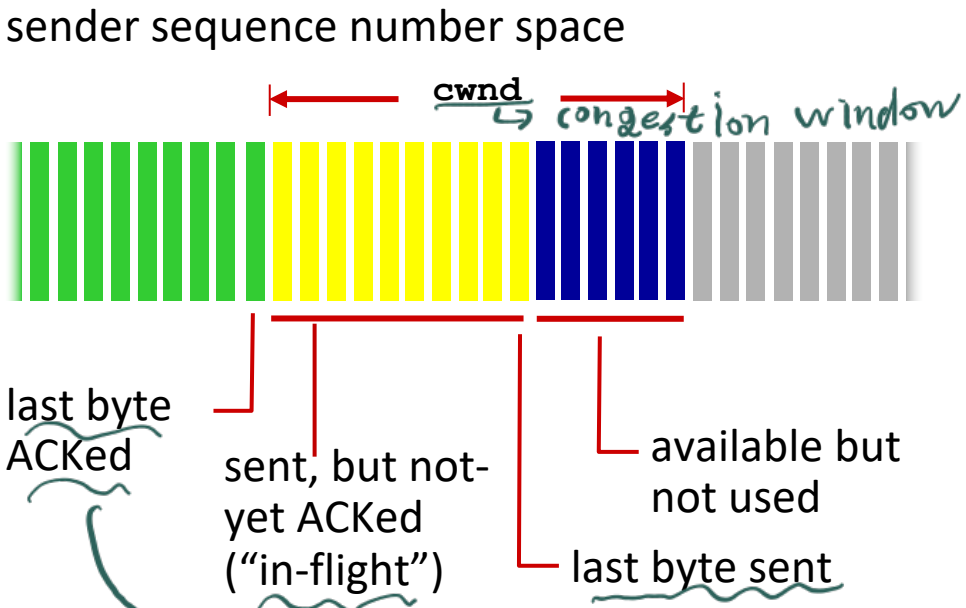
Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

TCP congestion control: details



TCP sending behavior:

- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

(cwnd: 1회 전송량, RTT: 전송&반응 시간)

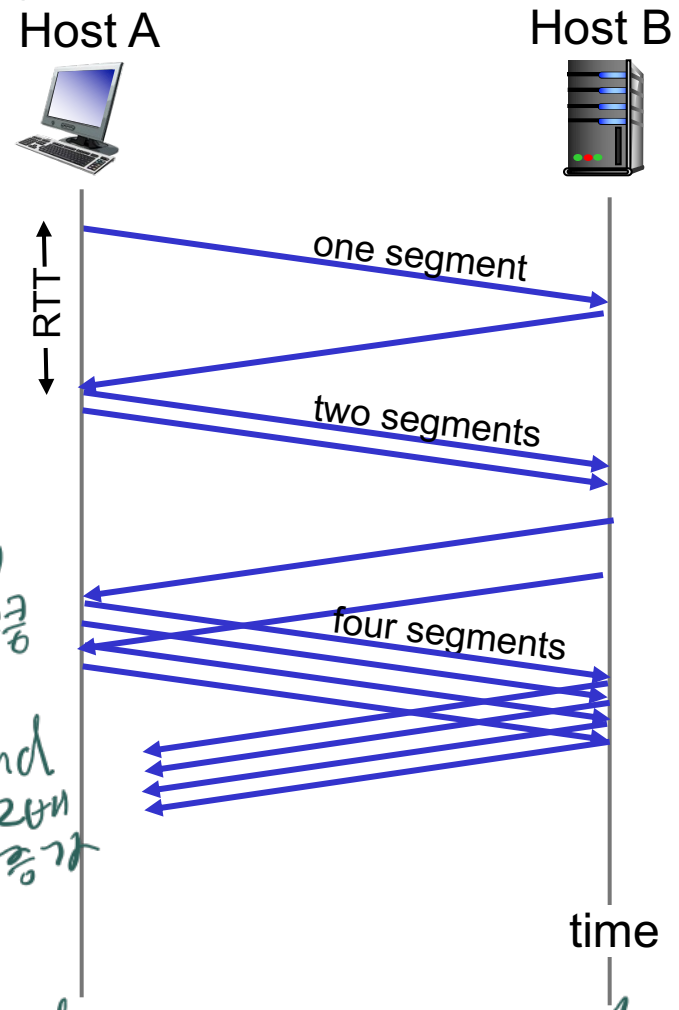
조건: in-flight ≤ cwnd

- TCP sender limits transmission: `LastByteSent- LastByteAcked ≤ cwnd`
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

↳ cwnd를 congestion 유무에 따라 동적 조정 { 1. slow start, 2. congestion avoidance, 3. fast recovery

TCP "slow" start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



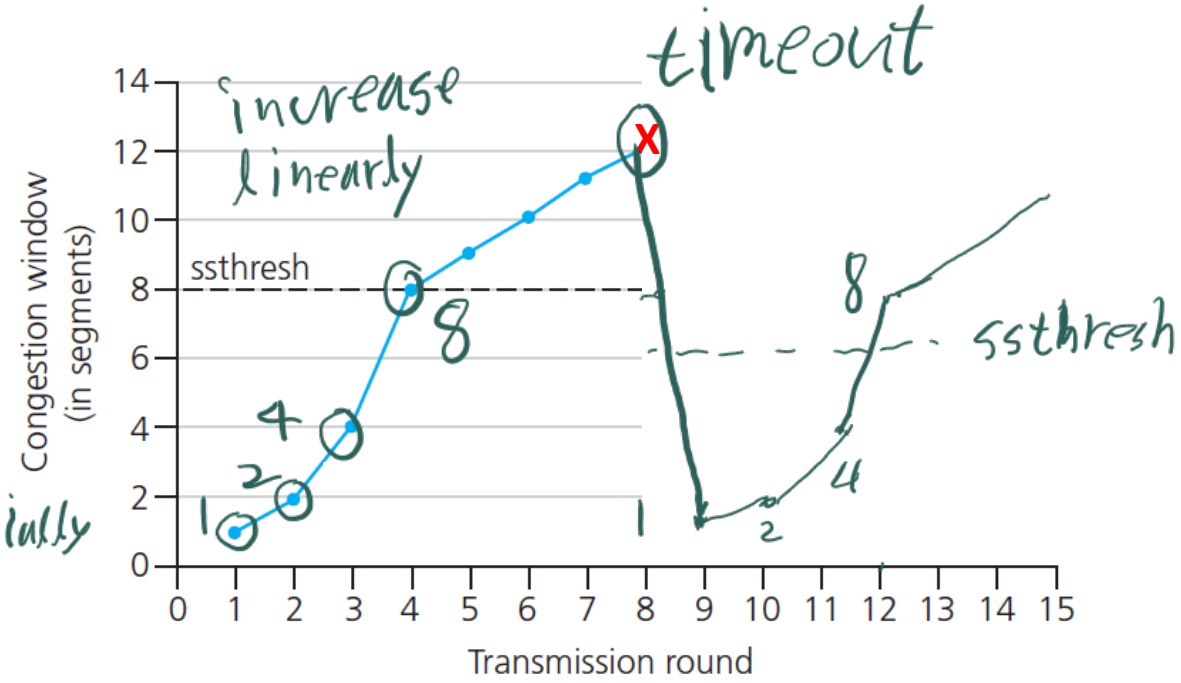
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.

Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



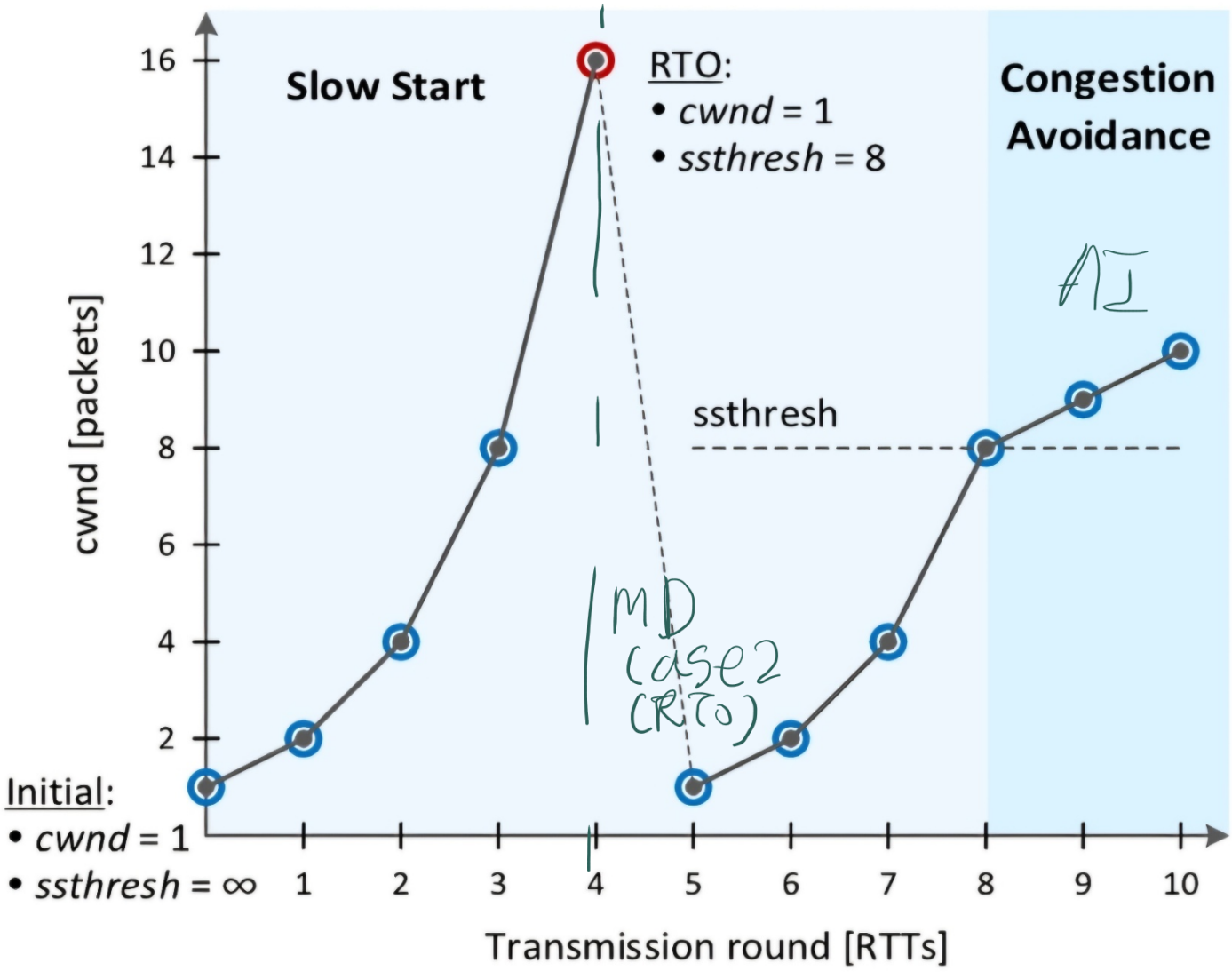
increase exponentially

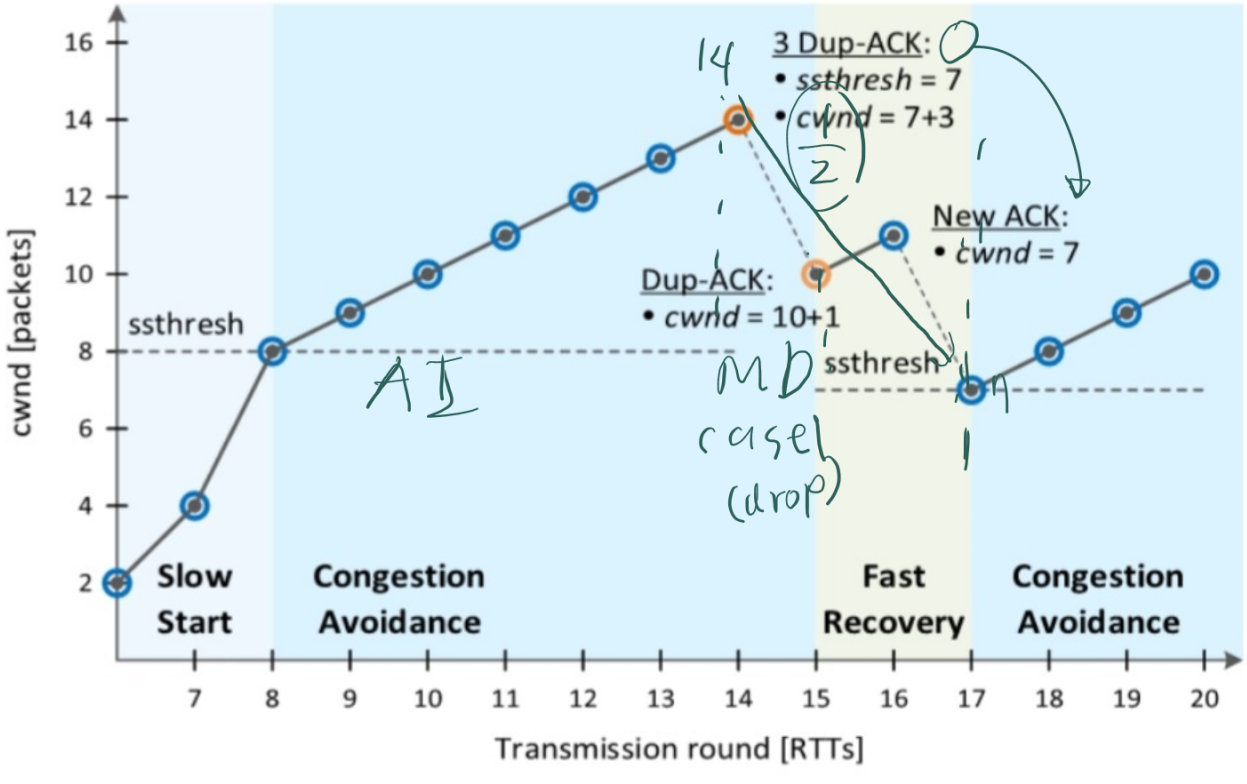
packet drop 발생을 인함

RTO(retransmission timeout)

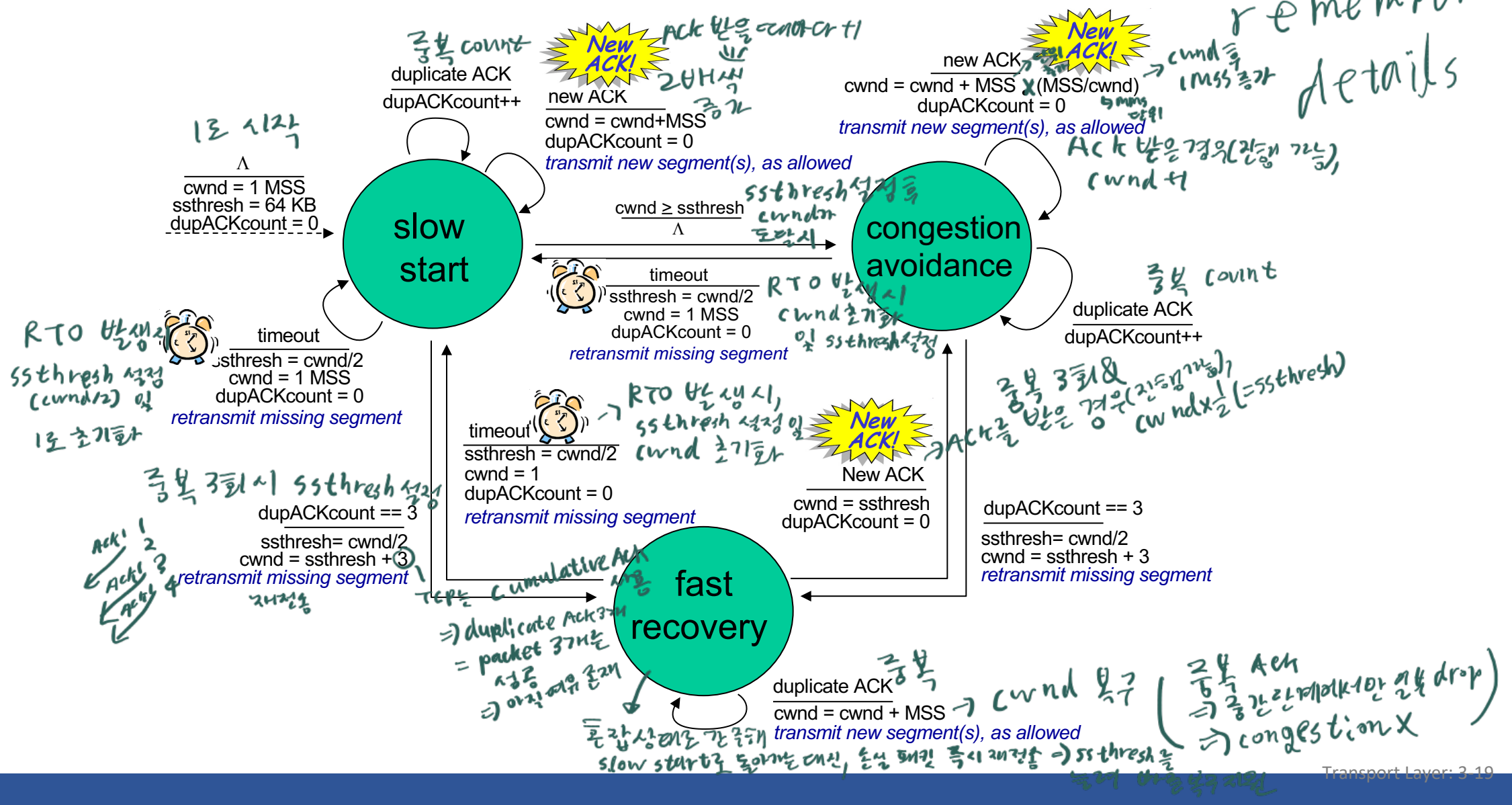
직전값의 1/2로 ssthresh 설정, cwnd 1로 초기화

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/


Slow Start
Congestion Avoidance
RTO:
• cwnd = 1
• ssthresh = 8
ssthresh
AI
MD
case2
(RTO)
Initial:
• cwnd = 1
• ssthresh = ∞
cwnd [packets]
Transmission round [RTTs]
16
14
12
10
8
6
4
2
1
2
3
4
5
6
7
8
9
10

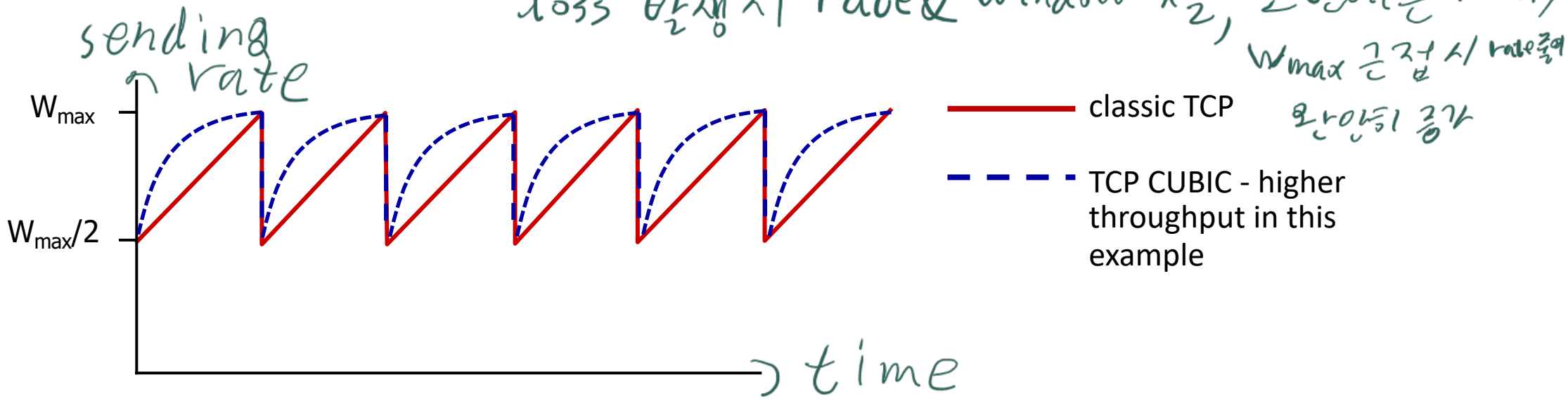

3 Dup-ACK:
• ssthresh = 7
• cwnd = 7+3
New ACK:
• cwnd = 7
Dup-ACK:
• cwnd = 10+1
ssthresh
ssthresh
AI
MD
case1
(drop)
14
Slow Start
Congestion Avoidance
Fast Recovery
Congestion Avoidance
cwnd [packets]
Transmission round [RTTs]
16
14
12
10
8
6
4
2
7
8
9
10
11
12
13
14
15
16
17
18
19
20

Summary: TCP congestion control



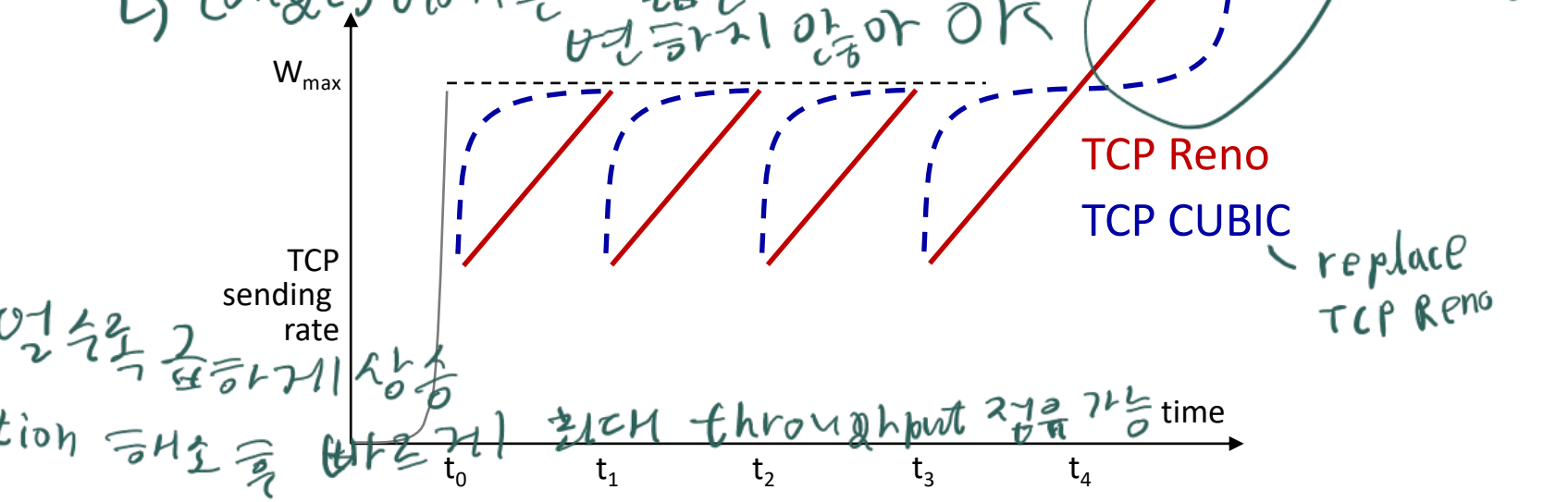
TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



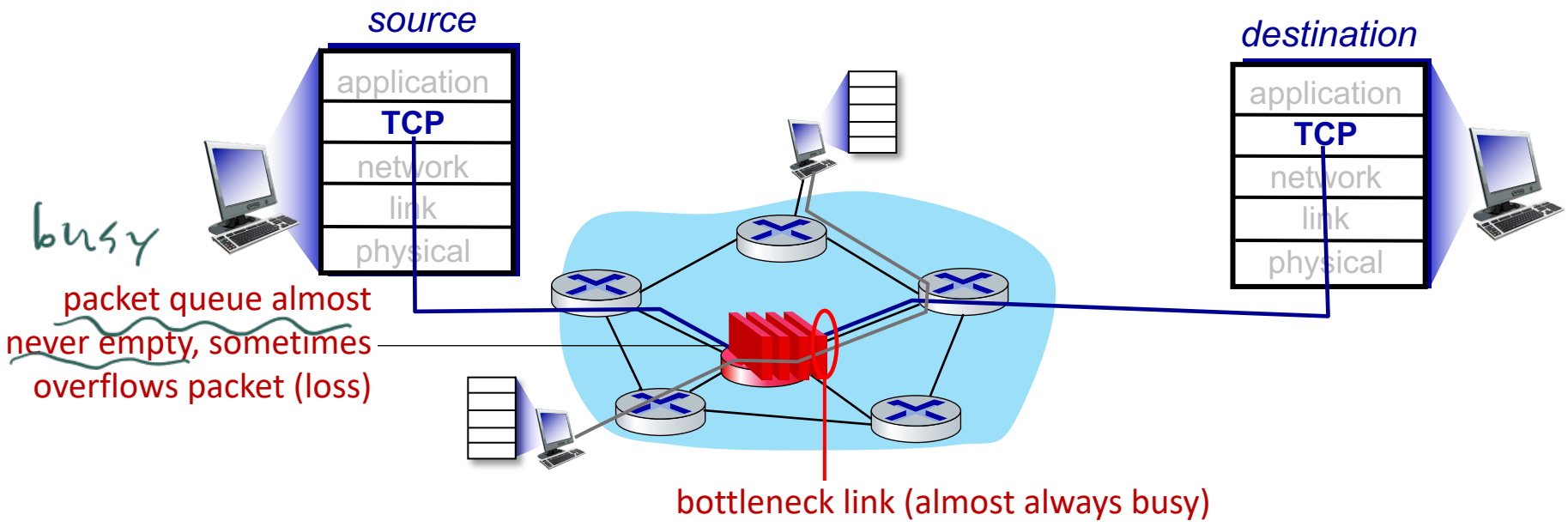
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



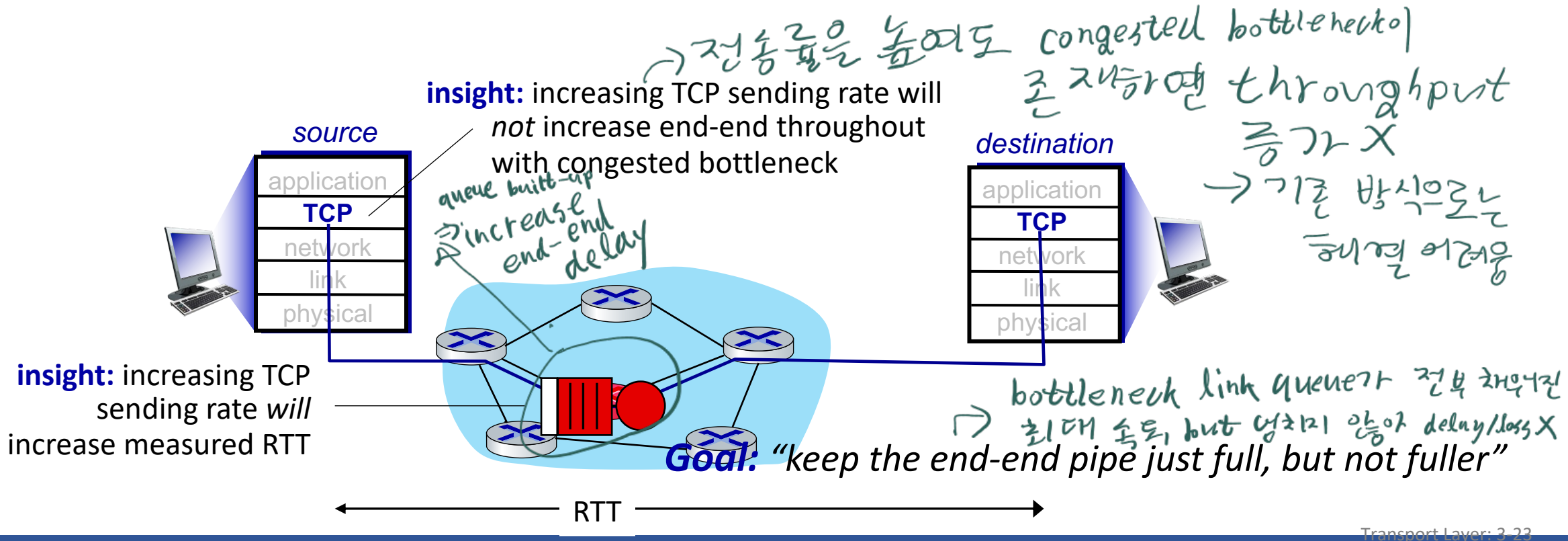
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



TCP and the congested "bottleneck link"

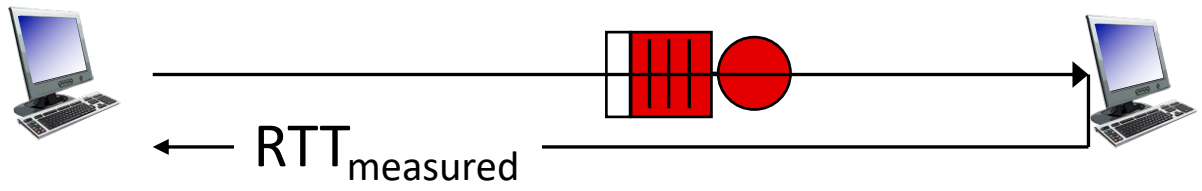
- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



Delay-based TCP congestion control

↳ bottleneck link에서 집중하는 방법

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{\text{RTT}_{measured}}$$

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is cwnd/RTT_{min}

혼잡X

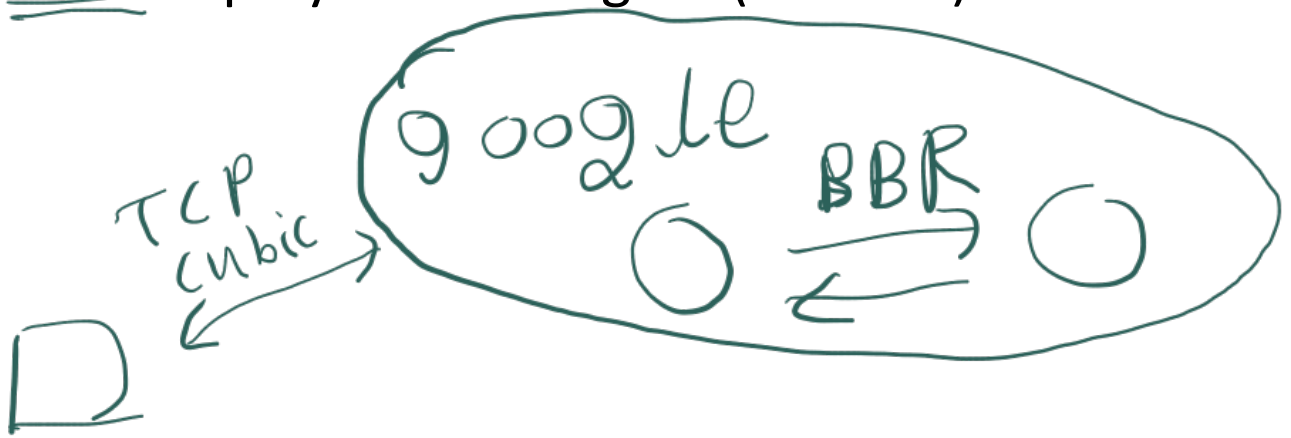
```
if measured throughput "very close" to  uncongested throughput
    increase cwnd linearly          /* since path not congested */
else if measured throughput "far below" uncongested throughout
    decrease cwnd  linearly         /* since path is congested */
```

→ 측정값과 최대치 유사 ⇒ 혼잡X 이므로 cwnd↑

→ 측정값과 최대치 차이↑ ⇒ 혼잡 상태이므로 cwnd↓

Delay-based TCP congestion control

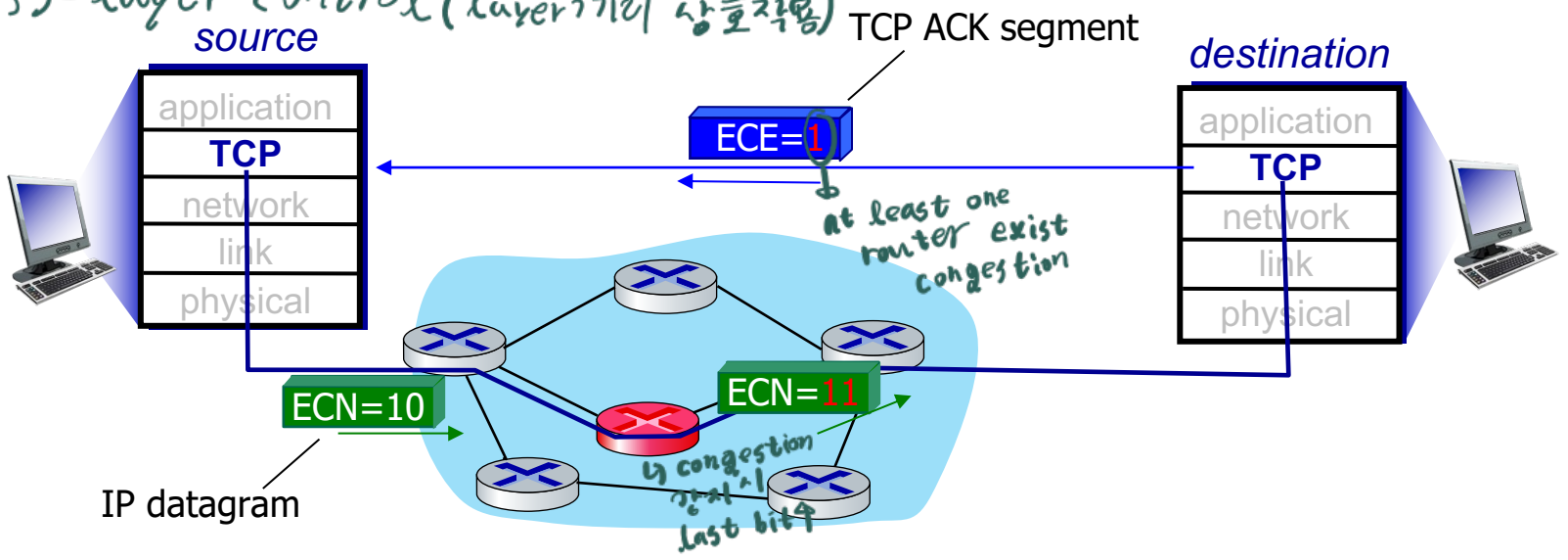
- congestion control without inducing/forcing loss
- maximizing throughout ("keeping the just pipe full… ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network



Explicit congestion notification (ECN)

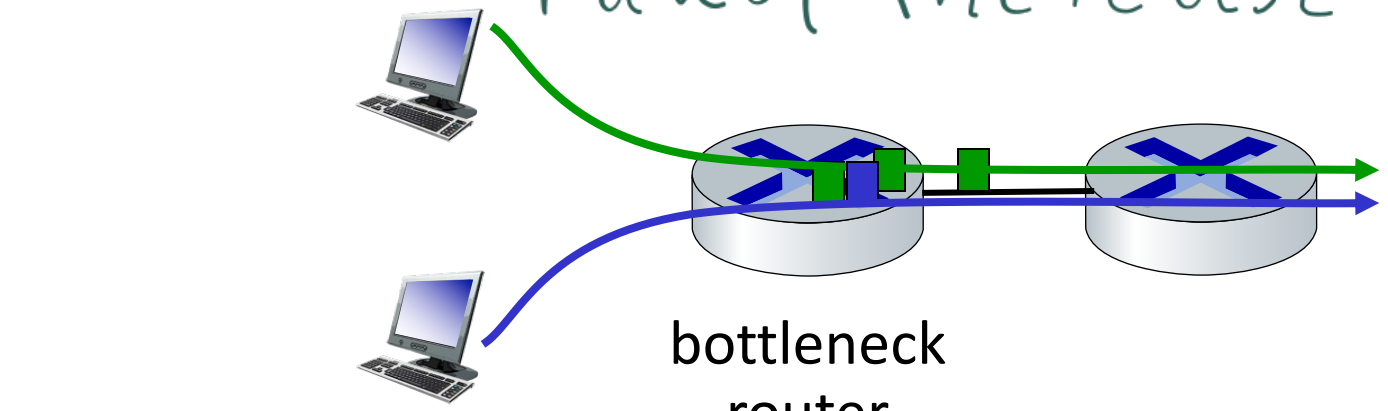
TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



TCP fairness

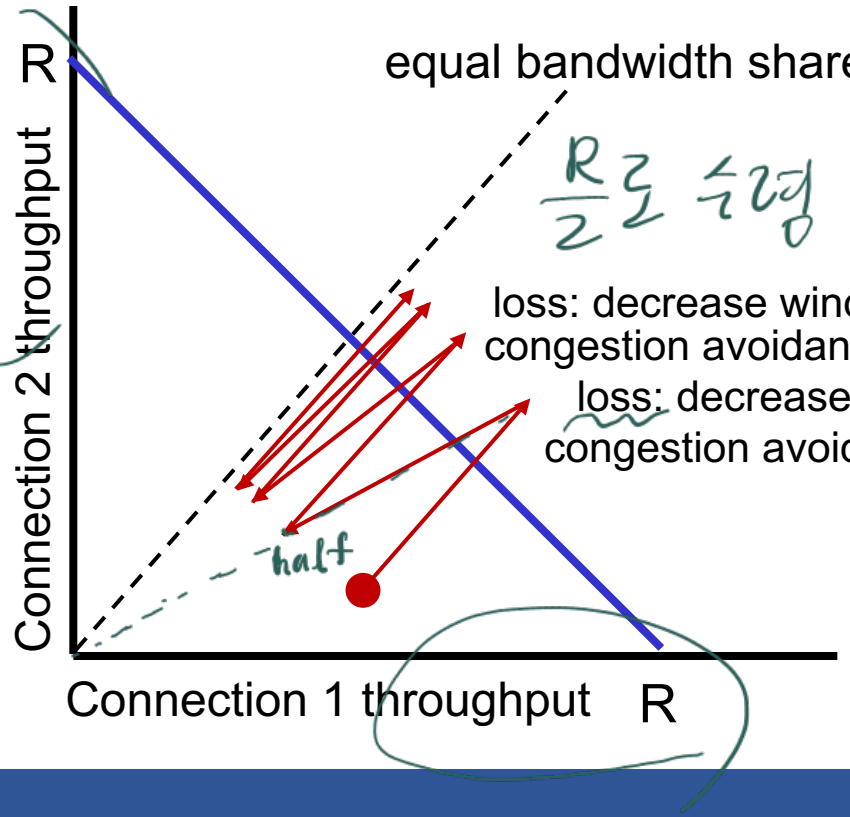
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be "fair"?

→ fairness is "not" robust ⇒ moniter the violent (개조 TCP로 과원동과지) traffic

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control (조절하다) → congestion control X ⇒ fair 에복 고민성 X
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss (허용하다) ↳ 고정값으로 제공
- there is no "Internet police" policing use of congestion control

UDP는 혼잡 통제가 없으나 그냥 사용

Fairness, parallel TCP connections ↳ TCP에는 기능 X

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 (11x R/10)

TCP는 connection 단위 fairness를 구현했기에 app에서 다수의 병렬 connection 사용시 fairness 보장 X

Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Evolving transport-layer functionality

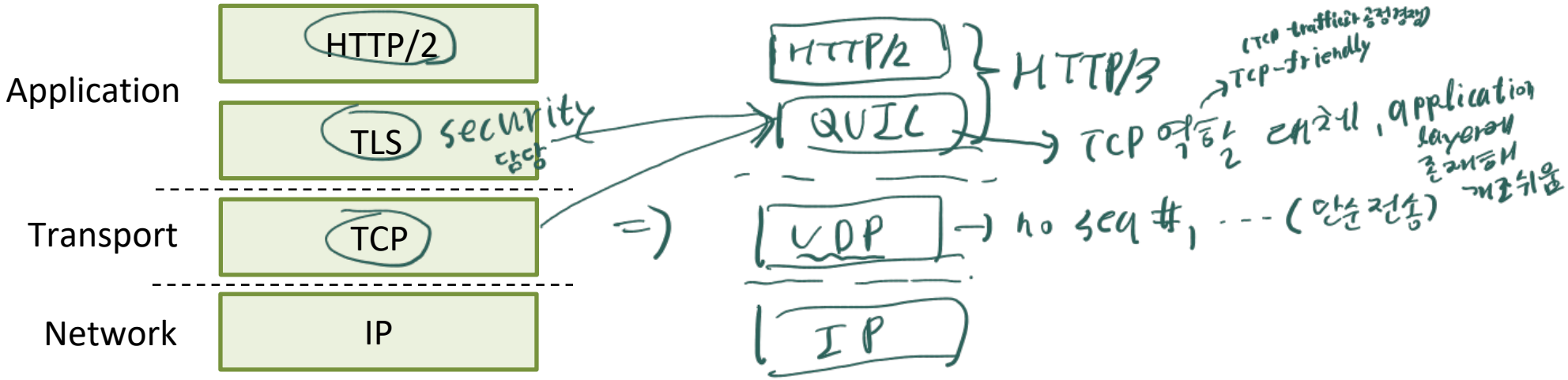
- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets "in flight"; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, "background" TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)



HTTP/2 over TCP

Mixed feelings about HTTP/3 (QUIC)

- https://youtu.be/wV9FSyFB8tk

Horrible, Helpful, http3 Hack - Computerphile

72K views • 3 months ago

Computerphile

http3 is here, but it wasn't an easy solution, Richard G Clegg of Queen

4K

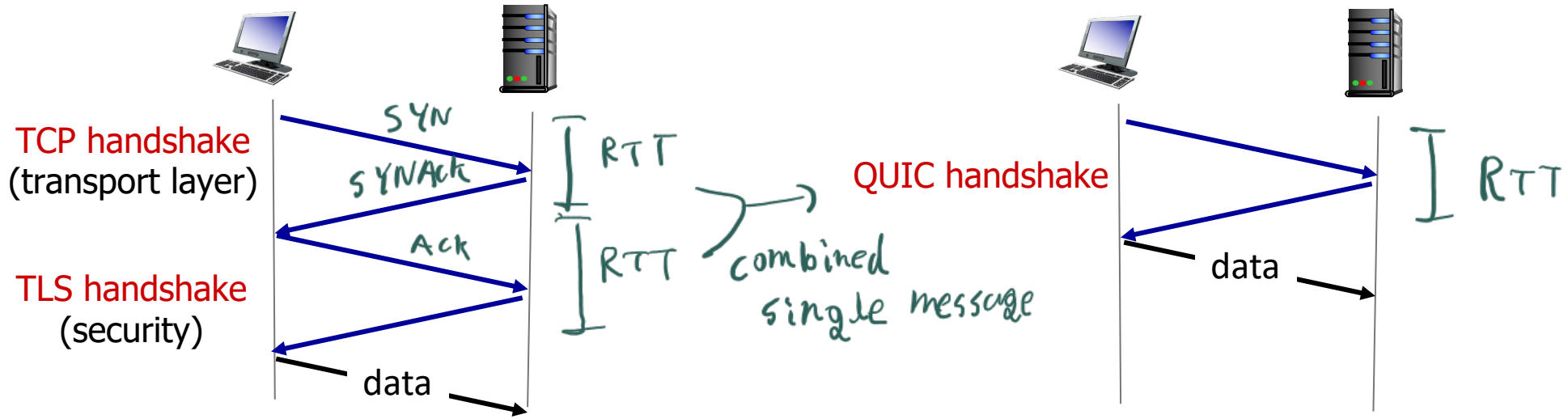
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Connection establishment



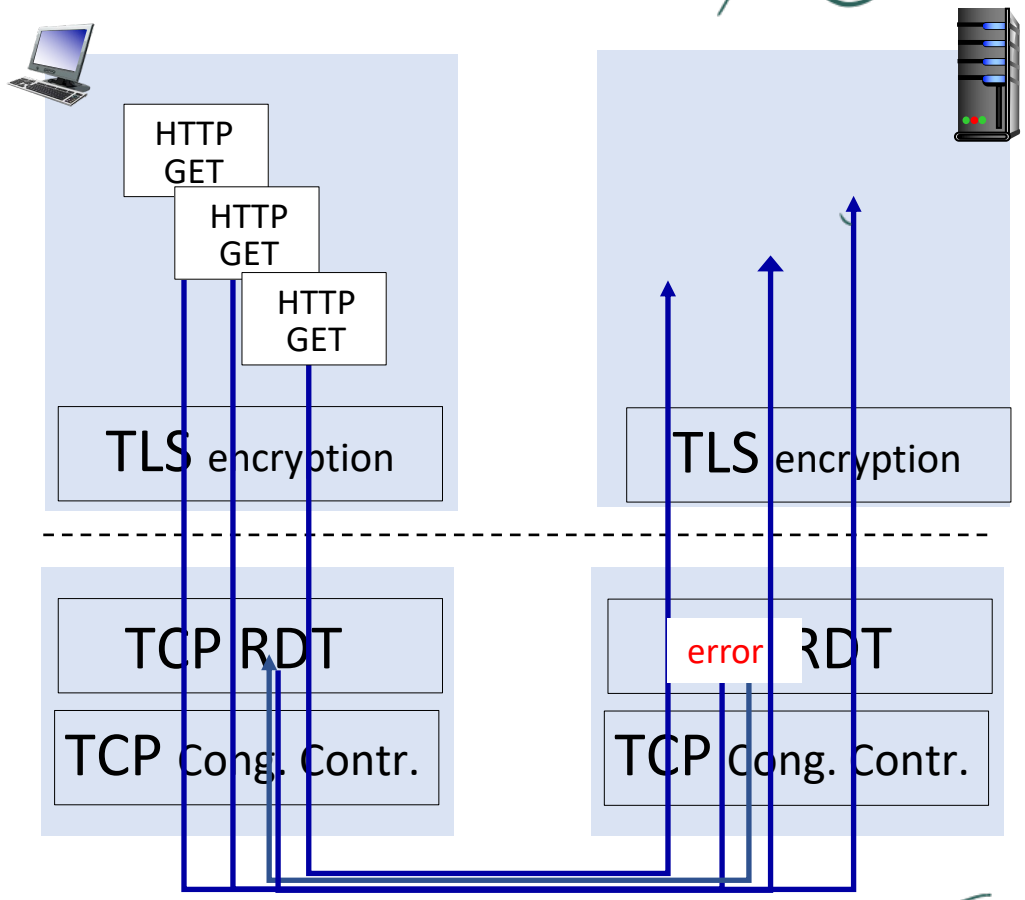
TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



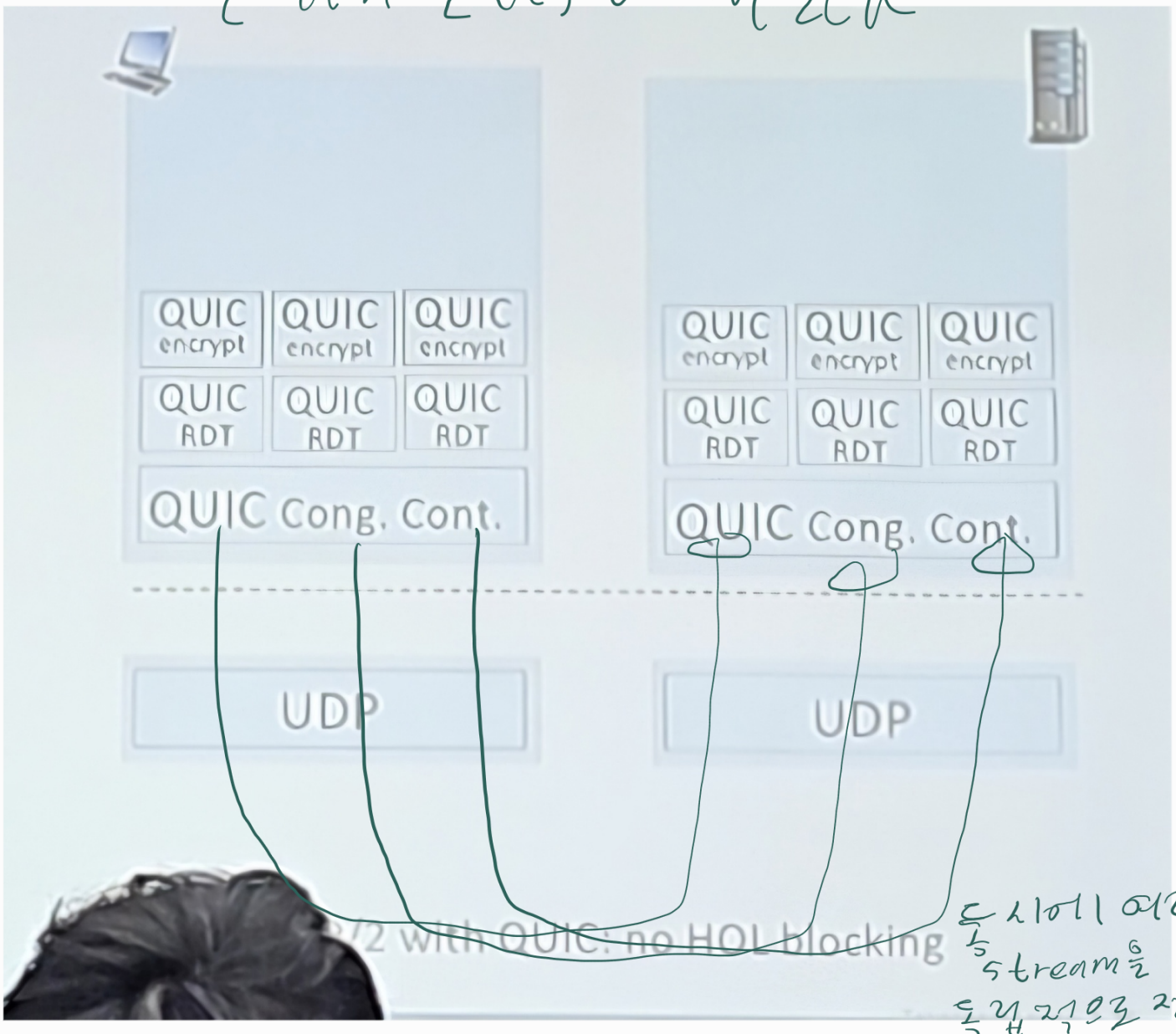
(a) HTTP 1.1

multiplex

↳ error 발생시 2개의 delay

하나의 stream

can customized



독립적인 여러 stream을 독립적으로 전송 (multiplexing)

Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane

Next…

- *4.1 Overview of Network Layer*
- *4.2 What's Inside a Router?*

  +
- *Review last year's midterm questions*