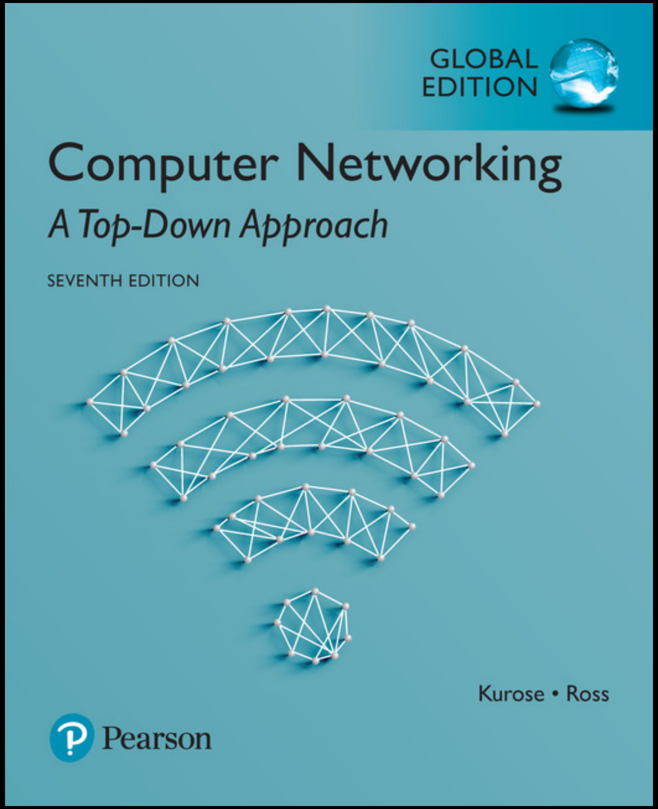
Computer Network

Transport Layer

School of Electric and Computer Engineering
Pusan National University, KOREA
Younghwan Yoo

Textbook



Computer Networking

A Top–Down Approach

7th edition

Jim Kurose, Keith Ross

Pearson

April 2016

Contents

Computer Network introduction

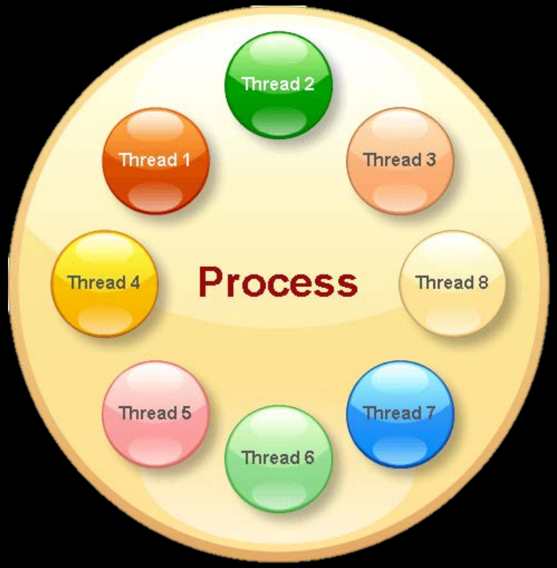
Contents

Computer Network introduction

01. Transport–Layer Services

Terminologies: Program, Process, and Thread

- Program
  - an executable file containing the set of instructions written to perform a specific job
  - stored on a disk
- Process
  - an executing instance of a program
  - resides on the primary memory
  - several processes related to same program at the same time
- Thread
  - the smallest executable unit of a process



출처 – https://www.quora.com/What-is-difference-between-thread-process-and-program

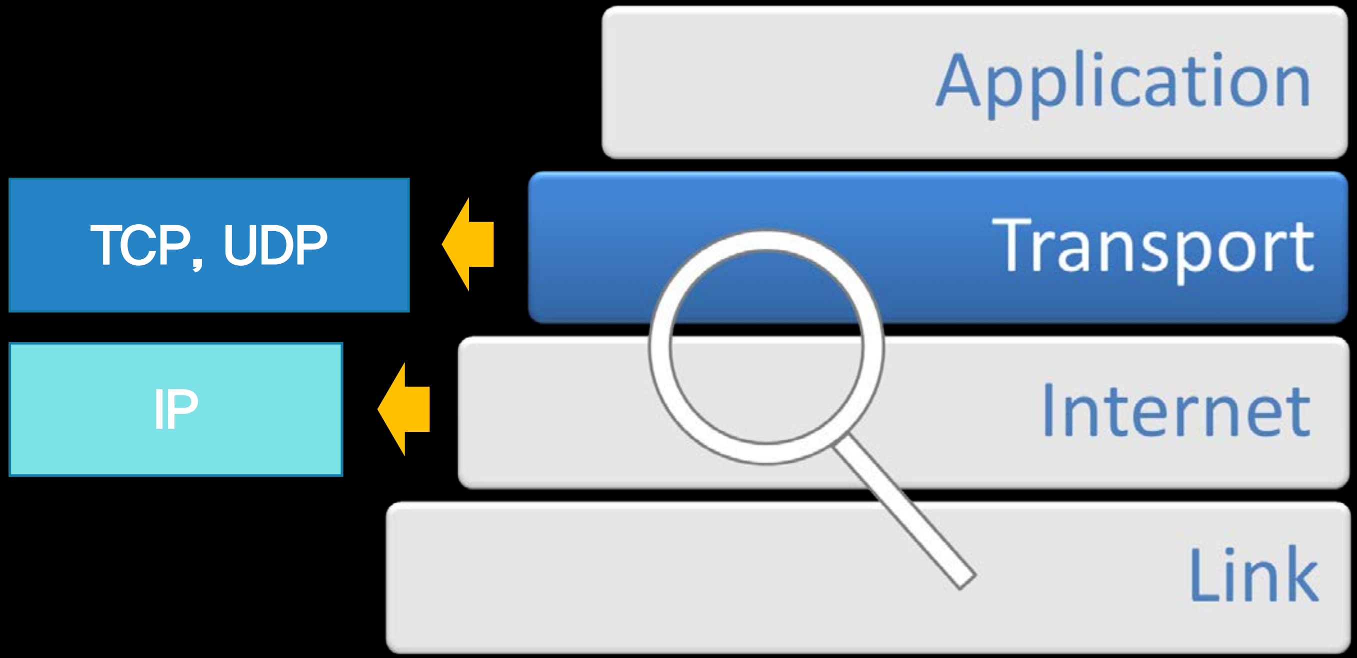
Transport Layer Function (1/2)

Transport layer

- logical communication between processes
- relies on, enhances, network layer services

Network layer

- logical communication between hosts



출처 – https://en.wikiversity.org/wiki/Internet_Protocol_Analysis/Transport_Layer

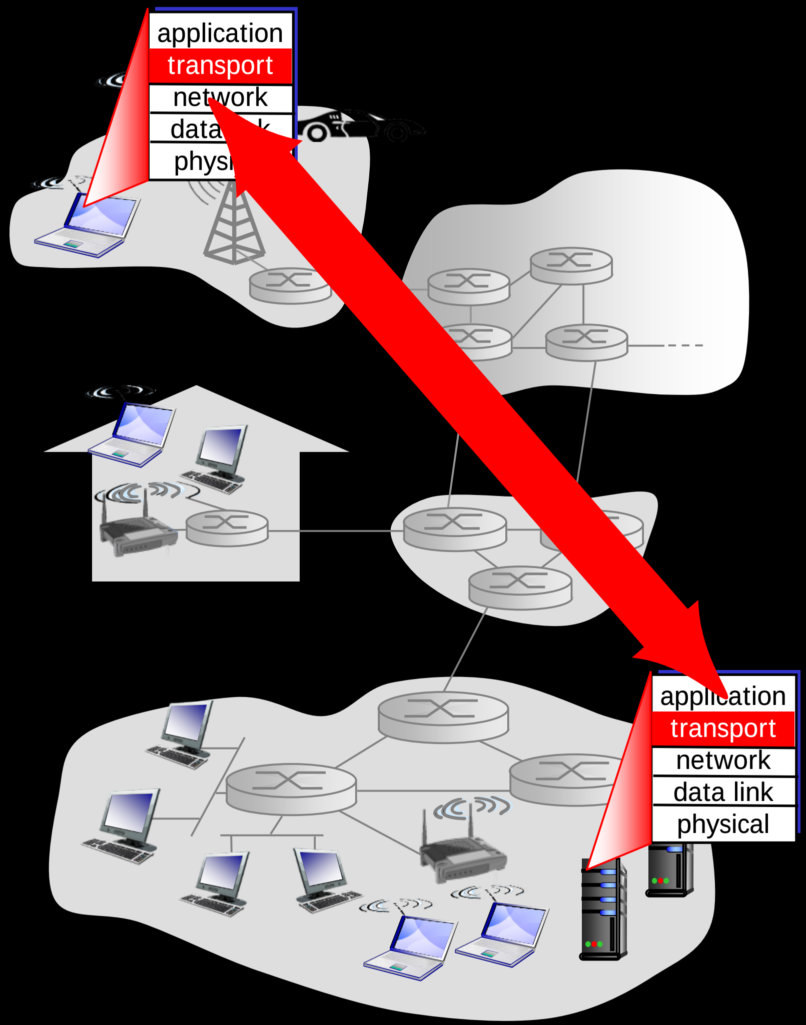
Transport Layer Function (2/2)

Send Side

- Application message fragmentation into segments
- Segment passing to network layer

Recv Side

- Segments reassembly into messages
- Message passing to application layer



Internet Transport Protocols

TCP, UDP

TCP vs. UDP

TCP

- Transmission Control Protocol
- Reliable, in–order delivery
- Connection–oriented service
  - connection setup
  - error control
  - flow control
  - congestion control

UDP

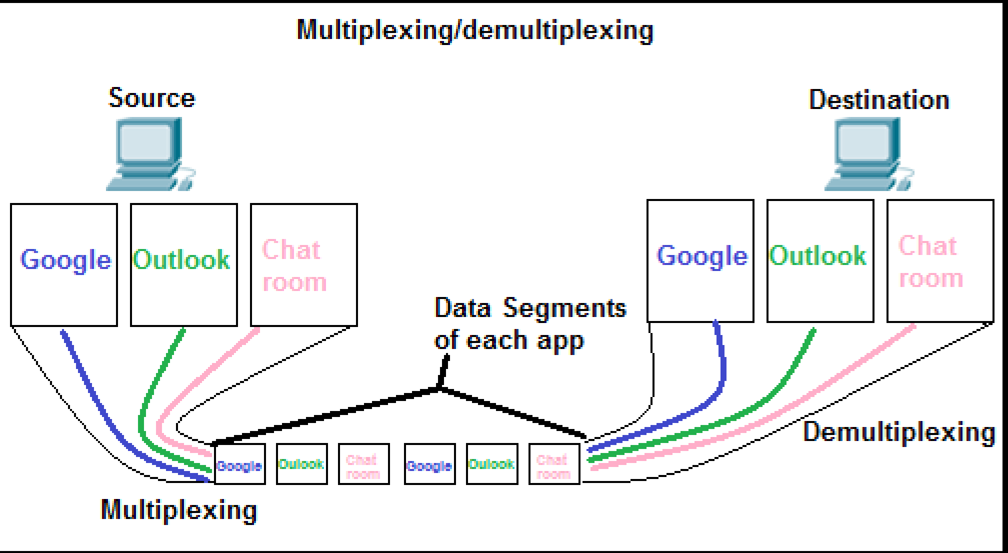
- User Datagram Protocol
- Unreliable, unordered delivery
- Connectionless service
  - faster than TCP

02. Multiplexing and Socket

Multiplexing and Demultiplexing

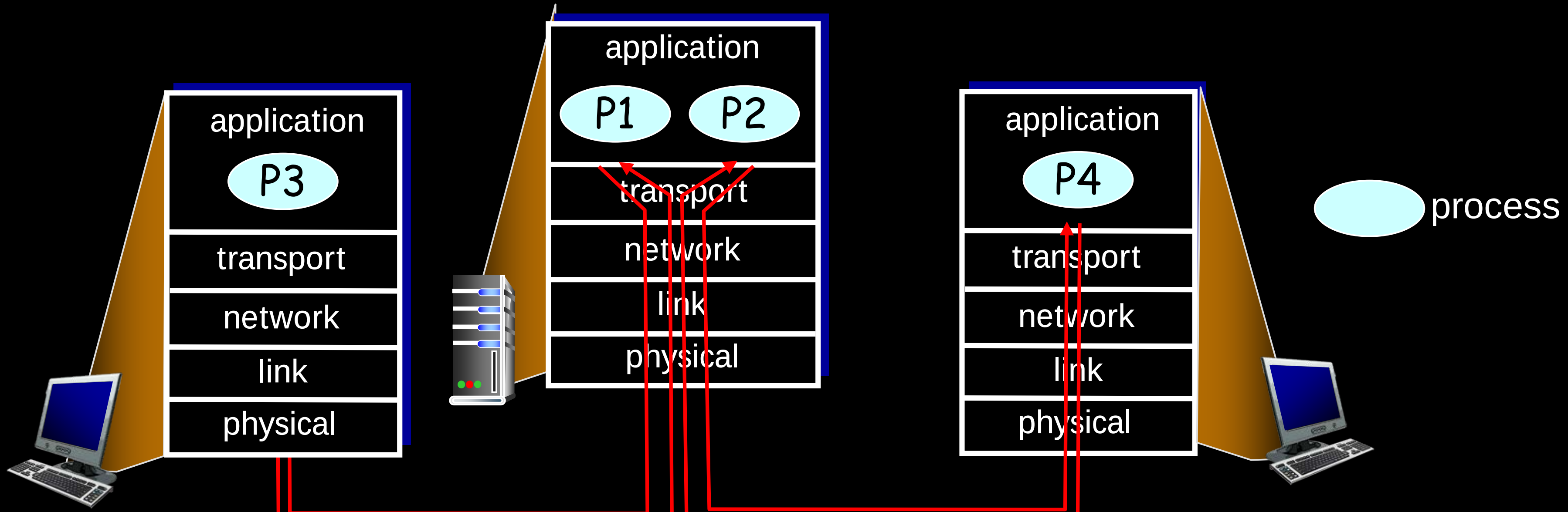
- Multiplexing at sender
  - sending data from its own multiple applications through network

- Demultiplexing at receiver
  - delivering data packets to their appropriate receivers among its own multiple applications



출처 – https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwiNn_qt7ubbAhUU87wKHVnKCbQQjRx6BAgBEAU&url=http%3A%2F%2Fwww.cnt4all.com%2F2016%2F08%2F07-transport-layer-multiplexing-and.html&psig=AOvVaw03CLdYUqYYGWH6m155fosy&ust=1529740774014370

Who Provide Multiplexing and Demultiplexing

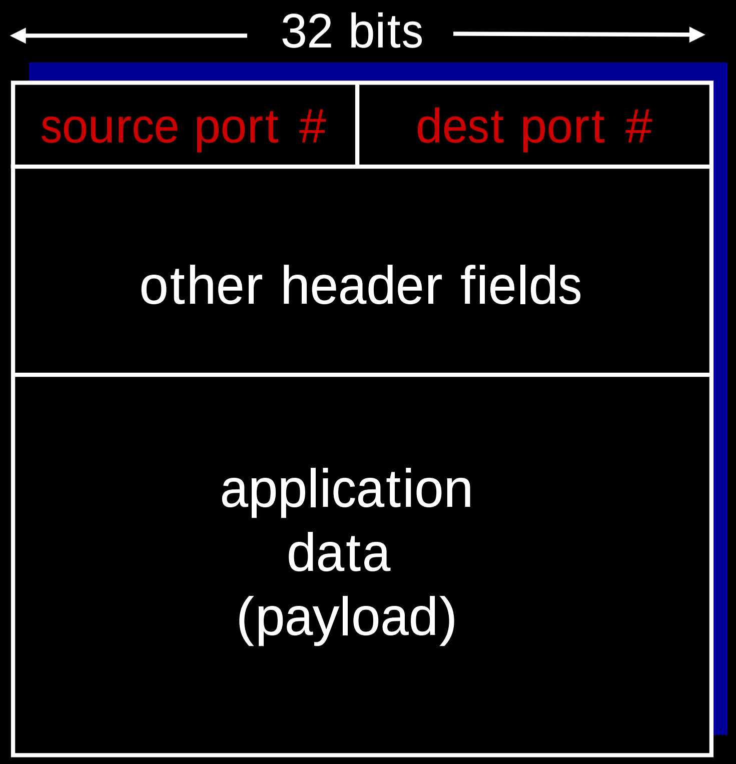


- The most basic role of the transport layer.

How Demultiplexing Works

- Port number
  - different applications are assigned different port numbers
  - transport layer segments have fields for src/dst port numbers in common
  - used for differentiating segments
- However, the transport layer must be instructed by the application which process on which host should be the destination. How?

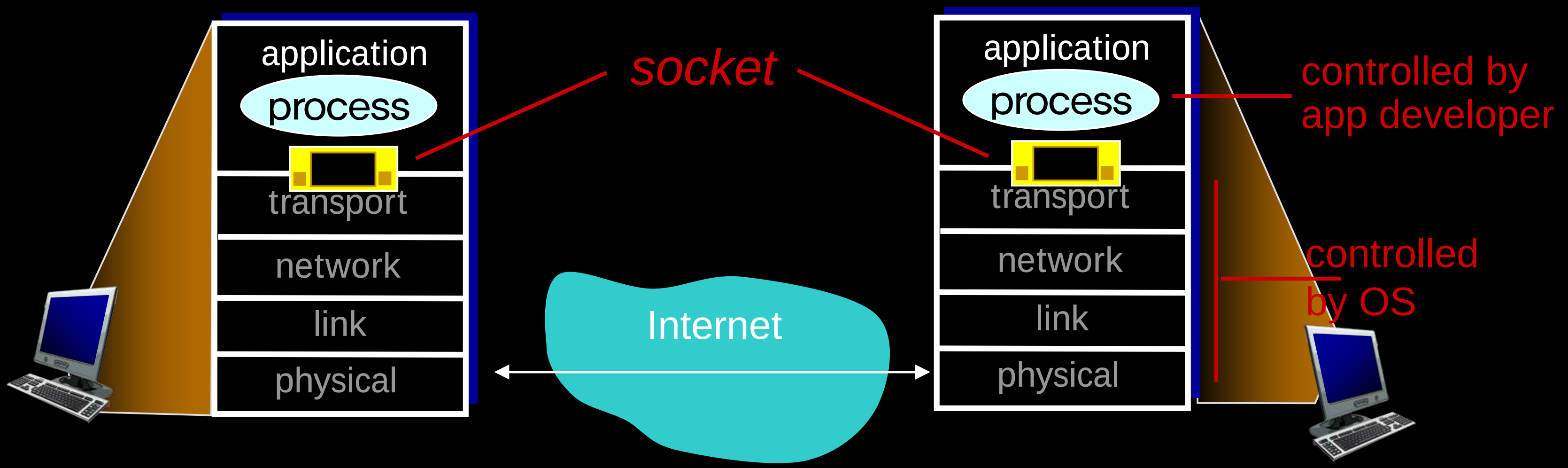
➡ *Socket !!!*



TCP/UDP segment format

Socket

- Process sends/receives messages to/from its socket
  - e.g., BSD socket, Winsock
- Analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



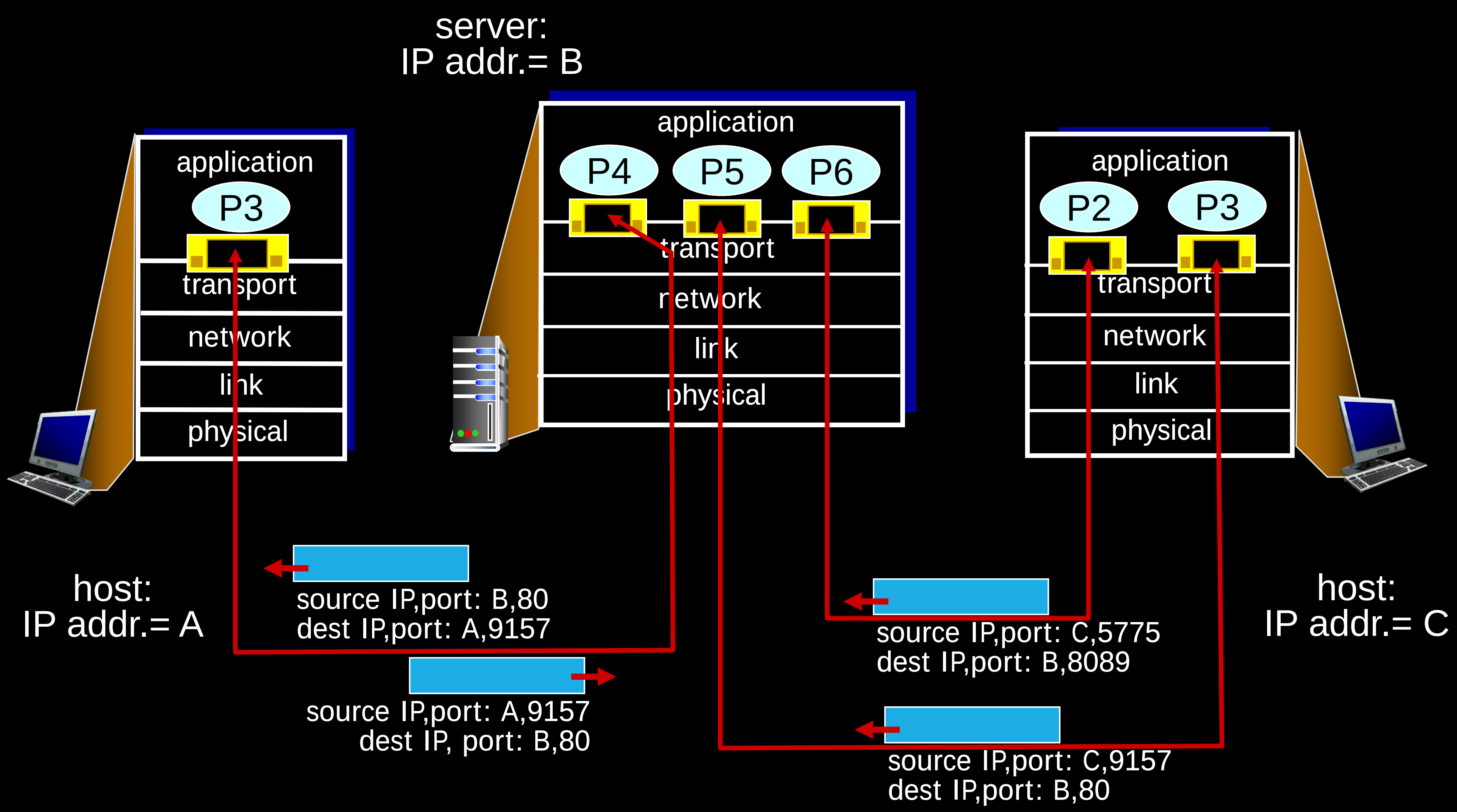
- The 'Socket' is provide as the form of APIs (application program interfaces) by OS
- Server

```python
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('127.0.0.1', 8089))  # localhost = 127.0.0.1
serversocket.listen(5)

while True:
    connection, address = serversocket.accept()
    buf = connection.recv(64)
    if len(buf) > 0:
        print(buf)
        break
```

- Client

```python
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send(b'GET /index.html')
```

Demultiplexing Example

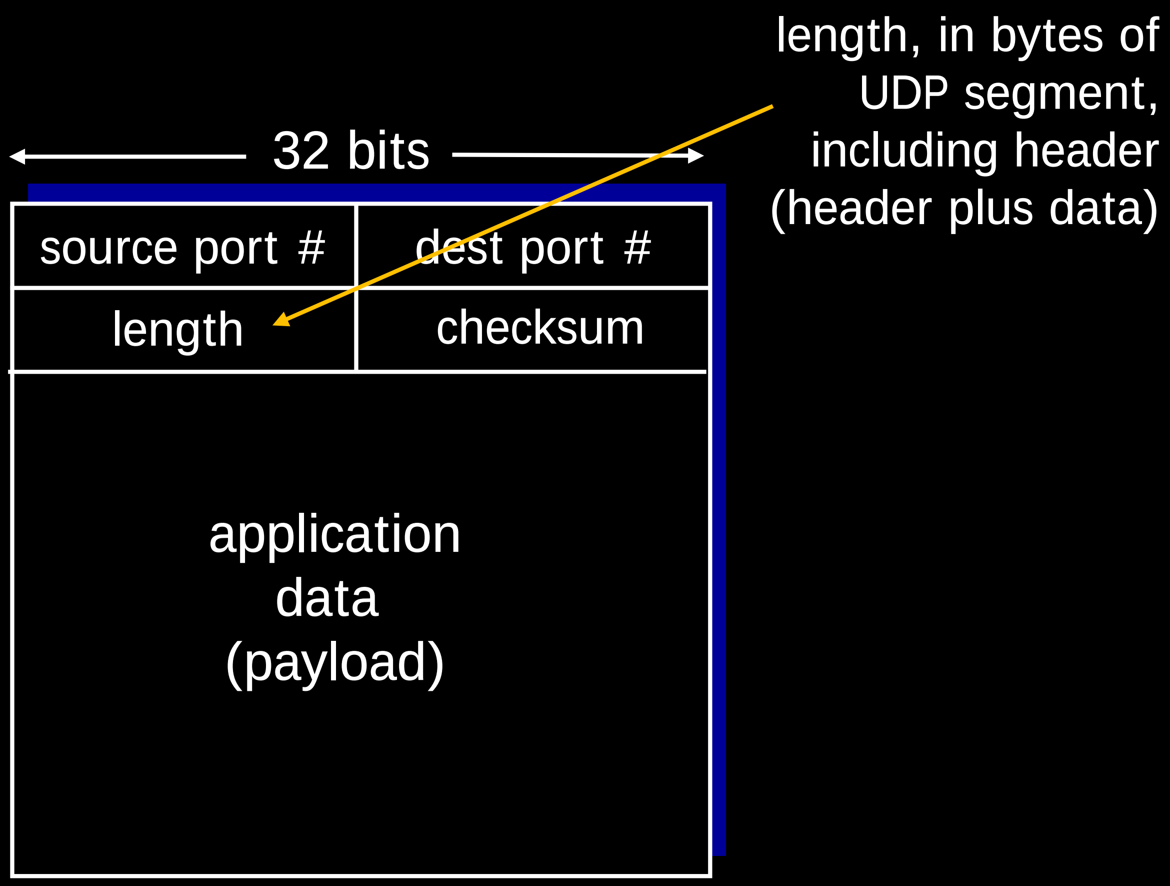


03. User Datagram Protocol

User Datagram Protocol [RFC 768]

- "No frills," "bare bones" Internet transport protocol
- <u>Connectionless service:</u>
  - each UDP segment handled independently of others
  - Unreliable: UDP segments may be lost or delivered out-of-order to app
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- Reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

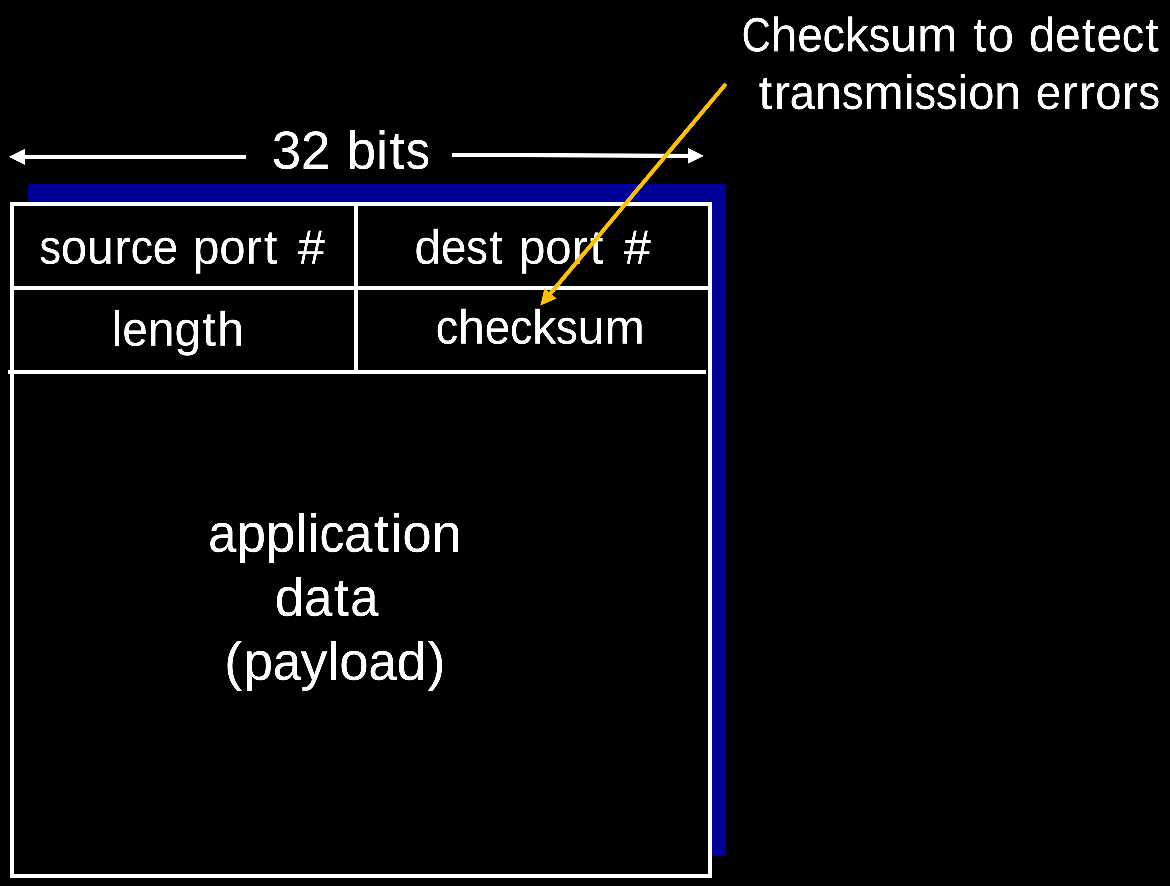
UDP Segment Header



UDP segment format

- Advantages of UDP
  - no connection establishment (which can add delay)
  - simple: no connection state at sender, receiver
  - small header size
  - no congestion control: UDP can blast away as fast as desired

UDP Checksum



UDP segment format

- Sender
  - make a 16–bit integer checksum code of segment contents including header
  - put the checksum into the checksum field and transmit the segment
- Receiver
  - compute checksum of received segment
  - check if the computed checksum equals all 1's or not

Internet Checksum Example

Ex) splits an entire segment into 16-bit words

and add them one by one

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

1s complement

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

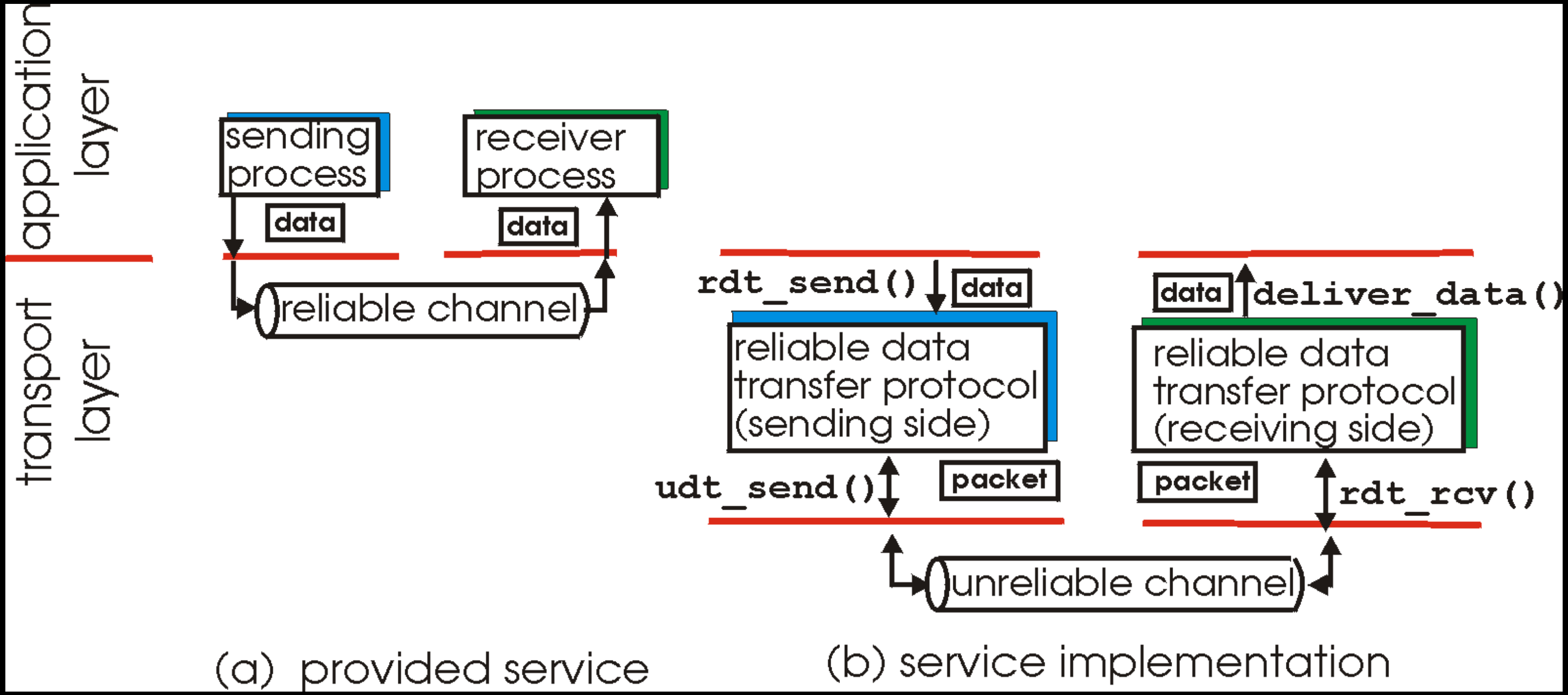
- Why checksum in UDP?

04. Reliable Data Transfer Principles

Principles of Reliable Data Transfer

- Services abstraction (provided to the upper-layer) through a reliable channel
  - no corruption and no lost of data
  - delivered in the order in which they were sent

Service model of TCP



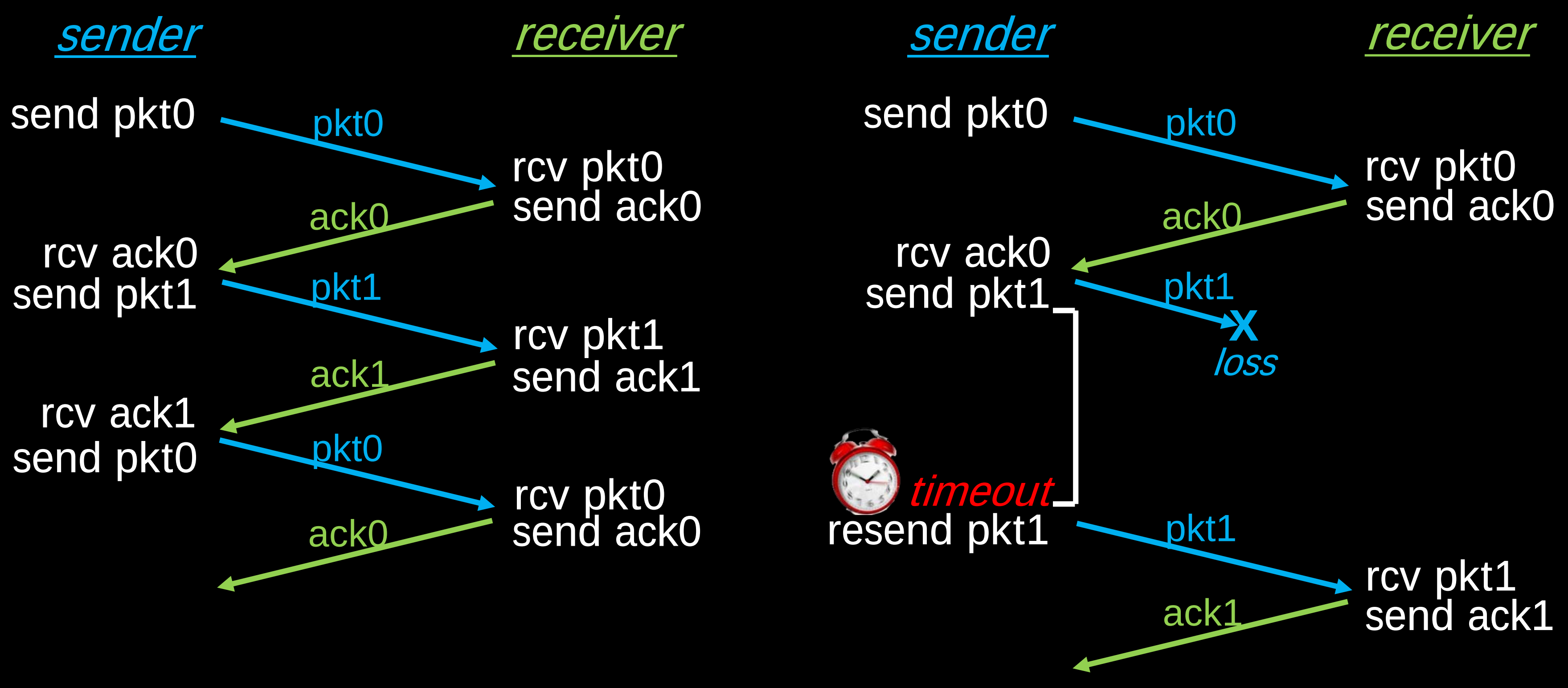
Error Type & Solution

Error Types	Solution
Bit error	• Checksum in every segment • ACK returned for successfully received packet
Packet loss (Data or ACK)	• Timeout of sender’s timer • Packet retransmission • Packet sequence number for 1) ordered delivery 2) data duplication prevention

- Packet retransmission method

- ARQ (Automatic Repeat reQuest)
  - stop-and-wait
  - pipelining method
    - go-back-N
    - selective repeat

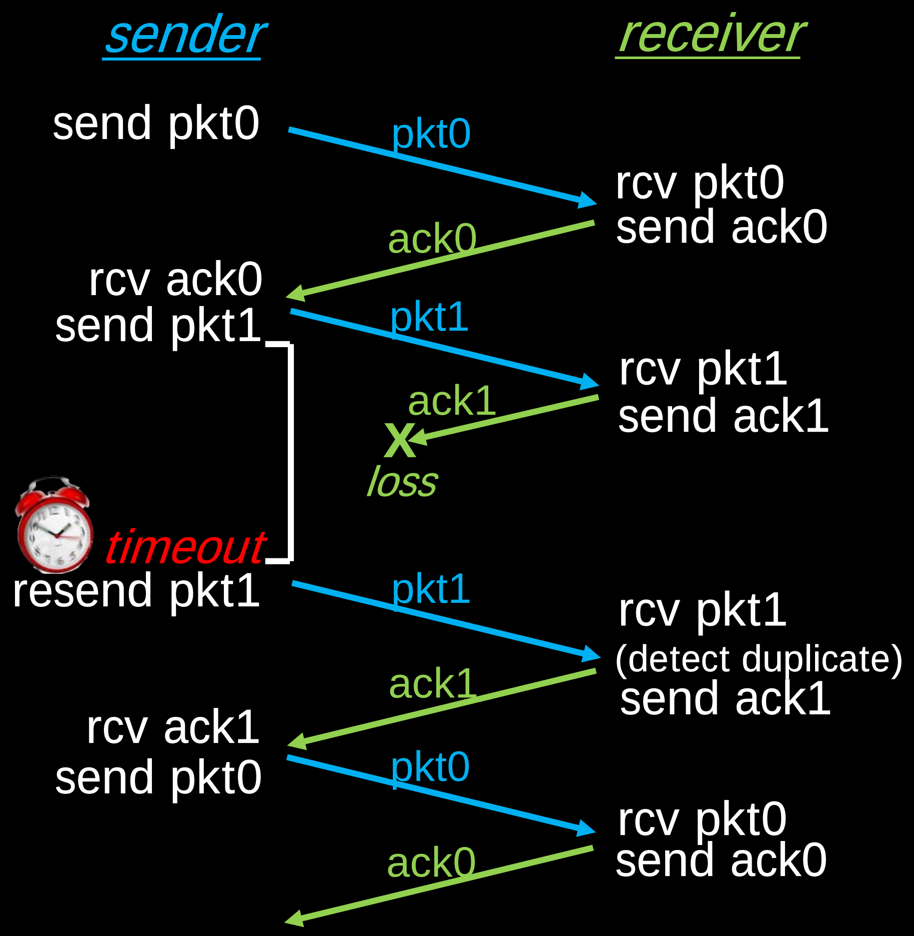
- Stop–and–wait
  - sender sends one packet, then waits for receiver response
  - after receiving ACK, sender resumes transmission
  - if timer expires without receiving ACK, sender retransmits the previous packet



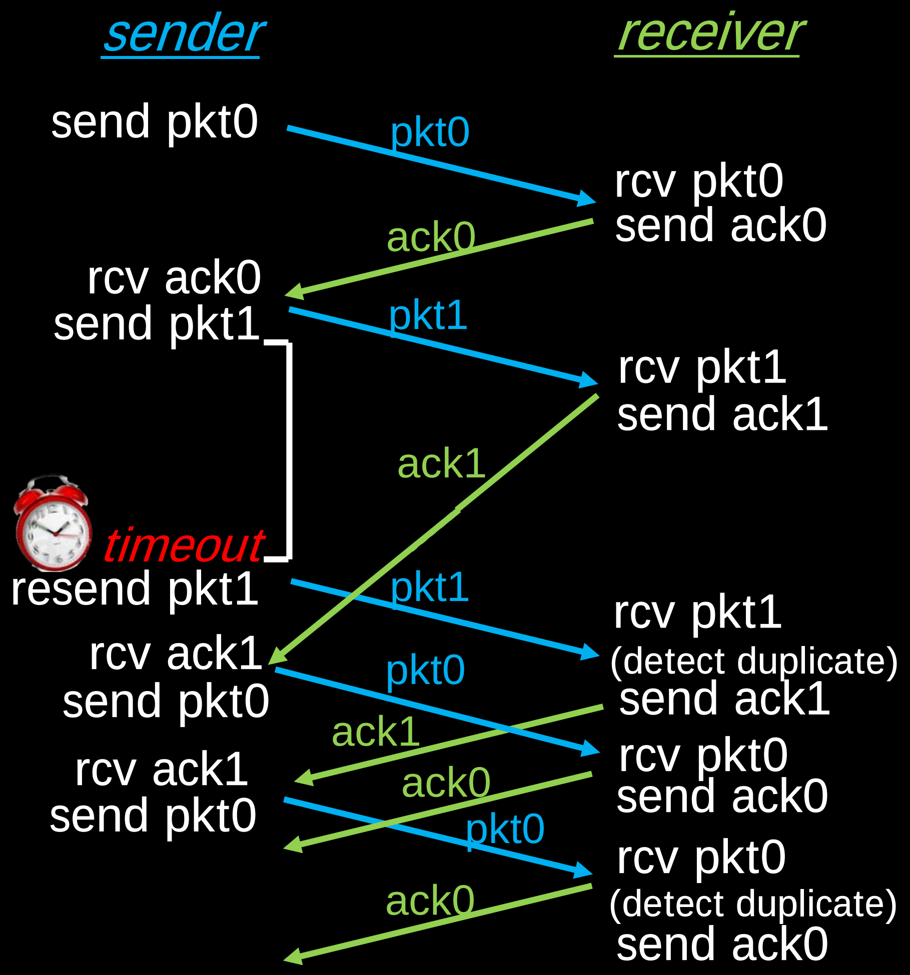
(a) no loss

(b) packet loss

Stop-and-Wait (2/2)



(c) ACK loss

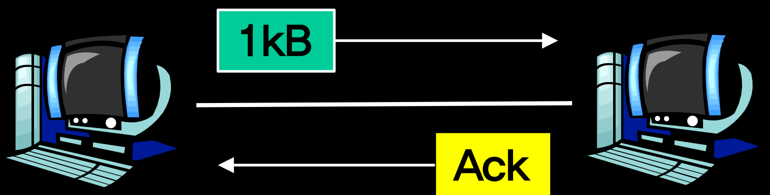


(d) premature timeout/ delayed ACK

Performance Limit of Stop-and-Wait

- E.g., 1 Gbps link, 15 ms propagation delay, 1kB (= 8000 bit) packet

$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ \mu s$$

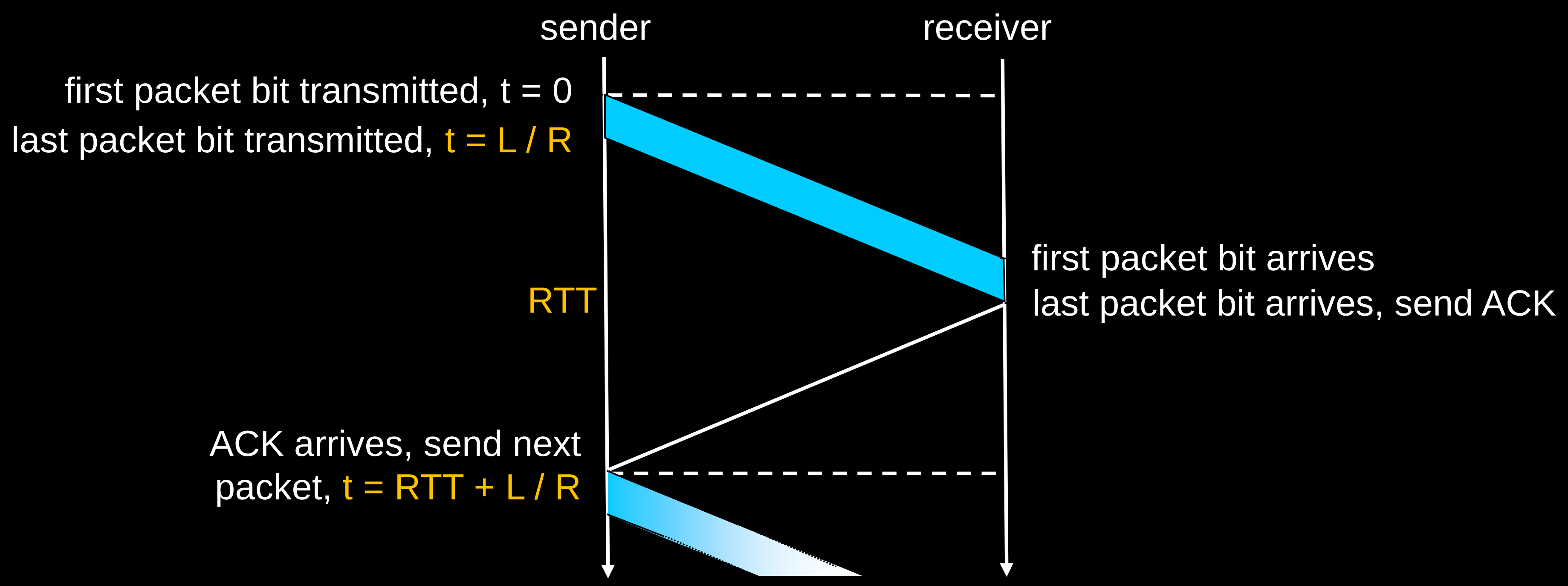


  - U : utilization – fraction of time sender busy sending

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

  - if RTT=30ms, 1kB packet every 30 ms: 33kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!

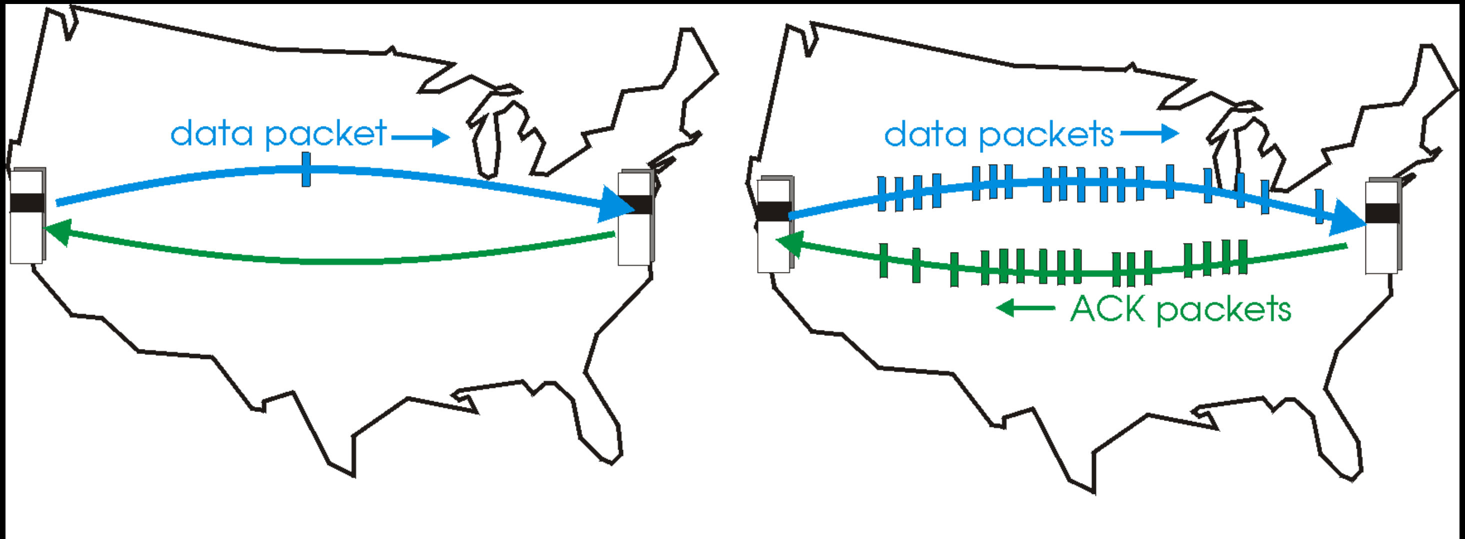
Stop-and-Wait Operation



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

Pipelined Protocols

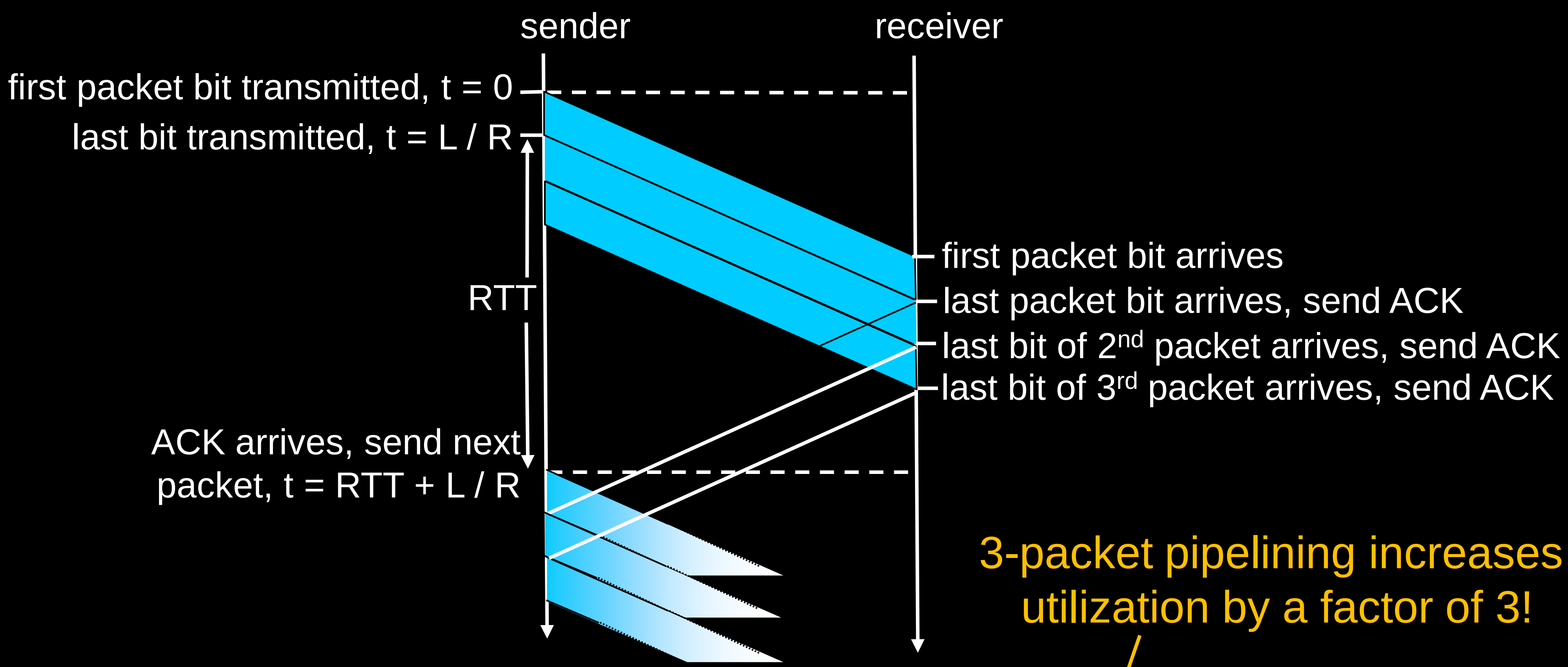
- Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-back-N, selective repeat*

Increased Utilization



3-packet pipelining increases utilization by a factor of 3!

$$U = \frac{3L/R}{RTT + L/R} = \frac{.024}{30.008} = 0.00081$$

Go-back-N

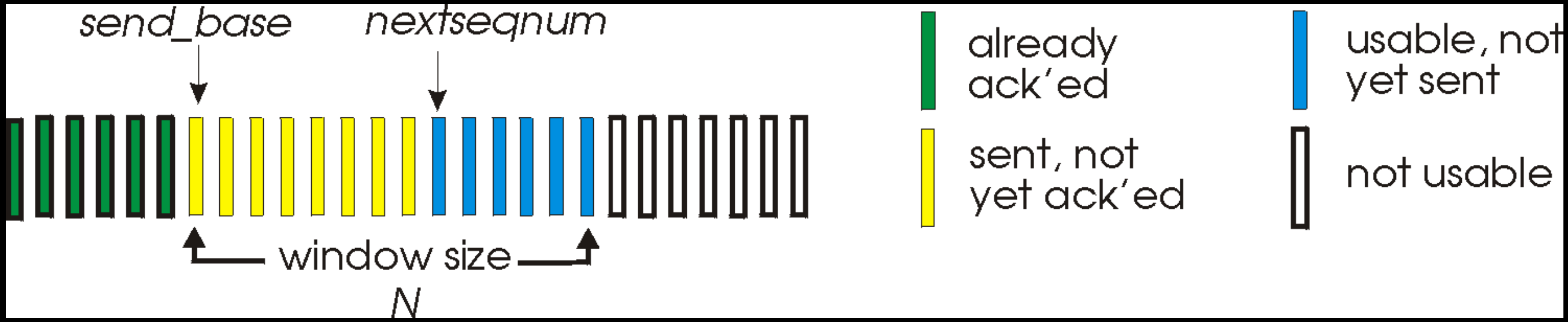
- Sender can have up to *N* unacked packets in pipeline
- Receiver only sends cumulative ack
  - in case a gap in seq. #, resends the last ack
- Sender has timer for oldest unacked packet
  - when timer expires, retransmit all unacked packets

Selective repeat

- Sender can have up to *N* unacked packets in pipeline
- Receiver sends individual ack for each packet
- Sender has timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Go-Back-N: Sender

- Sequence number in packet header
- "Window" of up to *N*, consecutive unacked packets allowed

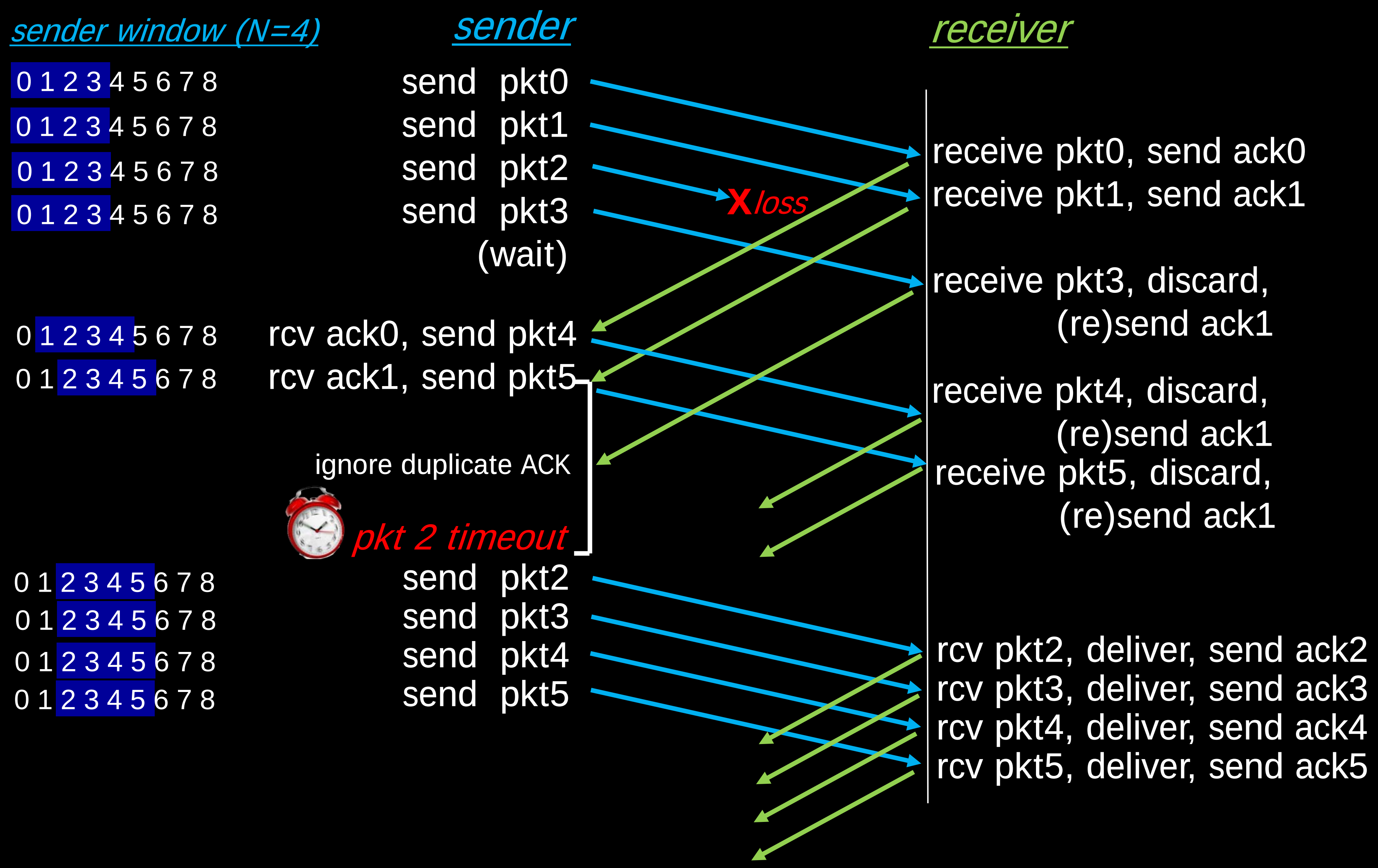


Sliding window

Window size *N* is dynamically changed in order to maximize the throughput, on condition that it does not cause network congestion or receiver buffer overflow

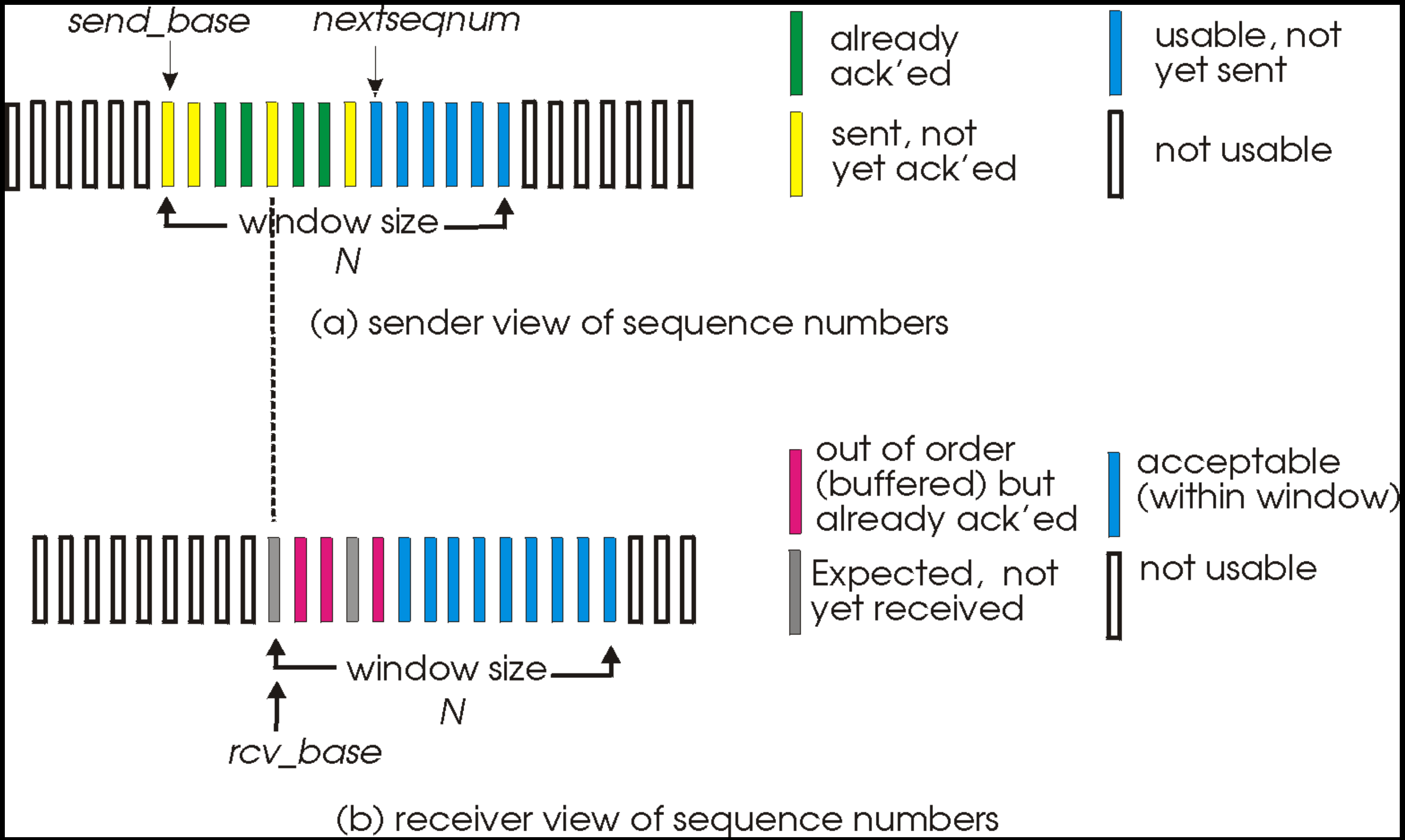
- Cumulative ACK: acknowledges all packets up to
- Timer for oldest in-flight packet, *send_base*
  - when timer expires for any packet with seq. # *n*, packet *n* and all higher seq. # packets in window are retransmitted

GBN Operation

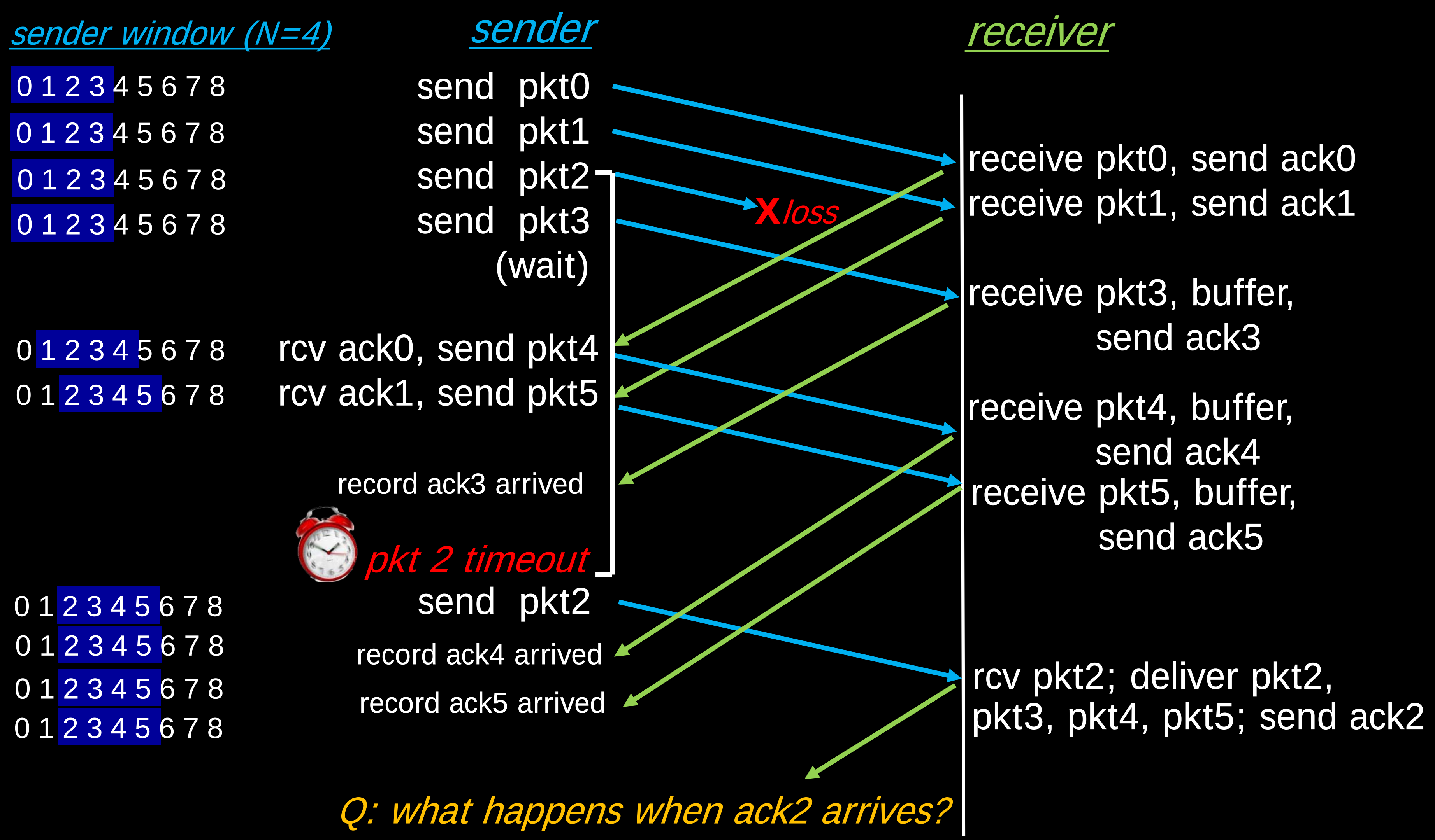


Selective Repeat: Sender and Receiver Windows

- Sender only resends packets for which ACK not received
  - sender timer for each unACKed packet
- Receiver individually acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer



Selective Repeat Operation

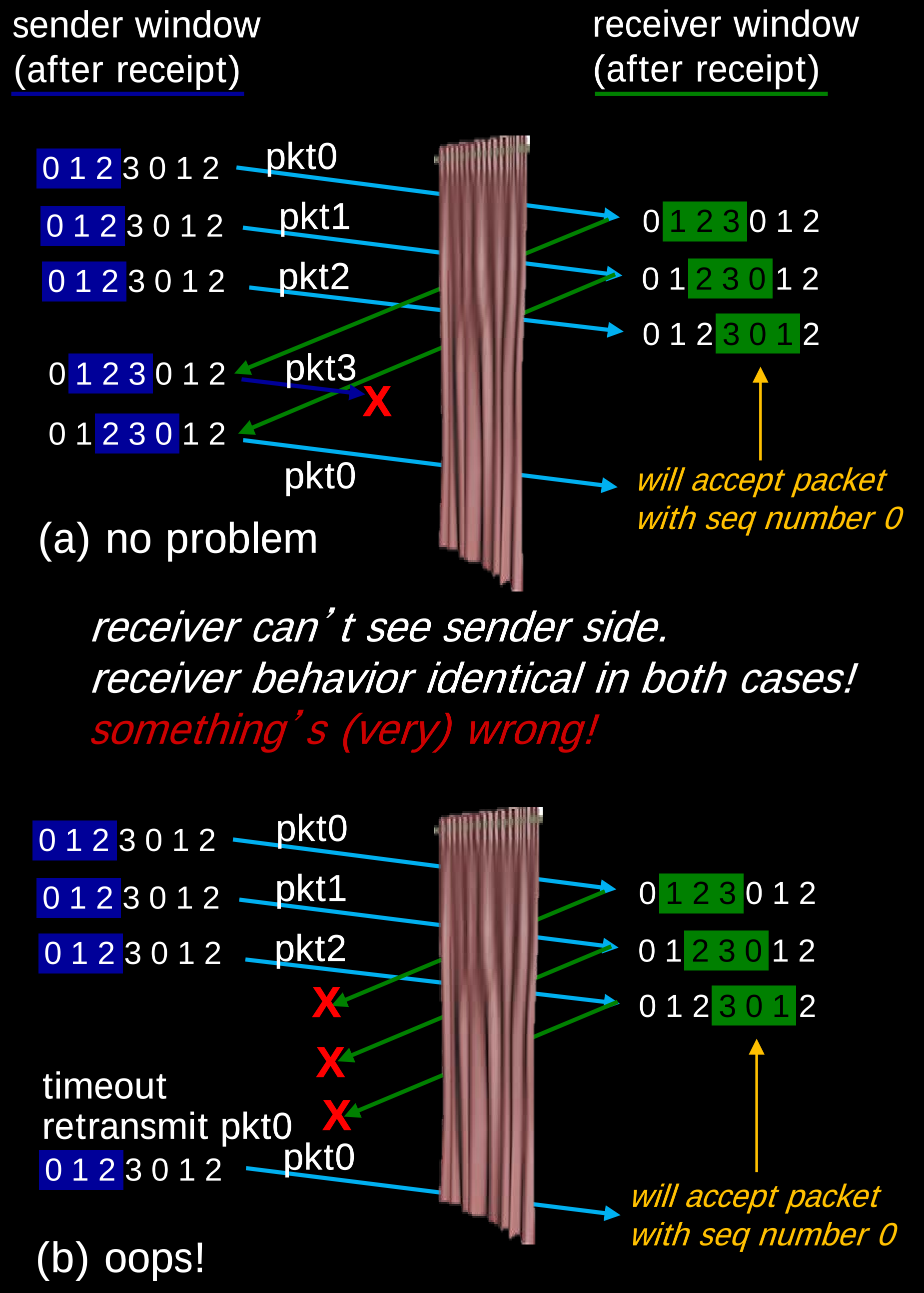


Relation Seq. Number & Window Size in SR

example:

- Seq. #'s: 0, 1, 2, 3
- Window size=3
- Receiver sees no difference in two scenarios!
- Duplicate data accepted as new in (b)

Q: What relationship between sequence number size and window size to avoid problem in (b)?



05. Transmission Control Protocol

TCP Overview

- Point-to-point: one sender, one receiver
- Connection-oriented service
  - Reliable transfer, in-order delivery
  - handshaking initializes sender and receiver state before data exchange
- Pipelined transmission
  - window size is set by TCP congestion and flow control
  - congestion control: transmission rate controlled not to make congestion in network
  - flow control: sender will not overwhelm receiver
  - MSS: maximum segment size
- Full-duplex connection
  - bi-directional data flow in same connection

TCP Segment Structure

Reserved for future use

Data Offset = Header length

Internet checksum (as in UDP)

TCP Segment Header Format

Bit #	0 … 7	8 … 15	16 … 23	24 … 31
0	Source Port		Destination Port	
32	Sequence Number			
64	Acknowledgment Number			
96	Data Offset / Res	Flags	Window Size	
128	Header and Data Checksum		Urgent Pointer (not used)	
160…	Options			

Flags: URG | ACK | PSH | RST | SYN | FIN

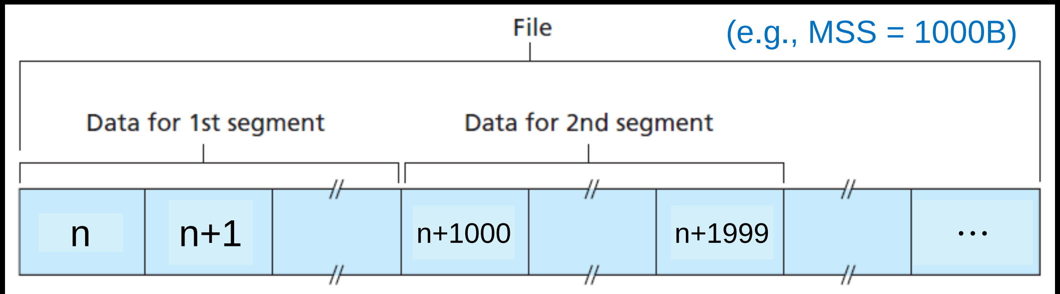
cf.

UDP Datagram Header Format

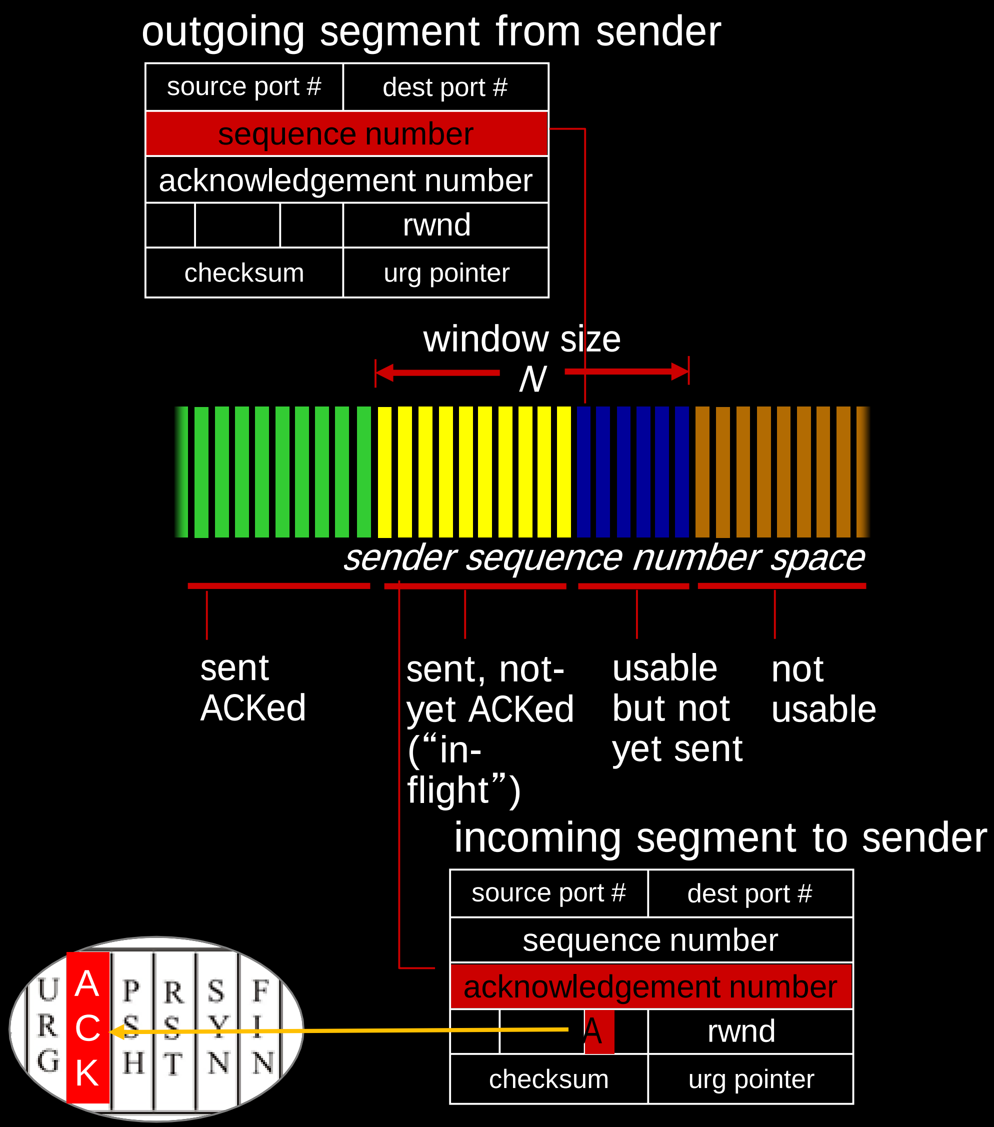
Bit #	0 … 7	8 … 15	16 … 23	24 … 31
0	Source Port		Destination Port	
32	Length		Header and Data Checksum	

출처 – https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjqvdKTnvXbAhWGgbwKHbLYChcQjRx6BAgBEAU&url=https%3A%2F%2Fwww.e-smartsolution.co.uk%2Fblog%2F2905-which-of-the-following-are-fields-in-the-tcp-header-2%2F&psig=AOvVaw3UKZiZgl2sHnBzm7KLhwL8&ust=1530175590273779

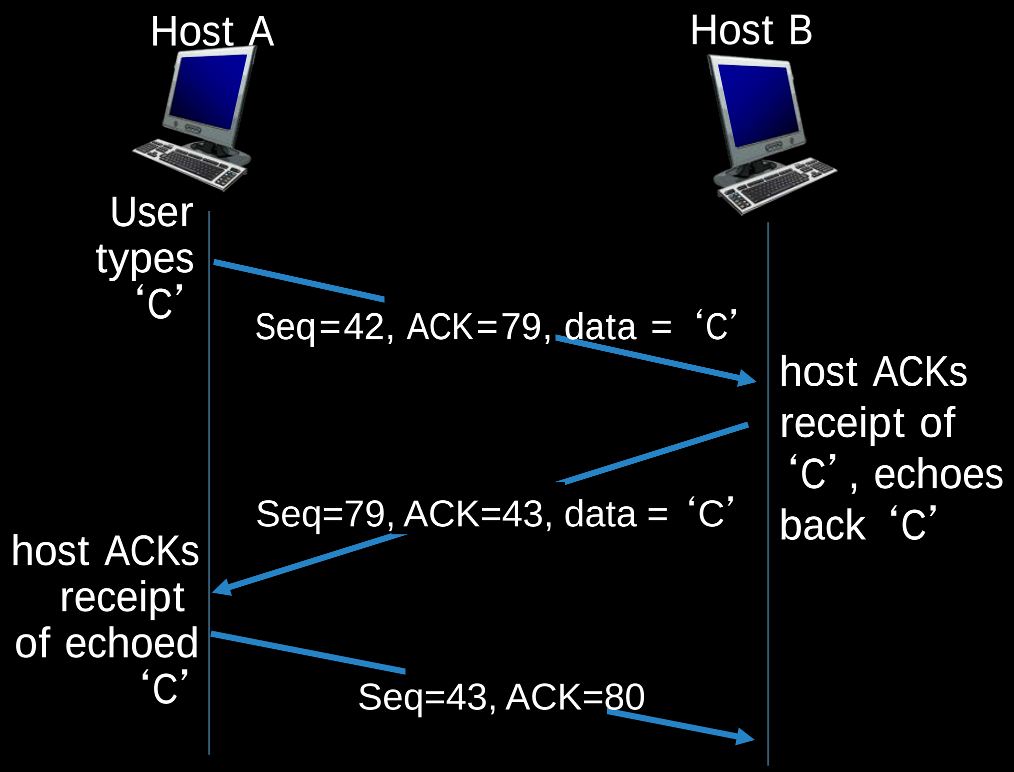
- **Sequence number**
  - byte stream "number" of first byte in segment's data



- **Acknowledgment number**
  - sequence number of the next segment expected by receiver
  - cumulative ACK



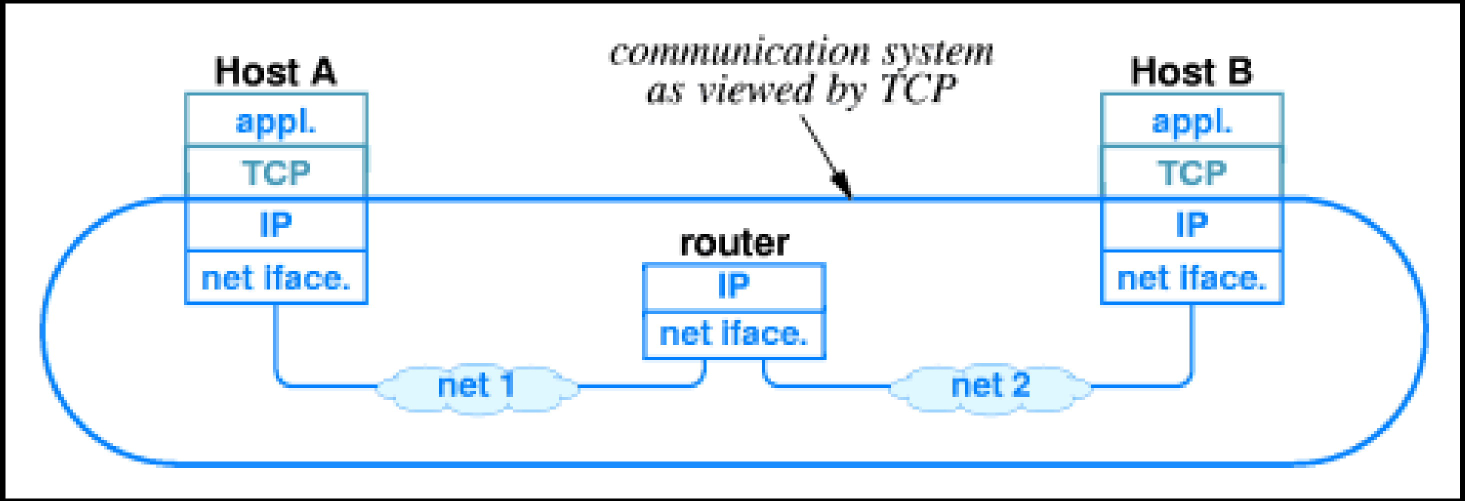
Usage of Seq. & ACK Numbers



simple telnet scenario

TCP Reliable Data Transfer

- TCP creates reliable data transfer service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer



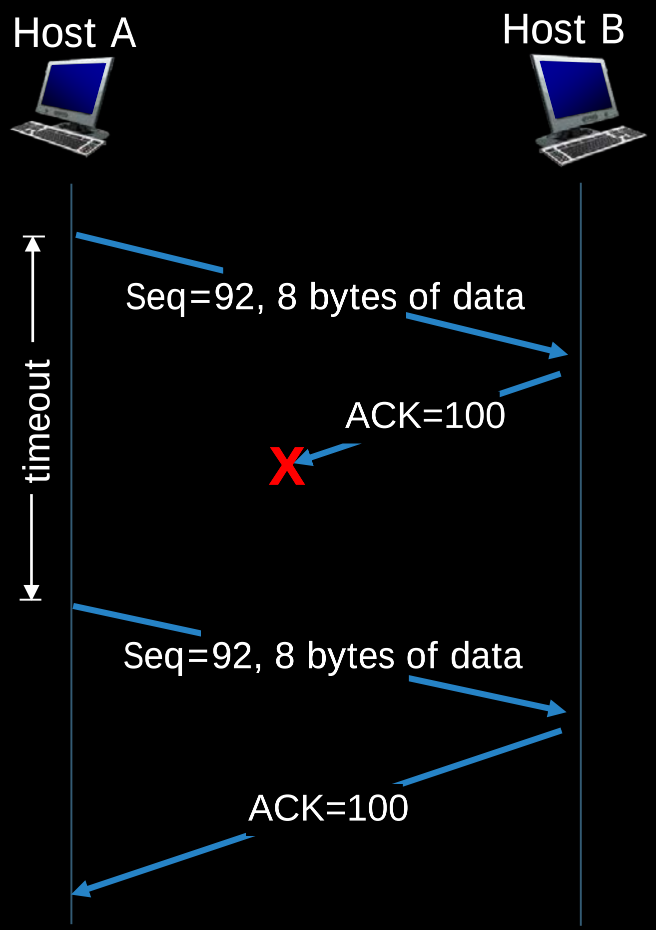
출처 – https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjf9czB0PXbAhUl5rwKHXblBqkQjRx6BAgBEAU&url=https%3A%2F%2Fwww.cs.csustan.edu%2F~john%2Fclasses%2Fprevious_semesters%2FCS3000_Communication_Networks%2F2007_02_Spring%2FNotes%2Fchap25.html&psig=AOvVaw2QofBlqkfTFxG8_J4eyPGl&ust=1530250010414409

TCP Reliable Data Transfer

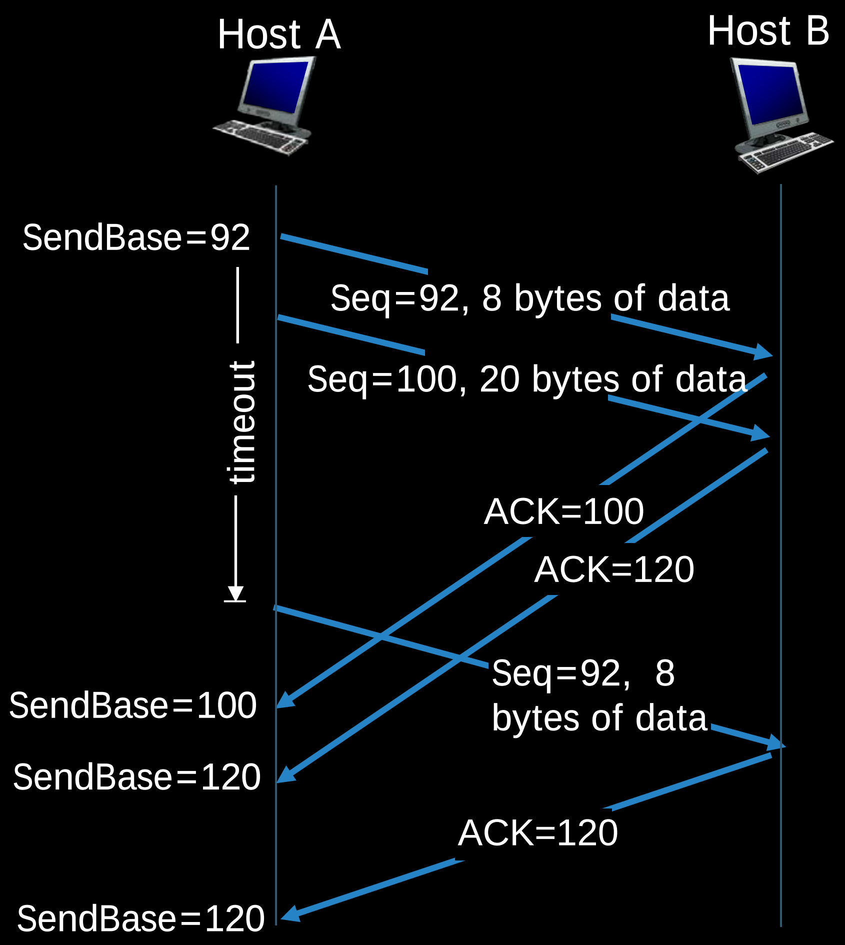
- TCP sender
  - when receiving data from app
    - create segment with seq. number
    - send it and start timer if not already running
  - when timer expires
    - retransmit the segment that caused timeout and restart timer
  - when receiving ack
    - update ACKed packet list
    - start timer if there are still unacked segments

- TCP receiver
  - when receiving data from sender
    - if discover bit error through checksum, then drop the packet
    - checking seq. # and there being no gap, send cumulative ACK
    - if packet is duplicated, drop it
- How receiver handles out-of-order segments?
  - TCP specification does not say explicitly – up to implementer

TCP Retransmission Scenarios (1/2)

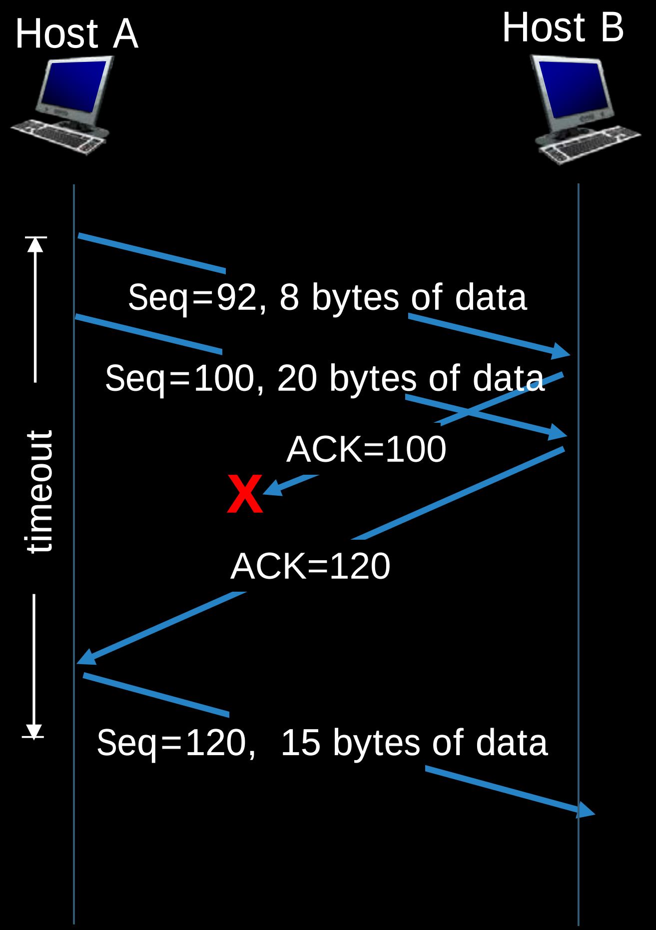


lost ACK scenario



premature timeout

TCP Retransmission Scenarios (2/2)



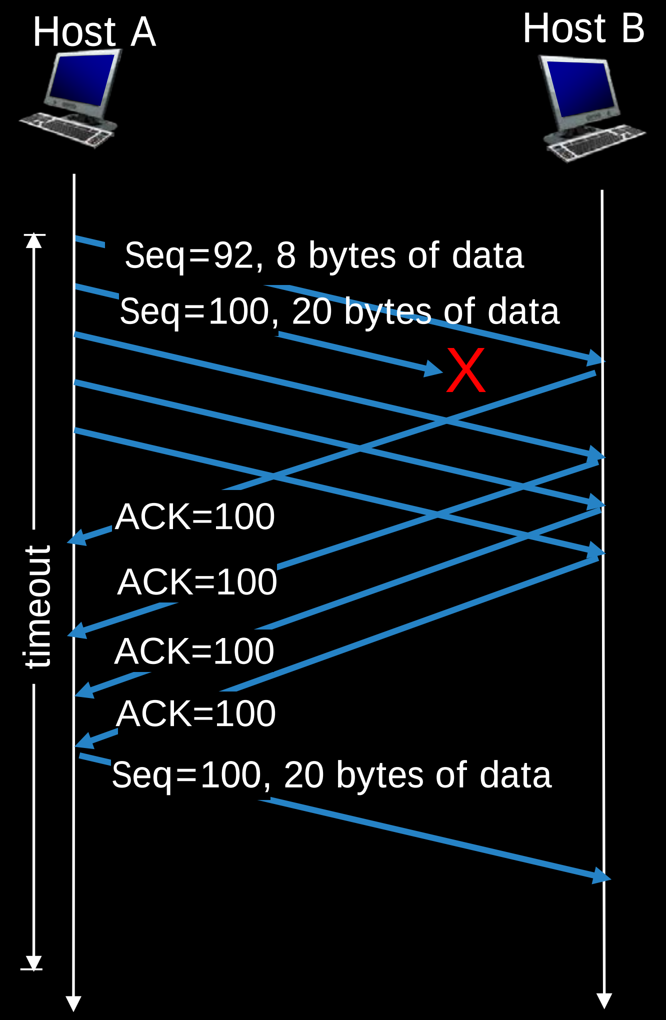
cumulative ACK

- How to set TCP timeout value?
  - must longer than RTT
  - `SampleRTT`
    - measured time from segment transmission until ACK receipt
  - `EstimatedRTT`
    - `SampleRTT` varies
    - moving average several recent measurements

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

TimeoutInterval = EstimatedRTT + "safety margin"

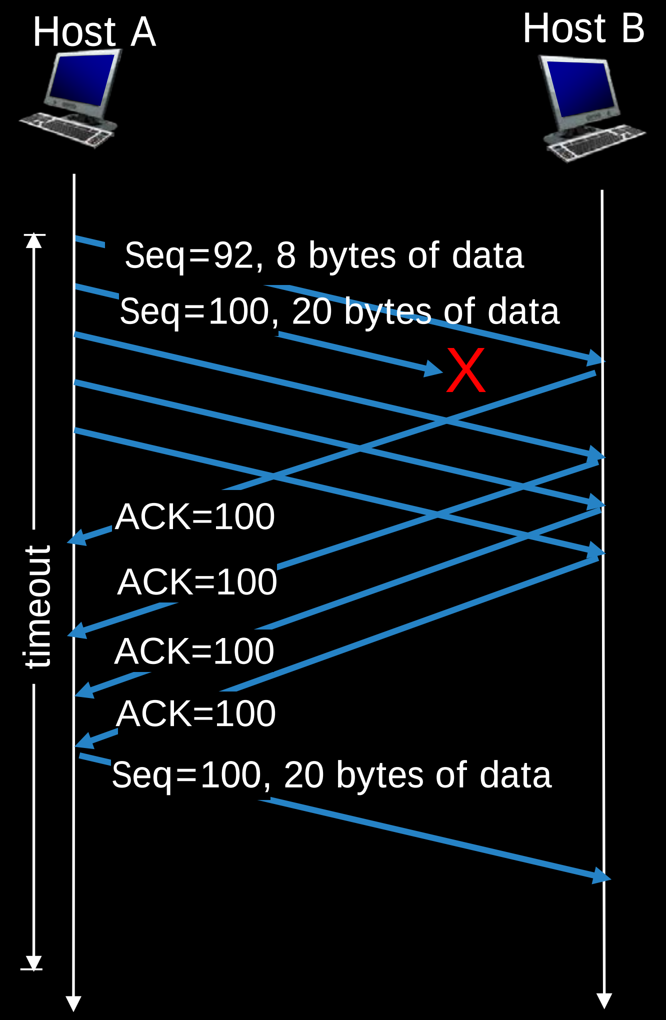
- Timeout period often relatively long:
  - long delay before resending lost packet
- Another way to detect segment lost
  - whenever a segment is received, receiver sends ACK to request the segment that has the next sequence number of the last in-order segment
  - sender often sends many segments back-to-back
  - if a segment is lost, there will likely be many duplicate ACKs



Fast Retransmit

- TCP fast retransmit
  - if sender receives 3 duplicate ACKs (except the first normal ACK) for same data, resend unacked segment with smallest seq. number

- As a result, TCP retransmissions triggered by:
  1) timeout events
  2) duplicate acks



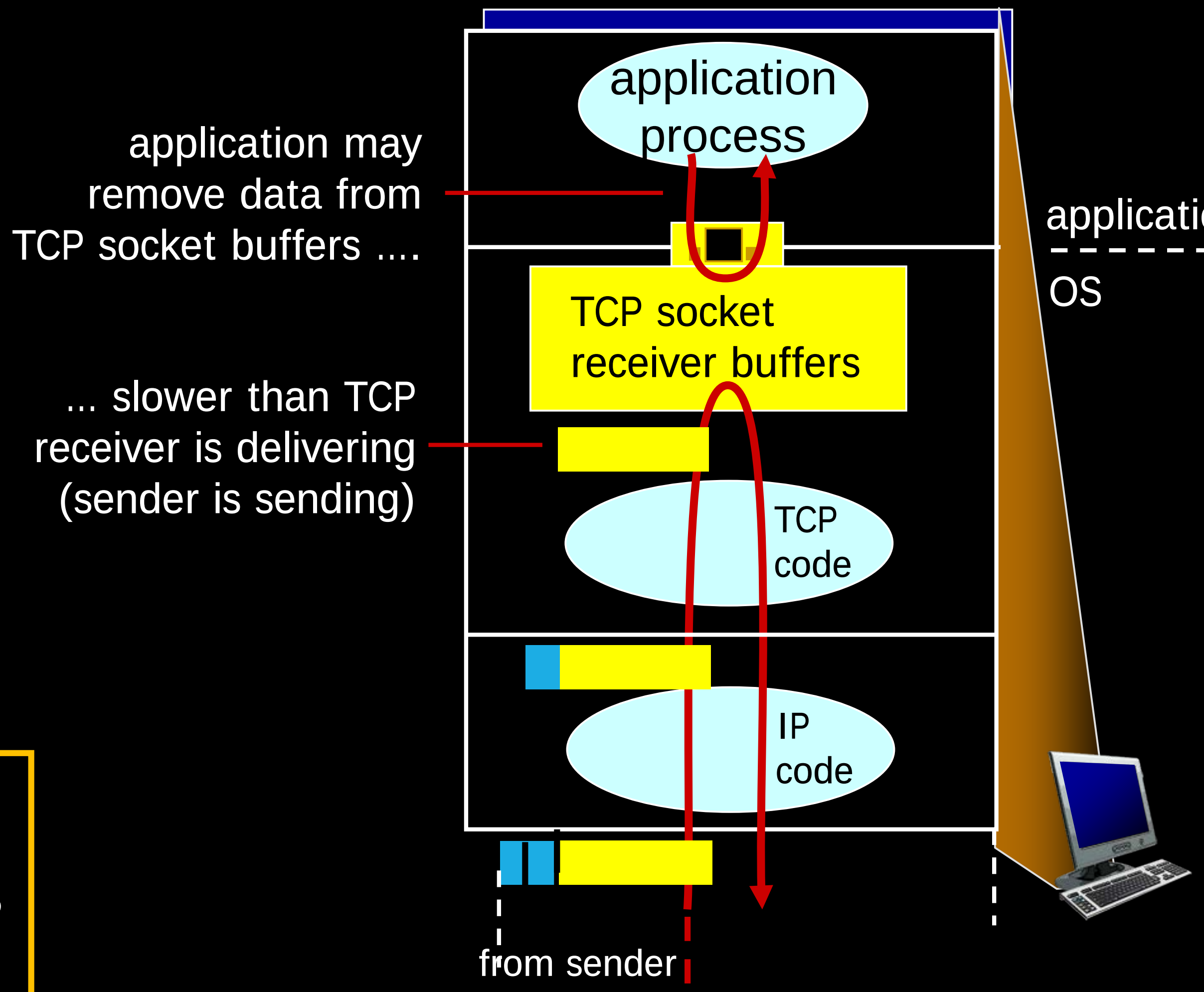
TCP Flow Control

TCP Header

source port #			dest port #
sequence number			
acknowledgement number			
h_len		flags	receive window
checksum			urgent pointer

flow control

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



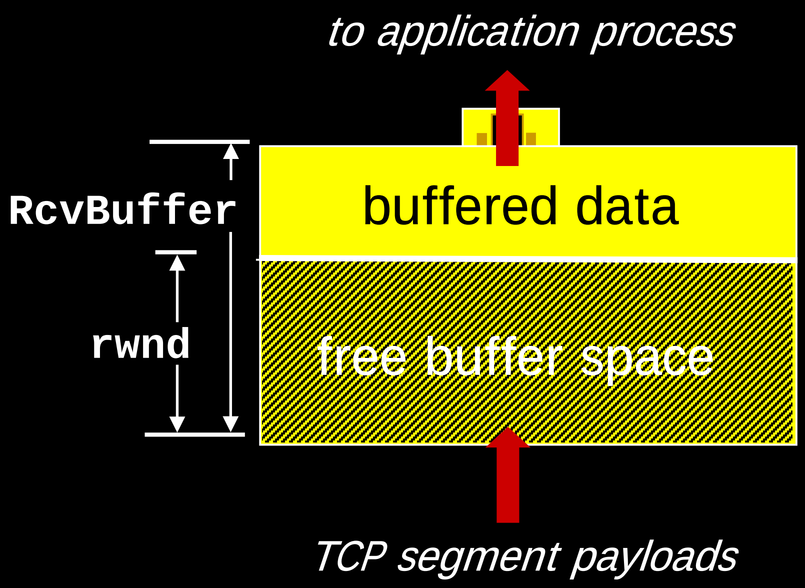
receiver protocol stack

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 OSs autoadjust **`RcvBuffer`**
- Sender limits amount of unacked (“in–flight”) data to receiver’s **`rwnd`** value
- Guarantees receive buffer will not overflow

TCP Header

source port #		dest port #	
sequence number			
acknowledgement number			
h_len		flags	receive window
checksum		urgent pointer	

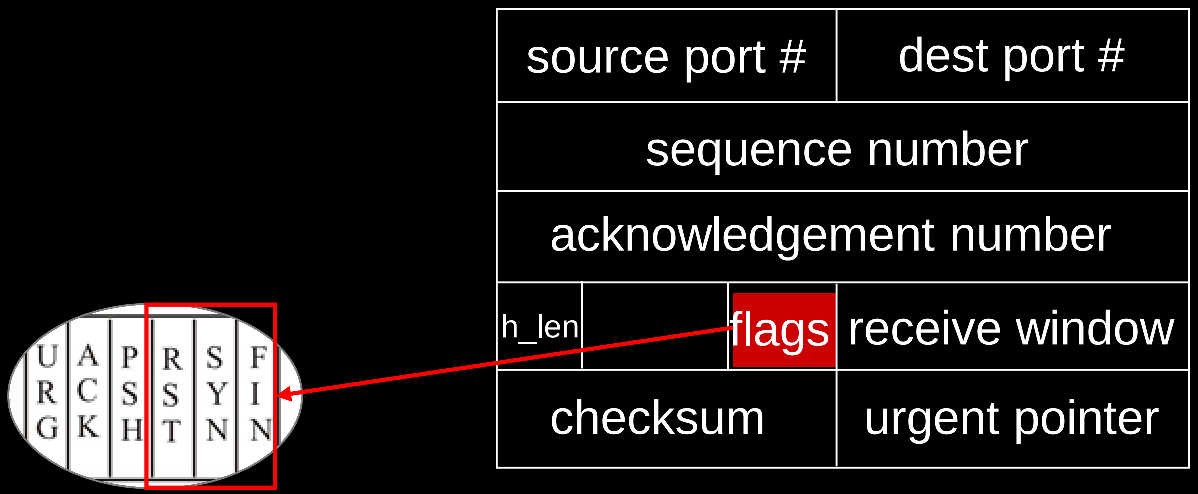


receiver-side buffering

Connection Management

- Before exchanging data, sender and receiver
  - agree to establish connection
  - agree on connection parameters

TCP Header



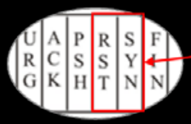
- URG: urgent data (generally not used)
- ACK: ACK # valid
- PSH: push data now (generally not used)

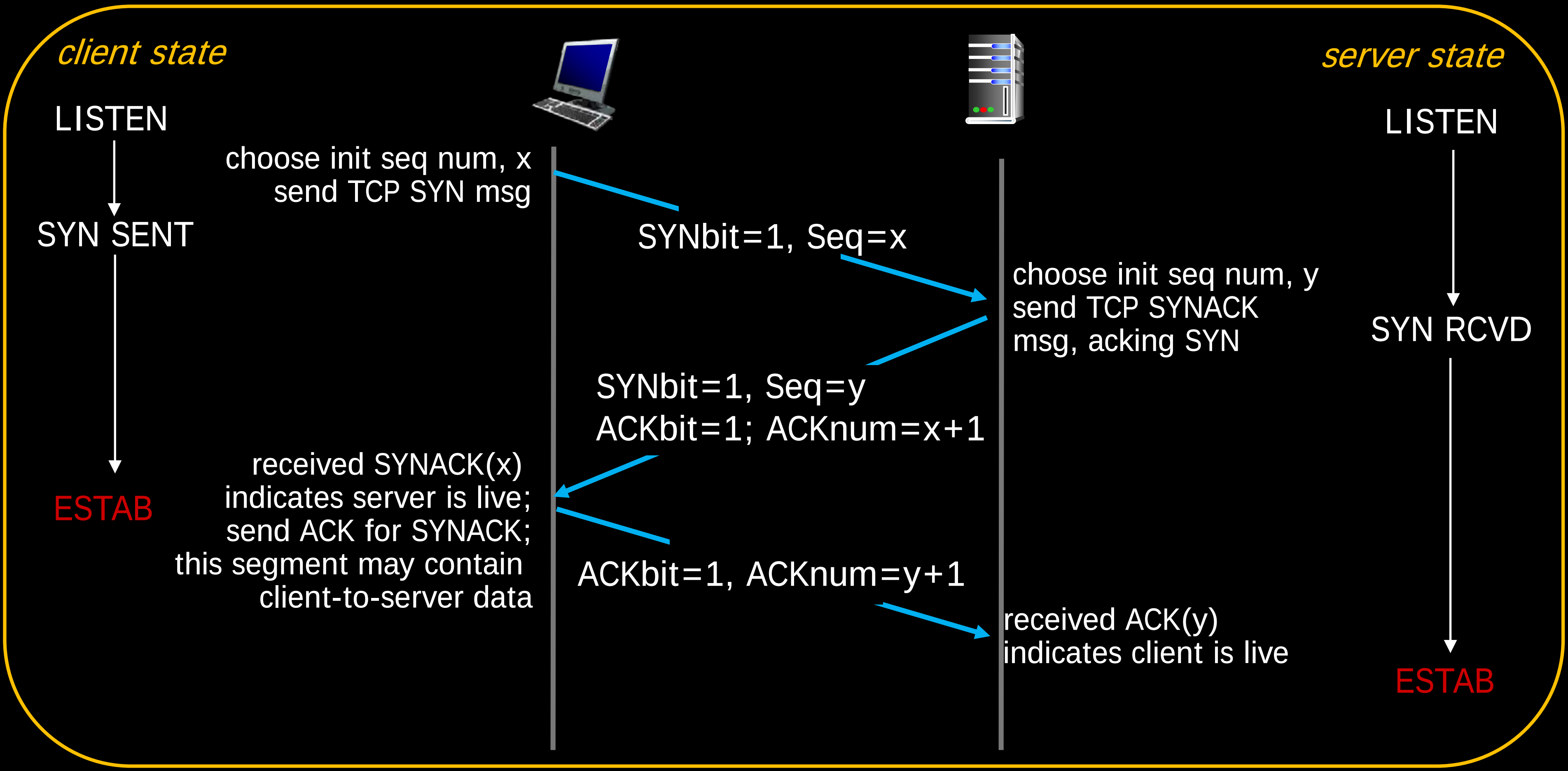
Establishing Connection

- Three–way handshake

TCP Header

source port #	dest port #
sequence number	
acknowledgement number	
h_len flags	receive window
checksum	urgent pointer



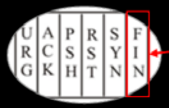


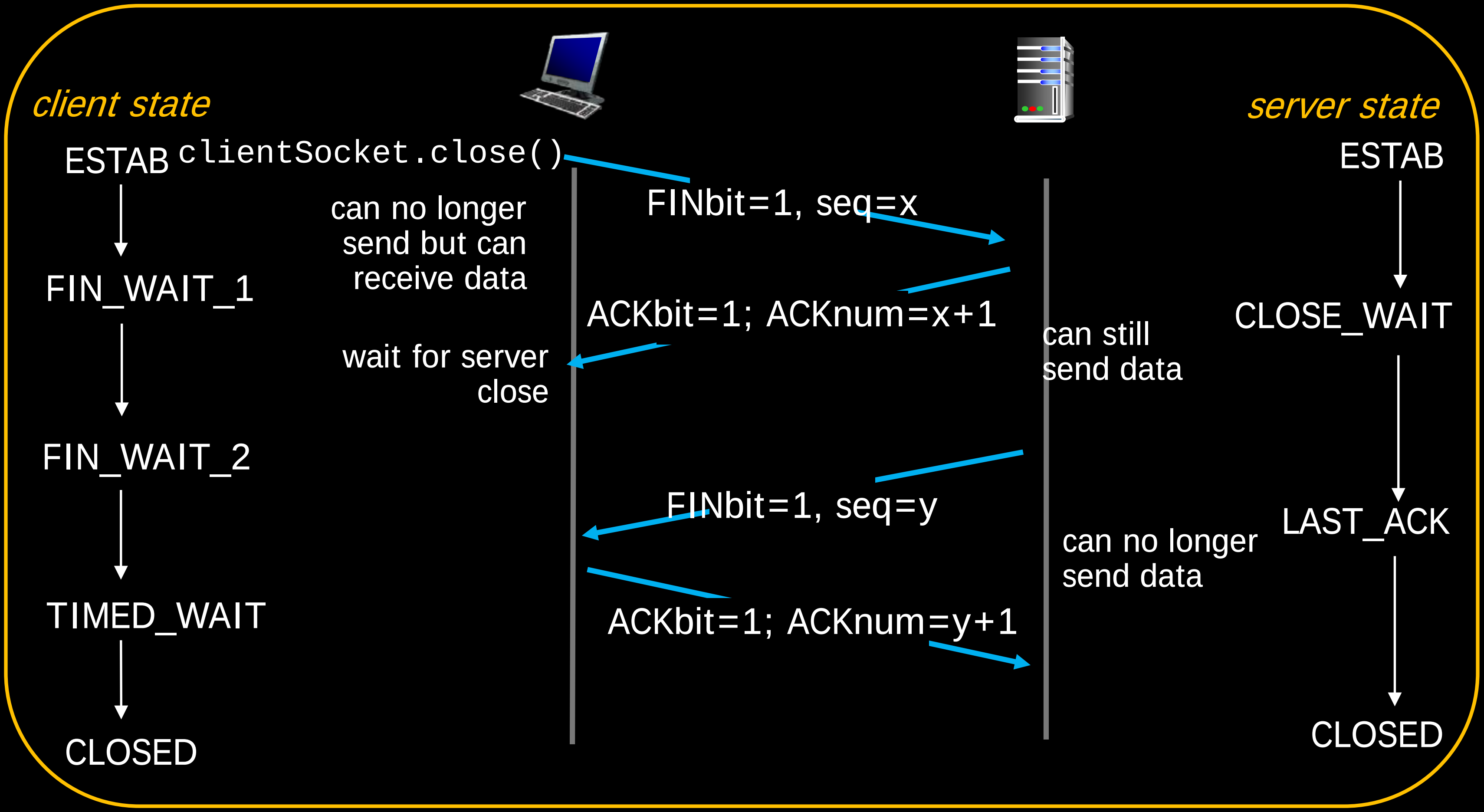
Closing Connection

- Client and 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

TCP Header

source port #	dest port #
sequence number	
acknowledgement number	
h_len flags	receive window
checksum	urgent pointer



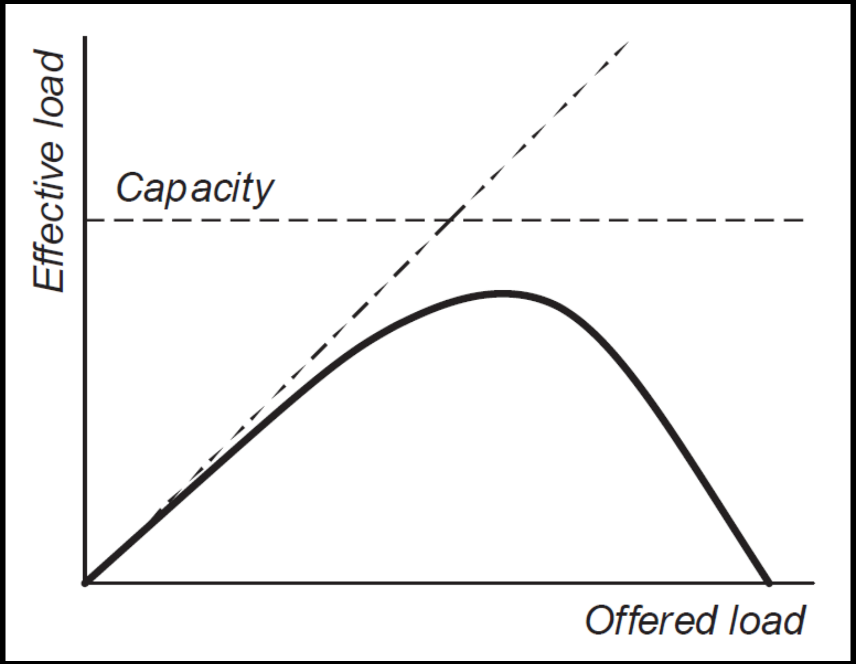


06. Congestion Control

- When the offered load in an distributed sharing system exceeds total system capacity, the effective load will go to zero (collapses) as load increases
  - ➡ Congestion collapse (1980)

Congestion

- "Too many sources sending too much data too fast for *network* to handle"
- Different from flow control!
  - congestion control is global issue
  - flow control is point-to-point issue
- Manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)



출처 – A. Afanasyev et al., "Host-to-Host Congestion Control for TCP," IEEE Comm. Surveys & Tutorials, Vol. 12, No. 3, 3rd Quarter, 2010, pp. 304–342

Congestion Control Approaches

- Two approaches towards congestion control:

end-to-end

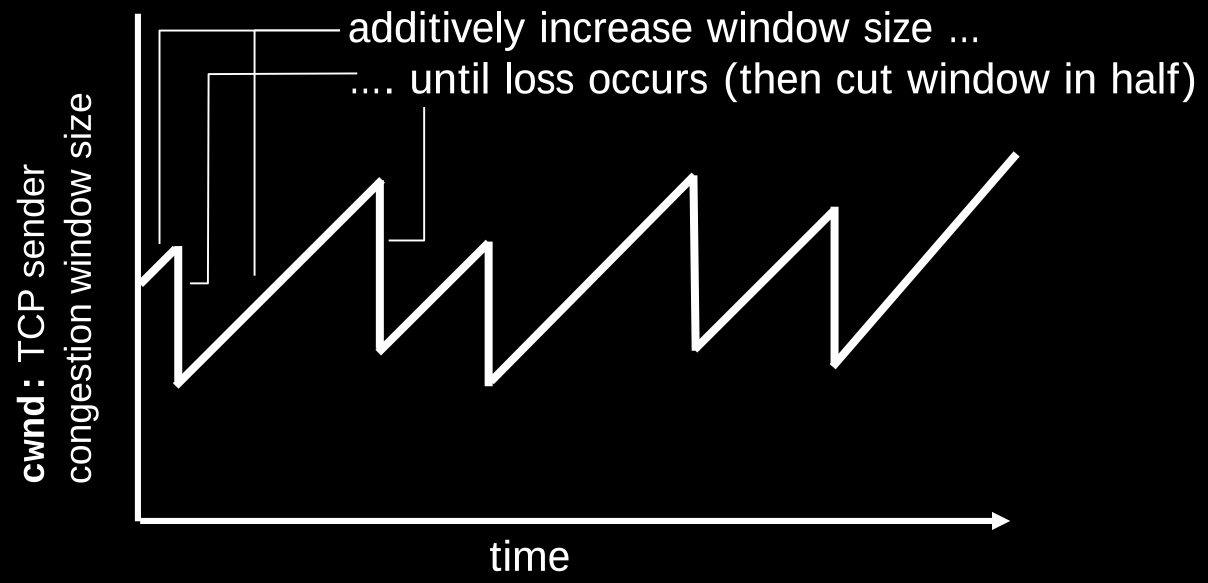
- No explicit feedback from network
- Congestion inferred from end-system observed loss, delay
- Approach taken by TCP

network-assisted

- Routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

- AIMD approach: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - additive increase: increase **cwnd** by 1 MSS every RTT until loss detected
  - multiplicative decrease: cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth



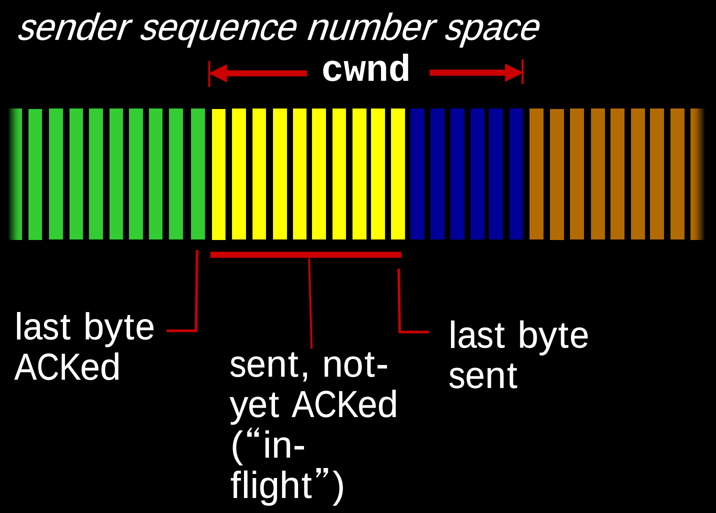
TCP Congestion Control Overview (2/2)

- Congestion window (**cwnd**)
  - the rate a TCP sender can send traffic (the amount of packets in transit)

    `LastByteSent - LastByteAcked ≤ min{cwnd, rwnd}`

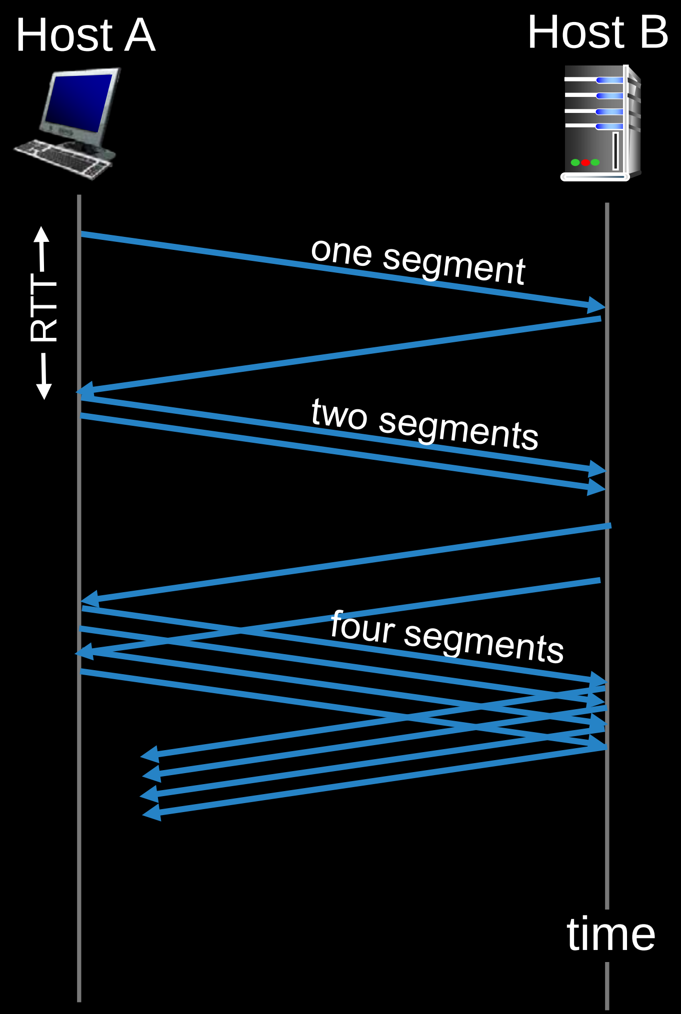
- **cwnd** is dynamic,

  function of perceived network congestion



Changes of **cwnd**: Slow Start

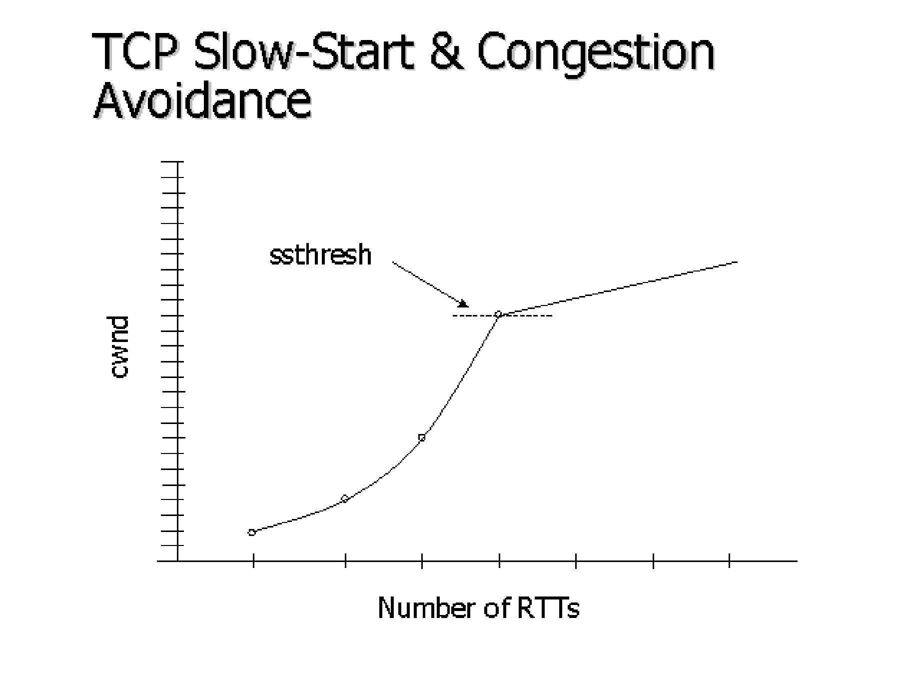
- When connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



Changes of `cwnd`: Congestion Avoidance

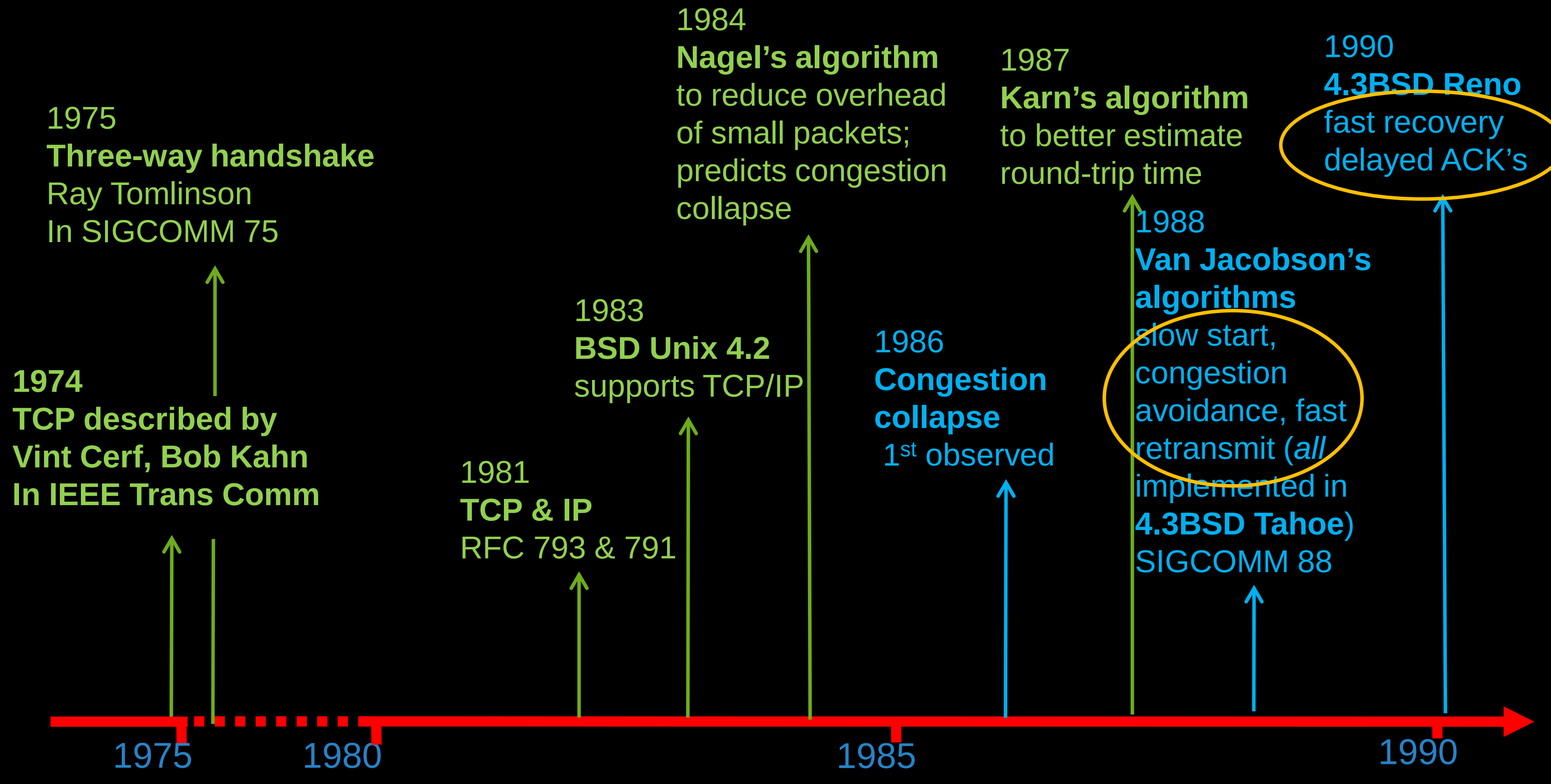
- Q: When should the exponential growth in the slow-start state end?
- A: When **`cwnd`** gets to 1/2 of its value before congestion was detected

- Implementation
  - variable **`ssthresh`**
  - on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event
  - surpassing **`ssthresh`**, **`cwnd`** increases by just a single MSS every RTT



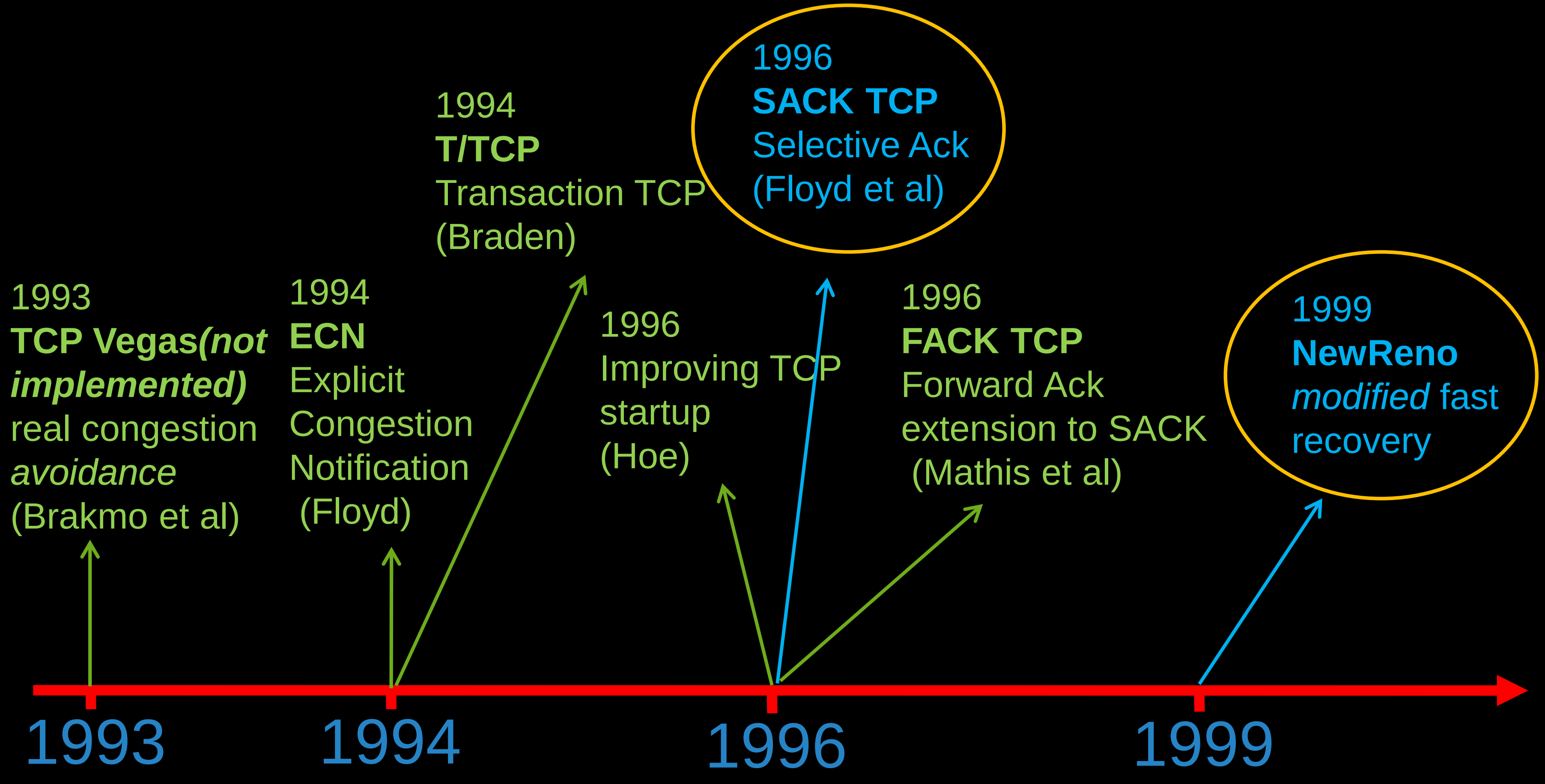
07. TCP Congestion Control Algorithm

Evolution of TCP (1/2)



Lecture note by Prof. Paul D. Amer (Univ. of Delaware)

Evolution of TCP (2/2)



Lecture note by Prof. Paul D. Amer (Univ. of Delaware)

TCP Variants

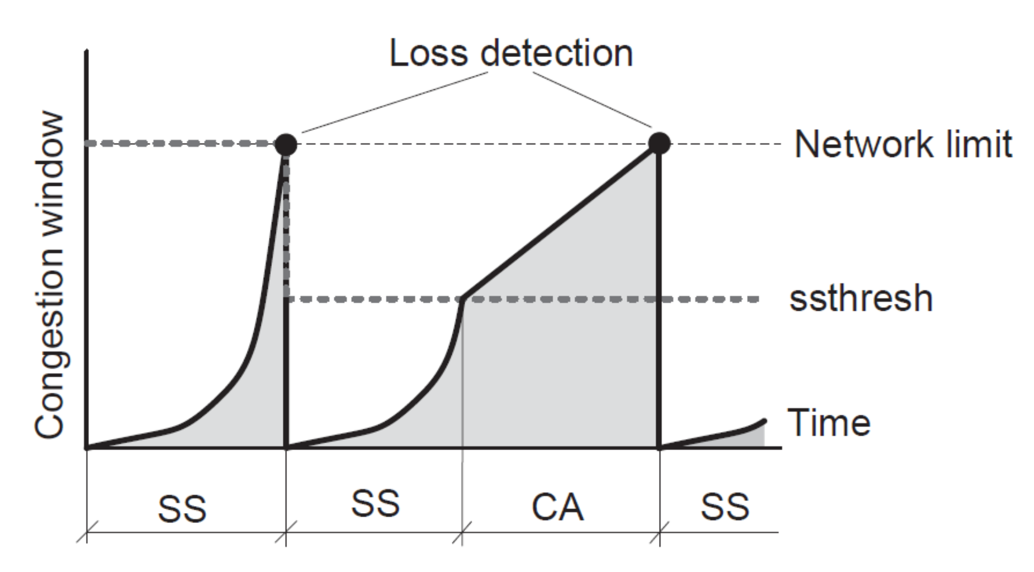
TCP Variant	Section	Year	Base	Added/Changed Modes or Features	Mod[1]	Status	Implementation			
							BSD[2]	Linux	Win	Mac
TCP Tahoe [14]	II-A	1988	RFC793	Slow Start, Congestion Avoidance, Fast Retransmit	S	Obsolete Standard	>4.3	1.0		
TCP-DUAL [15]	II-B	1992	Tahoe	Queuing delay as a supplemental congestion prediction parameter for Congestion Avoidance	S	Experimental				
TCP Reno [16], [17]	II-C	1990	Tahoe	Fast Recovery	S	Standard	>4.3 >F2.2	> 1.3.90	>95/NT	
TCP NewReno [18], [19]	II-D	1999	Reno	Fast Recovery resistant to multiple losses	S	Standard	>F4	> 2.1.36		>10.4.6 (opt)
TCP SACK [20]	II-E	1996	RFC793	Extended information in feedback messages	P+S+R	Standard	>S2.6, >N1.1, >F2.1R	> 2.1.90	> 98	> 10.4.6
TCP FACK [21]	II-F	1996	Reno, SACK	SACK-based loss recovery algorithm	S	Experimental	>N1.1	>2.1.92		
TCP-Vegas [22]	II-G	1995	Reno	Bottleneck buffer utilization as a primary feedback for the Congestion Avoidance and secondary for the Slow Start	S	Experimental		> 2.2.10		
TCP-Vegas+ [23]	II-H	2000	NewReno, Vegas	Reno/Vegas Congestion Avoidance mode switching based of RTT dynamics	S	Experimental				
TCP-Veno [24]	II-I	2002	NewReno, Vegas	Reno-type Congestion Avoidance and Fast Recovery increase/decrease coefficient adaptation based on bottleneck buffer state estimation	S	Experimental		> 2.6.18		
TCP-Vegas A [25]	II-J	2005	Vegas	Adaptive bottleneck buffer state aware Congestion Avoidance	S	Experimental				

[1] TCP specification modification: S = the sender reactions, R = the receiver reactions, P = the protocol specification
[2] S for Sun, F for FreeBSD, N for NetBSD

출처 – A. Afanasyev et al., "Host-to-Host Congestion Control for TCP," IEEE Comm. Surveys & Tutorials, Vol. 12, No. 3, 3rd Quarter, 2010, pp. 304–342

TCP Tahoe (1988)

- Features
  - slow start
  - congestion avoidance
  - fast retransmit
    - retransmission upon 3 duplicate ACKs
- The slow start begins (i.e., **cwnd** = 1 MSS) on
  - timeout
  - fast retransmit



출처 – A. Afanasyev et al., "Host–to–Host Congestion Control for TCP," IEEE Comm. Surveys & Tutorials, Vol. 12, No. 3, 3rd Quarter, 2010, pp. 304–342

TCP Reno (1990)

- Limitation of Tahoe
  - no differentiates losses indicated by timeout and duplicate ACKs
  - always sets **cwnd** to 1 MSS
- Two responses to packet losses by Reno
  - loss indicated by timeout
    - **cwnd** set to 1 MSS
    - slow start to **ssthresh**, then congestion avoidance begins
  - loss indicated by fast retransmit
    - **cwnd** cut in half
    - fast recovery, then congestion avoidance

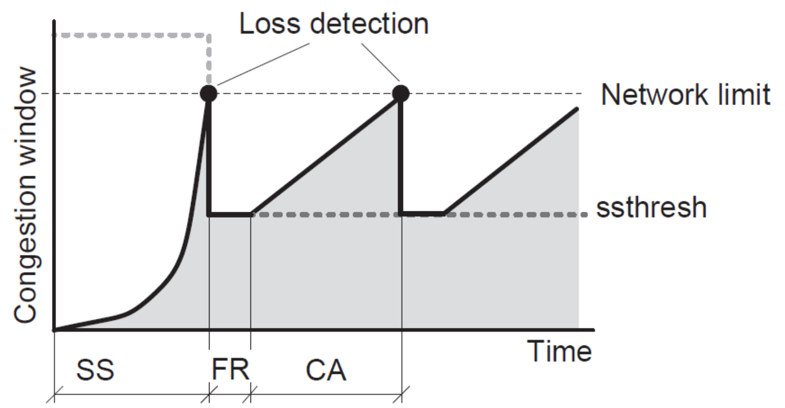
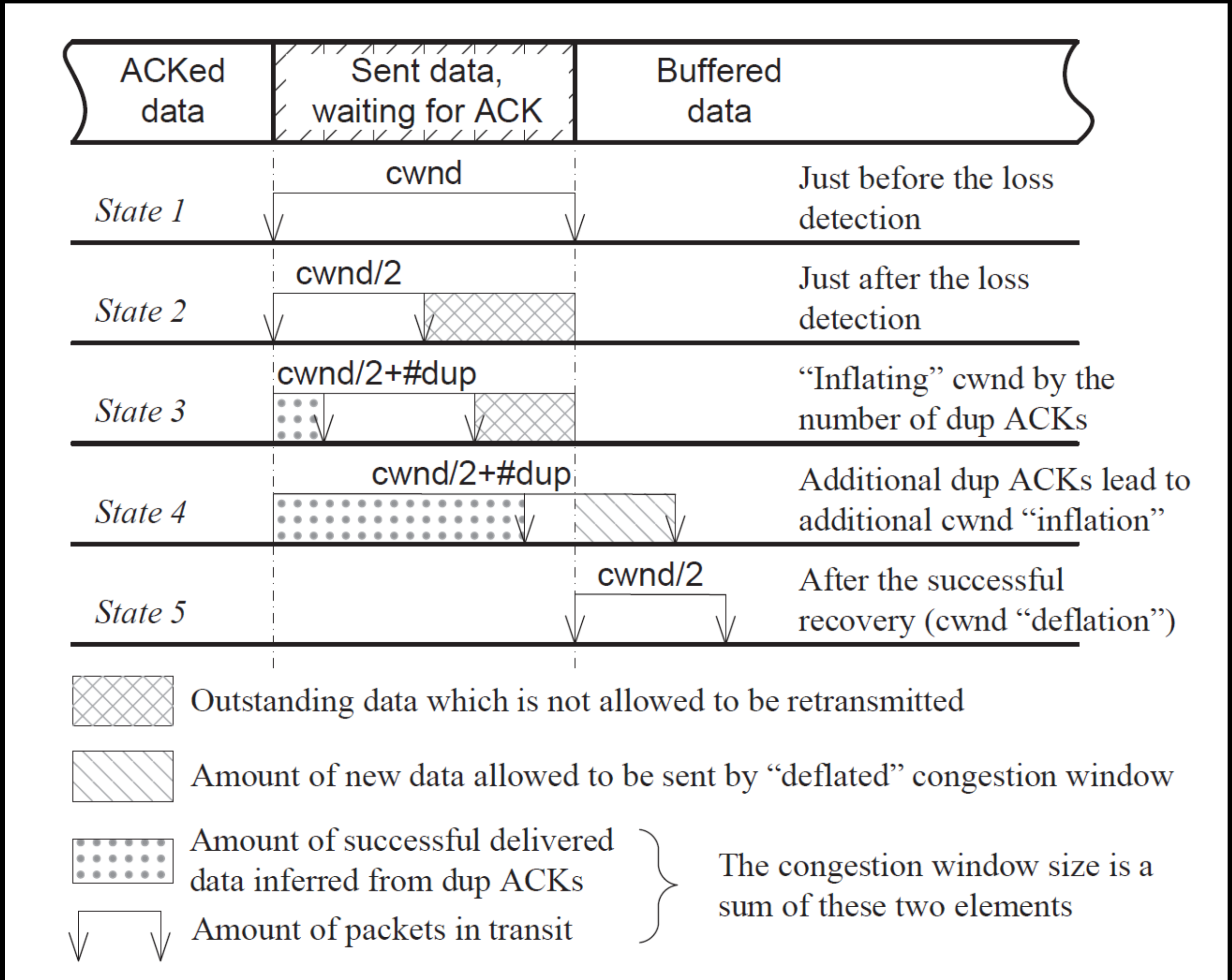


Fig. 15. Congestion window dynamics of TCP Reno (SS: the *Slow Start* phase, CA: the *Congestion Avoidance* phase, FR: the *Fast Recovery* phase)

출처 – A. Afanasyev et al., "Host-to-Host Congestion Control for TCP," IEEE Comm. Surveys & Tutorials, Vol. 12, No. 3, 3rd Quarter, 2010, pp. 304–342

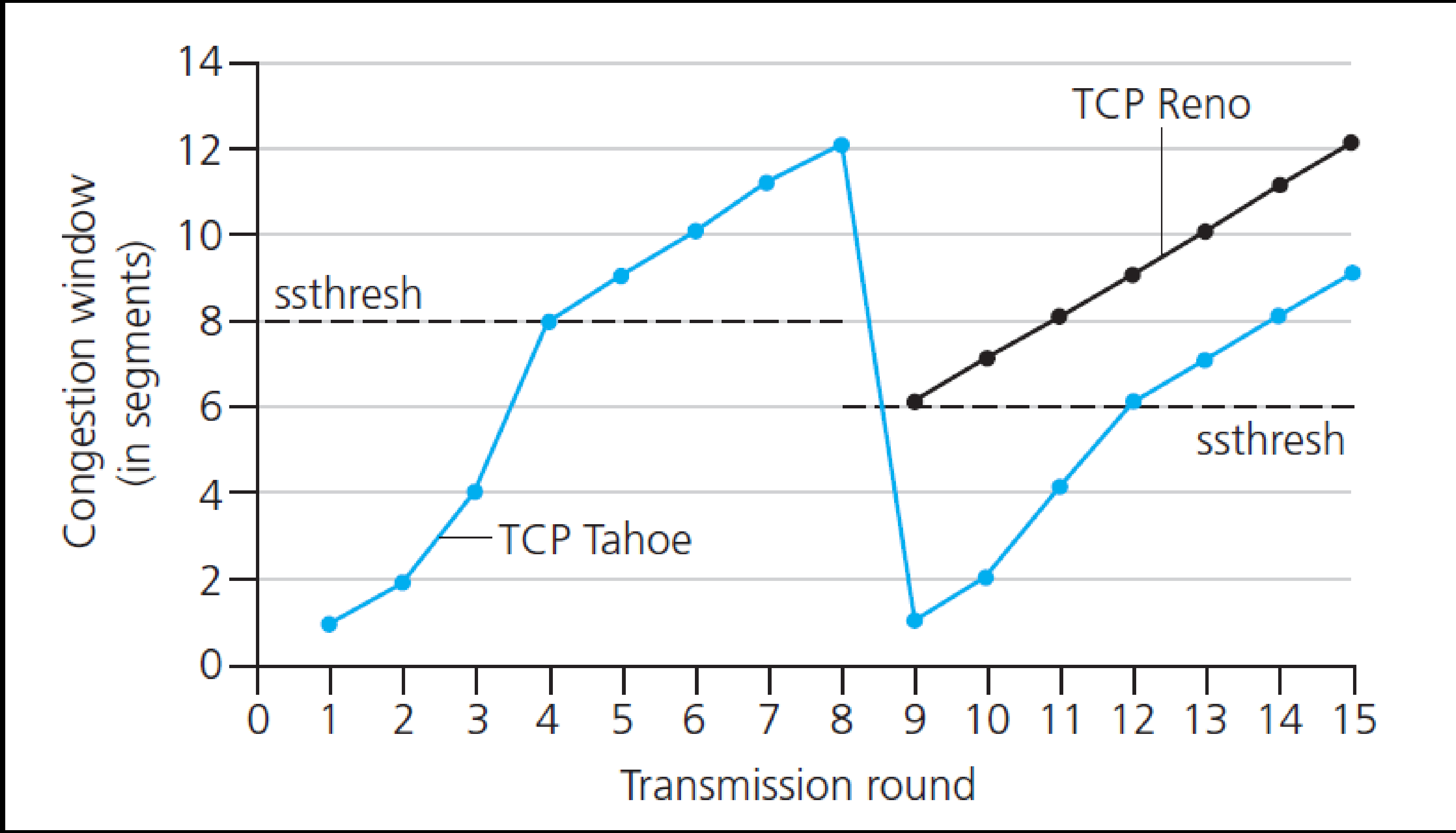
Fast Recovery

- Half **cwnd** maintained until a non-duplicate ACK is received
  - amount of packets in transit kept from decreasing more than expected
- Operation
  - Upon the event of 3 duplicate ACKs,
  - after reducing **cwnd** to 1/2,
  - the algorithm not only retransmits the oldest unacknowledged packet,
  - but also inflates the congestion window by the number of duplicate packets.
  - A non-duplicate ACK makes the congestion avoidance start.



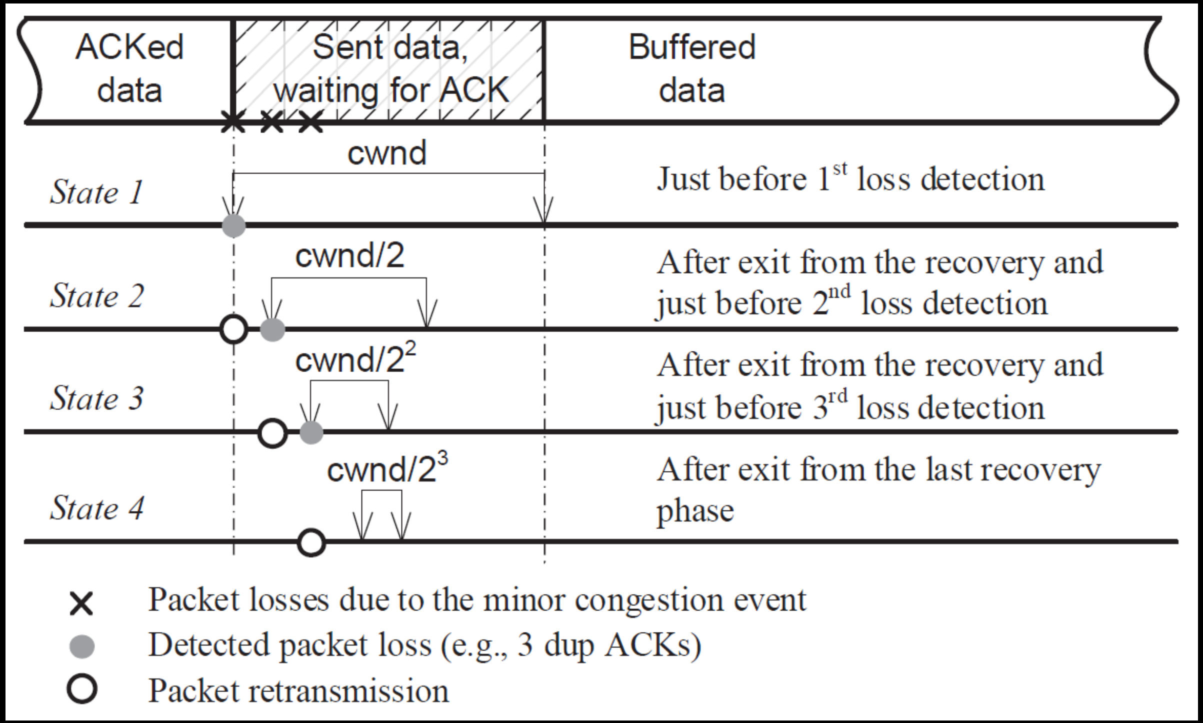
출처 – A. Afanasyev et al., "Host-to-Host Congestion Control for TCP," IEEE Comm. Surveys & Tutorials, Vol. 12, No. 3, 3rd Quarter, 2010, pp. 304–342

TCP Tahoe vs. TCP Reno



TCP NewReno (1999)

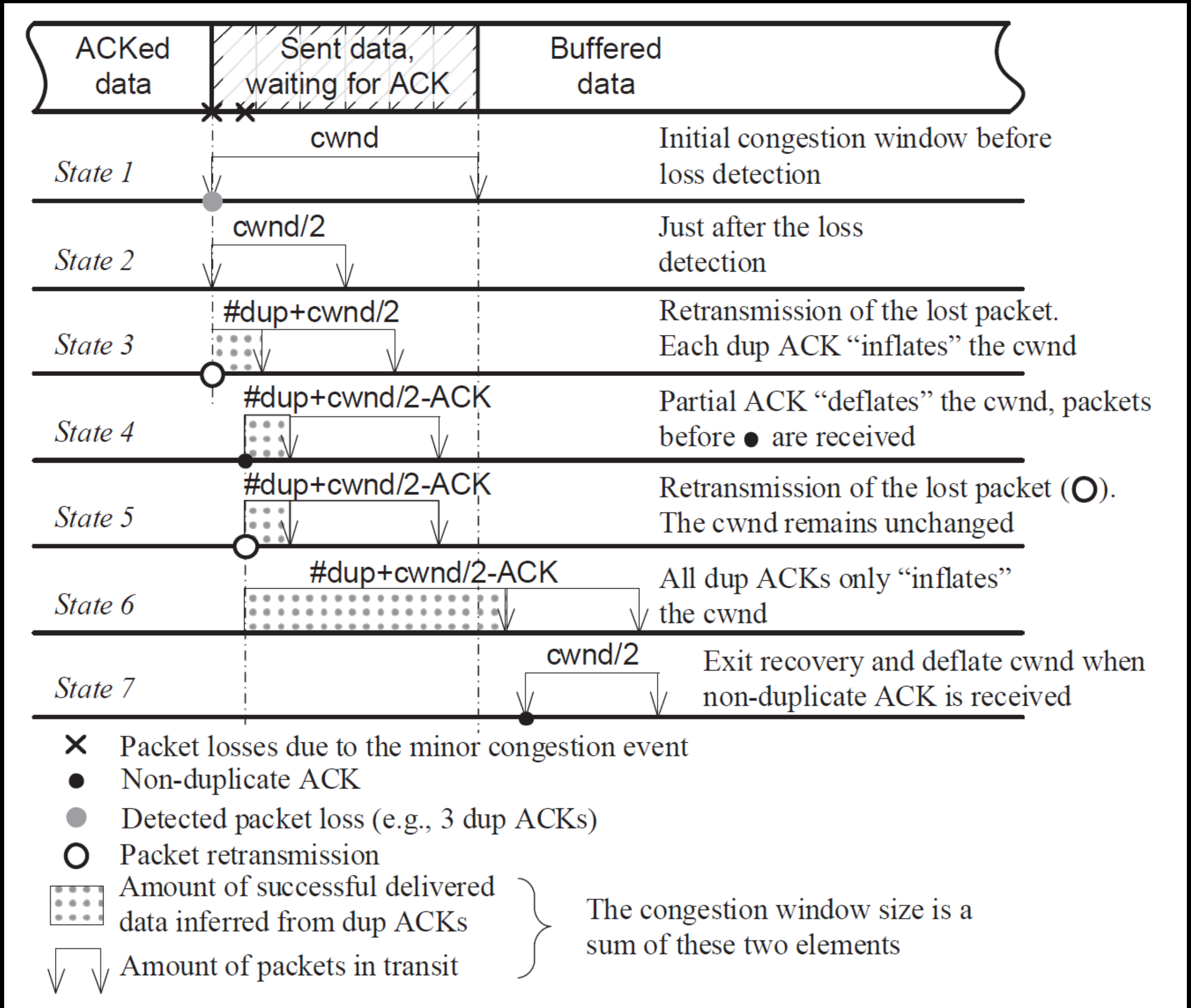
- Vulnerability of Reno's fast recovery
  - in case of multiple packet losses, **cwnd** may be reduced too much (exponential way)



출처 – A. Afanasyev et al., "Host-to-Host Congestion Control for TCP," IEEE Comm. Surveys & Tutorials, Vol. 12, No. 3, 3rd Quarter, 2010, pp. 304–342

TCP NewReno (1999)

- Refinement of Reno's fast recovery
  - resistant to multiple losses
  - restricts the exit from the recovery phase until all data packets from the initial congestion window are acknowledged
  - exits from the NewReno's fast recovery only when a new data ACK is received
  - keeps the sequence number of the last data packet sent before entering the fast recovery state



출처 – A. Afanasyev et al., "Host-to-Host Congestion Control for TCP," IEEE Comm. Surveys & Tutorials, Vol. 12, No. 3, 3rd Quarter, 2010, pp. 304–342

Tahoe → Reno → NewReno

Tahoe (1988)

- Slow start
- Congestion avoidance
- Fast retransmit

Reno (1990)

- Slow start
- Congestion avoidance
- Fast retransmit
- Fast recovery in case of fast retransmit

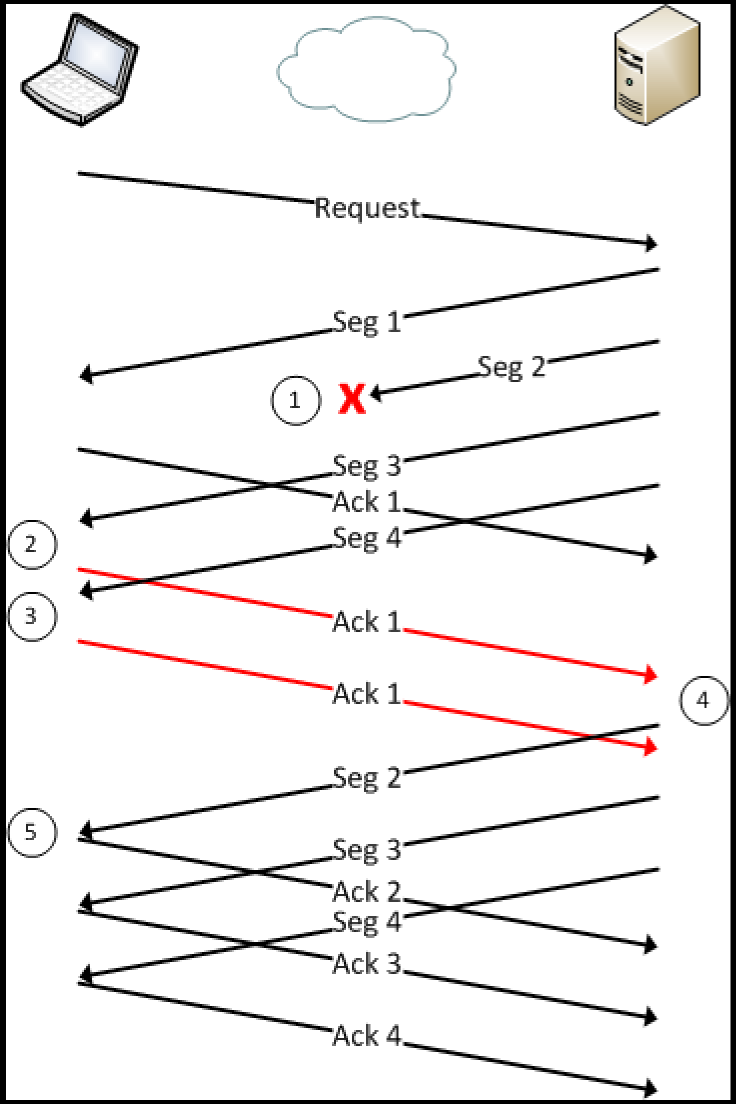
NewReno (1999)

- Slow start
- Congestion avoidance
- Fast retransmit
- Modified fast recovery for multiple packet losses

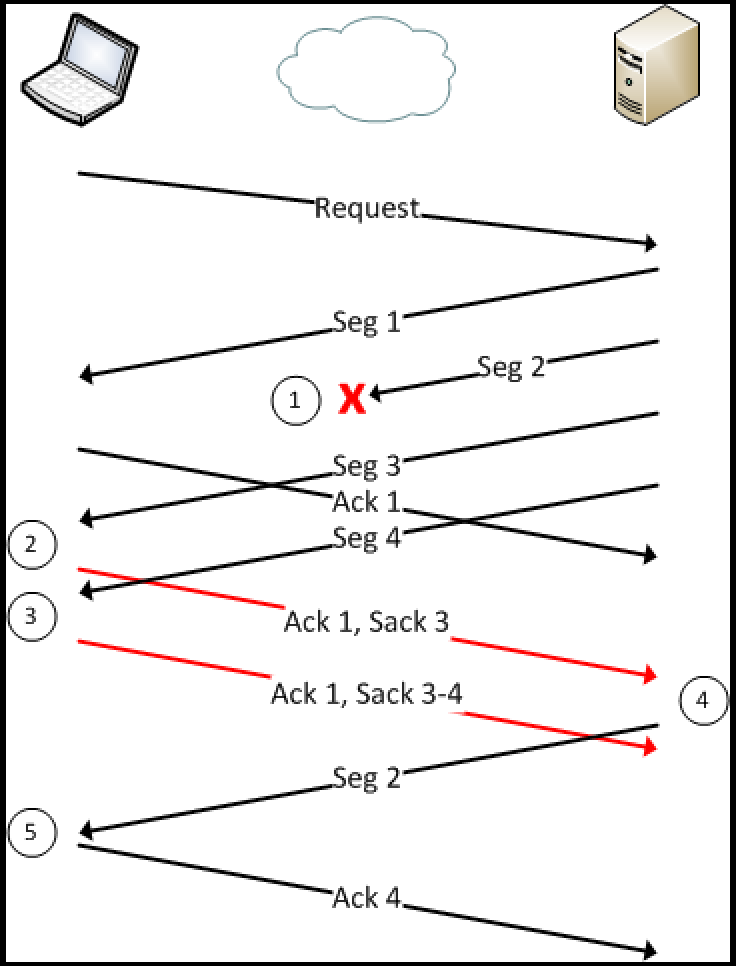
TCP SACK (1996)

- Limited information of cumulative ACKs
  - ACK of only the last in–order packet
  - Reno's fast recovery assumes loss of only one data packet
  - a duration of the recovery is directly proportional to the number of packet losses
- SACK (Selective ACK) option
  - receiver provides information about several packet losses in a single ACK message by reporting blocks of successfully delivered data packets

TCP SACK (1996)



Cumulative ACK

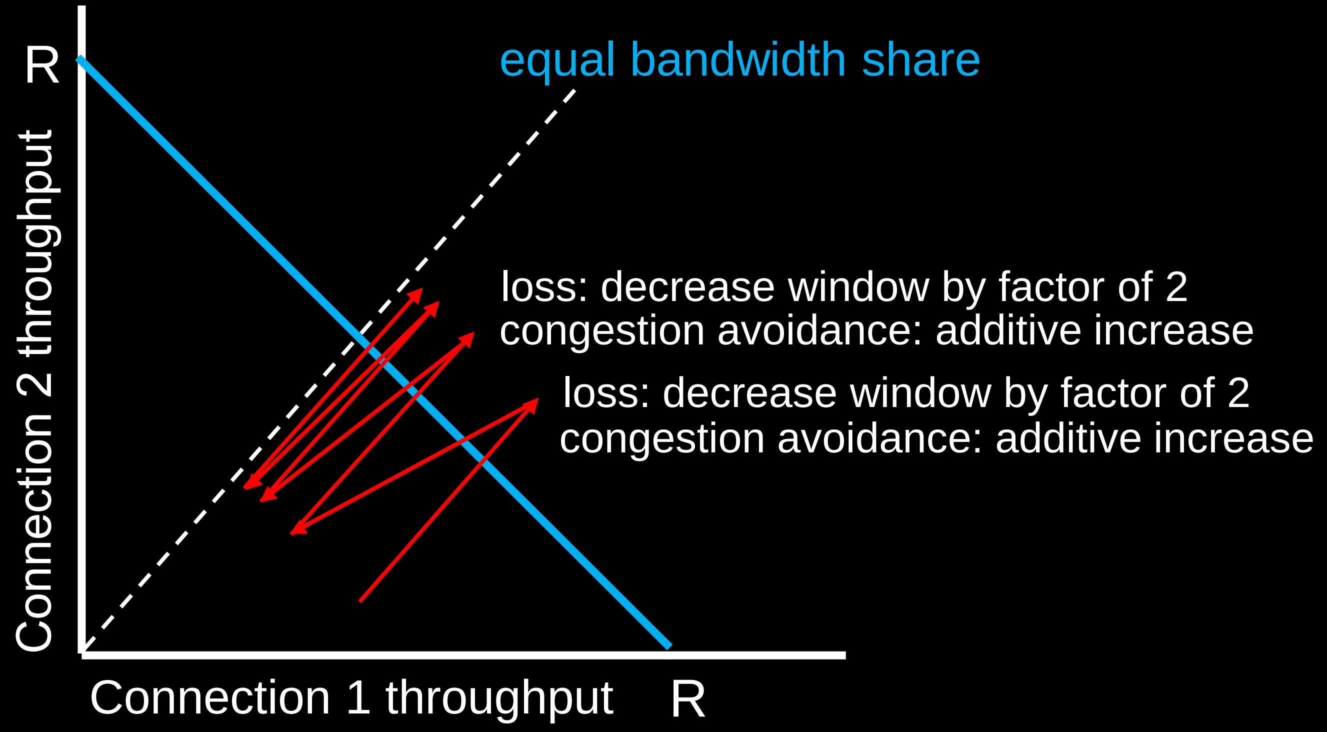


Selective ACK

08. TCP vs. UDP

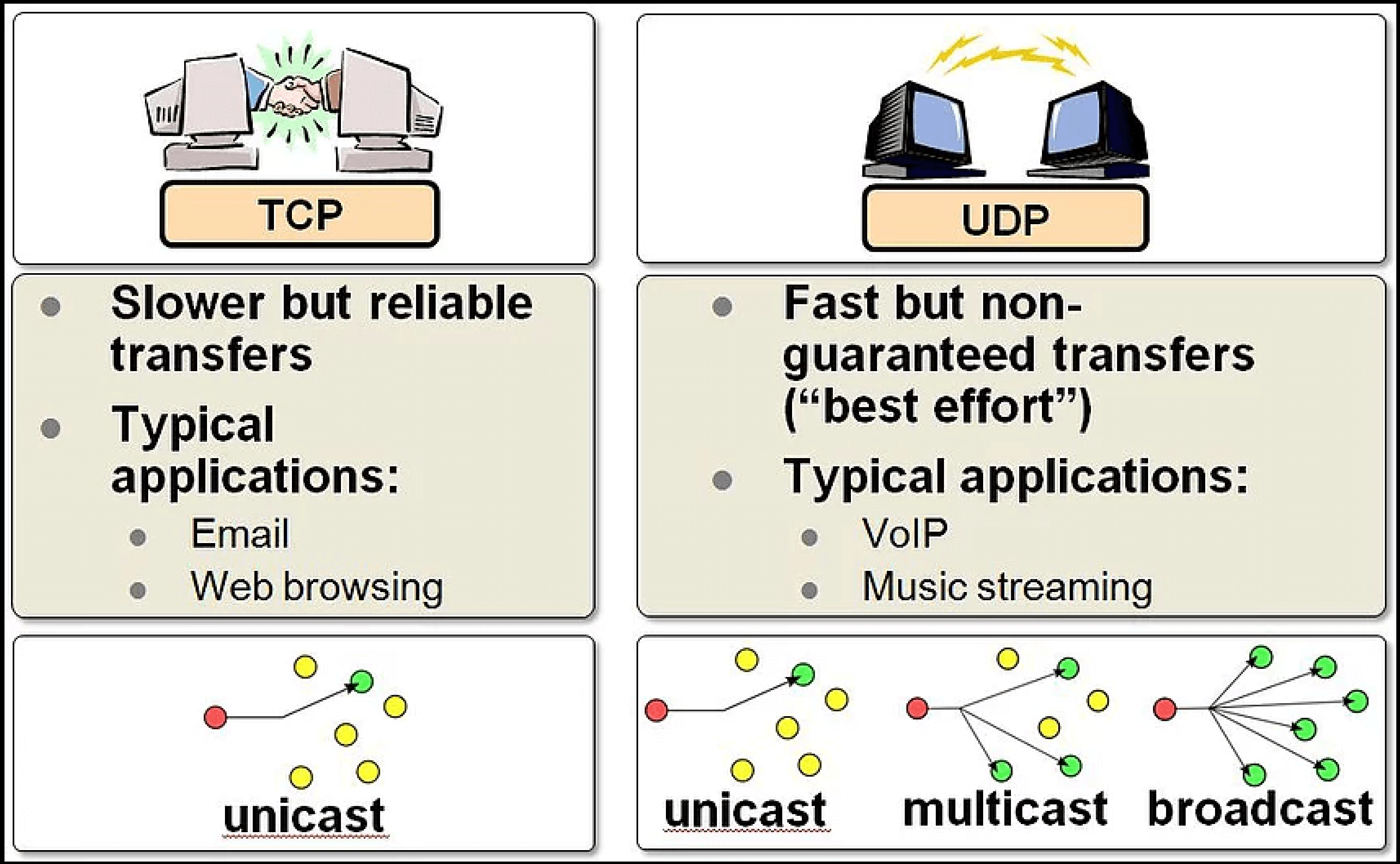
TCP vs. UDP

- TCP fairness through AIMD (additive increase and multiplicative decrease)
  - two competing sessions



- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP
  - send audio/video at constant rate, tolerate packet loss

TCP vs. UDP



출처 –
https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwju_8aR0PXbAhVGf7wKHfEmAYUQjRx6BAgBEAU&url=https%3A%2F%2Fknowledgeofthings.com%2Ftcpip-vs-udp-internet-protocol-suite%2F&psig=AOvVaw2QofBlqkfTFxG8_J4eyPGl&ust=1530250010414409/

Summary

Summary (1/2)

01 Transport layer services
- program, process, thread
- logical communication between processes

02 Multiplexing and socket
- multiplexing and demultiplexing
- socket and socket program

03 User datagram protocol
- connectionless service
- fast but unreliable transfer

04 Reliable data transfer principles
- handling of bit error and packet loss
- ARQ: stop-and-wait, go-back-N, selective-repeat

Summary (2/2)

05	**Transmission control protocol** ▪ connection–oriented service ▪ reliable data transfer through error control and flow control
06	**Congestion control** ▪ congestion: "Load is higher than network capacity!" ▪ additive increase and multiplicative decrease
07	**TCP congestion control algorithm** ▪ dynamics of congestion window ▪ TCP Tahoe, Reno, NewReno, SACK
08	**TCP vs. UDP** ▪ TCP: slower but reliable and fair ▪ UDP: fast but unguaranteed