

Netzwerktechnik und IT-Netze

Chapter 3: Transport Layer

Vorlesung im WS 2016/2017

Bachelor Informatik

(3. Semester)

Prof. Dr. rer. nat. Andreas Berl

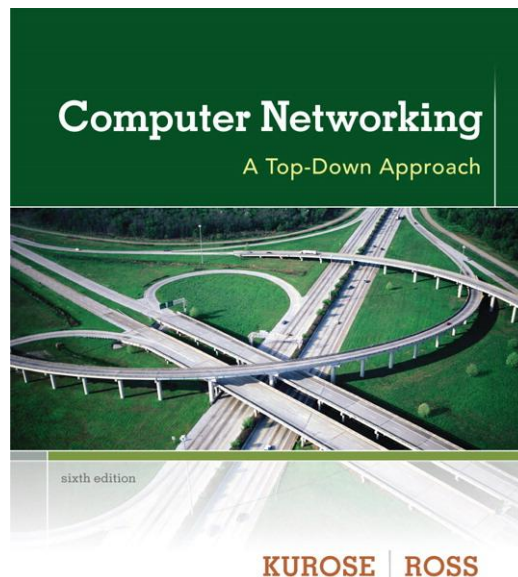
Fakultät für Elektrotechnik, Medientechnik und Informatik

Overview

- Introduction
- Computer Networks and the Internet
- Application Layer
 - WWW, Email, DNS, and more
 - Socket programming
 - Web service
- **Transport Layer**
- Network Layer
- Link Layer
- Wireless and Mobile Networking
- P2P Networks

Introduction

- A note on the use of these power point slides:
 - All material copyright 1996-2012© J.F Kurose and K.W. Ross, All Rights Reserved
 - Do not copy or distribute this slide set!



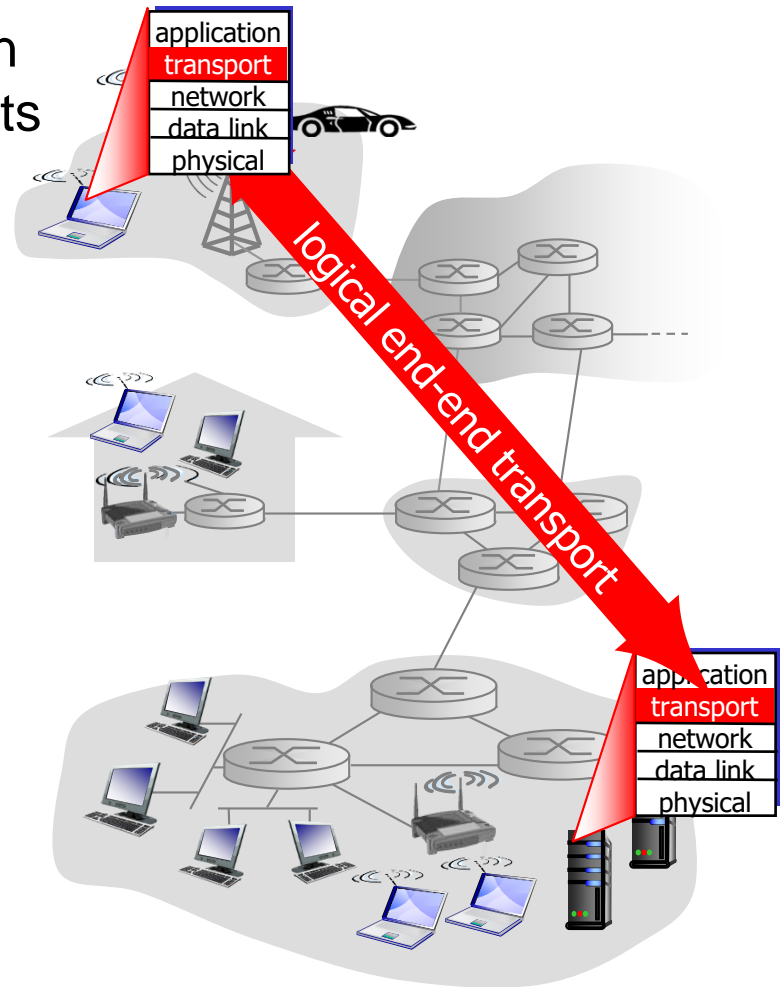
*Computer
Networking: A Top
Down Approach*
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

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

Transport services and protocols

- Provide logical communication between app processes running on different hosts
- Transport protocols run in end systems
 - Send side: breaks app messages into segments, passes to network layer
 - RCV side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
- Internet: TCP and UDP

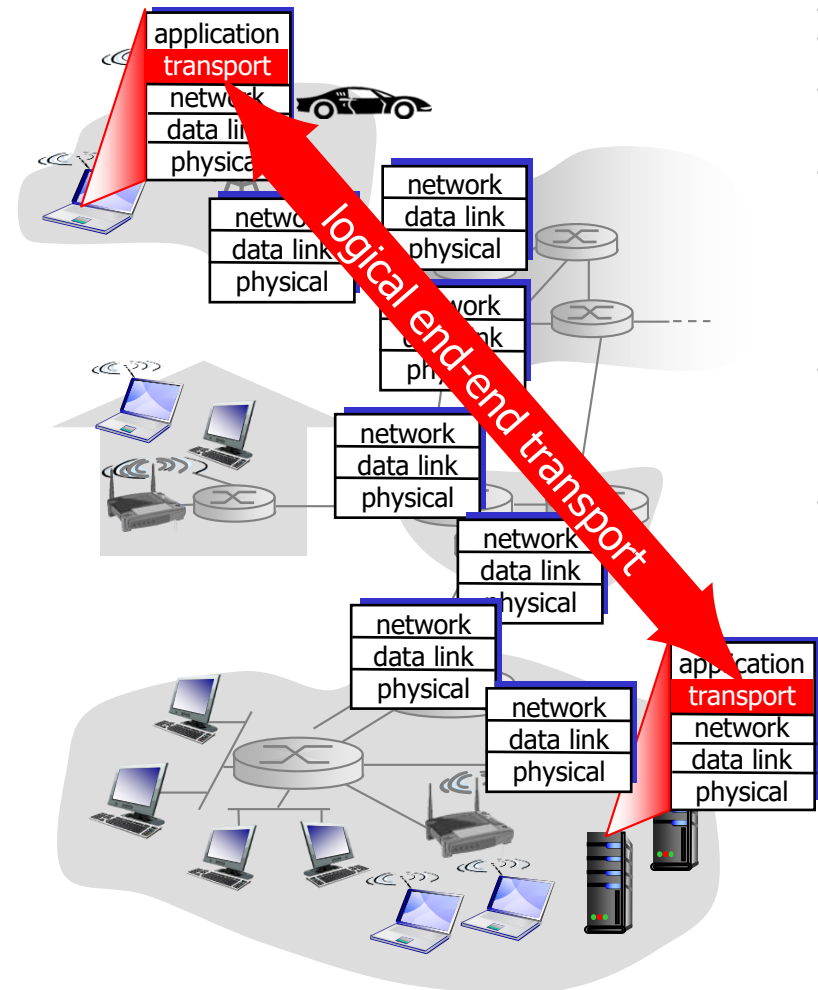


Transport vs. network layer

- Network layer:
 - Logical communication between hosts
- Transport layer:
 - Logical communication between processes
 - Relies on, enhances, network layer services
- Household analogy
 - 12 kids in Ann's house sending letters to 12 kids in Bill's house:
 - Hosts = houses
 - Processes = kids
 - App messages = letters in envelopes
 - Transport protocol = Ann and Bill who demux to in-house siblings
 - Network-layer protocol = postal service

Internet transport-layer protocols

- Reliable, in-order delivery (TCP)
 - Congestion control
 - Flow control
 - Connection setup
- Unreliable, unordered delivery: UDP
 - No-frills extension of “best-effort” IP
- Services not available:
 - Delay guarantees
 - Bandwidth guarantees

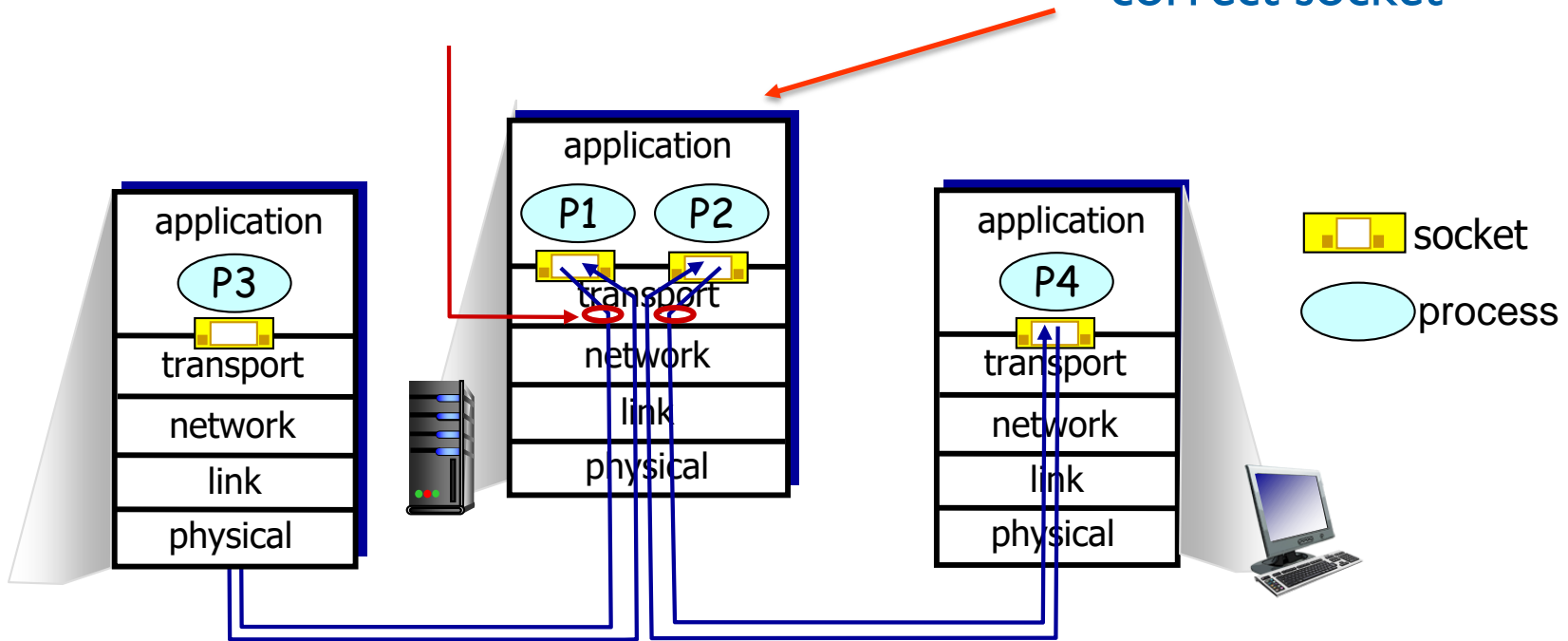


Chapter 3: Roadmap

- 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

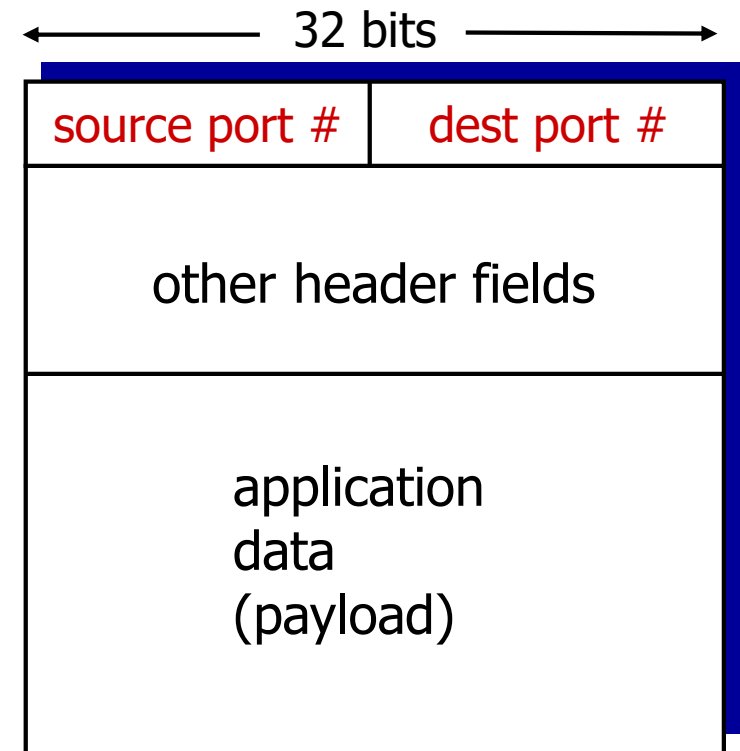
Multiplexing/Demultiplexing

- Multiplexing at sender:
 - Handle data from multiple sockets, add transport header (later used for demultiplexing)
- Demultiplexing at receiver:
 - Use header info to deliver received segments to correct socket



How demultiplexing works

- Host receives IP datagrams
 - Each datagram has source IP address, destination IP address
 - Each datagram carries one transport-layer segment
 - Each segment has source, destination port number
- Host uses **IP addresses & port numbers** to direct segment to appropriate socket



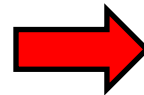
TCP/UDP segment format

Connectionless demultiplexing

- Recall: created socket has host-local port #:
 - `DatagramSocket mySocket1 = new DatagramSocket(12534);`
- Recall: when creating datagram to send into UDP socket, must specify
 - Destination IP address
 - Destination port #

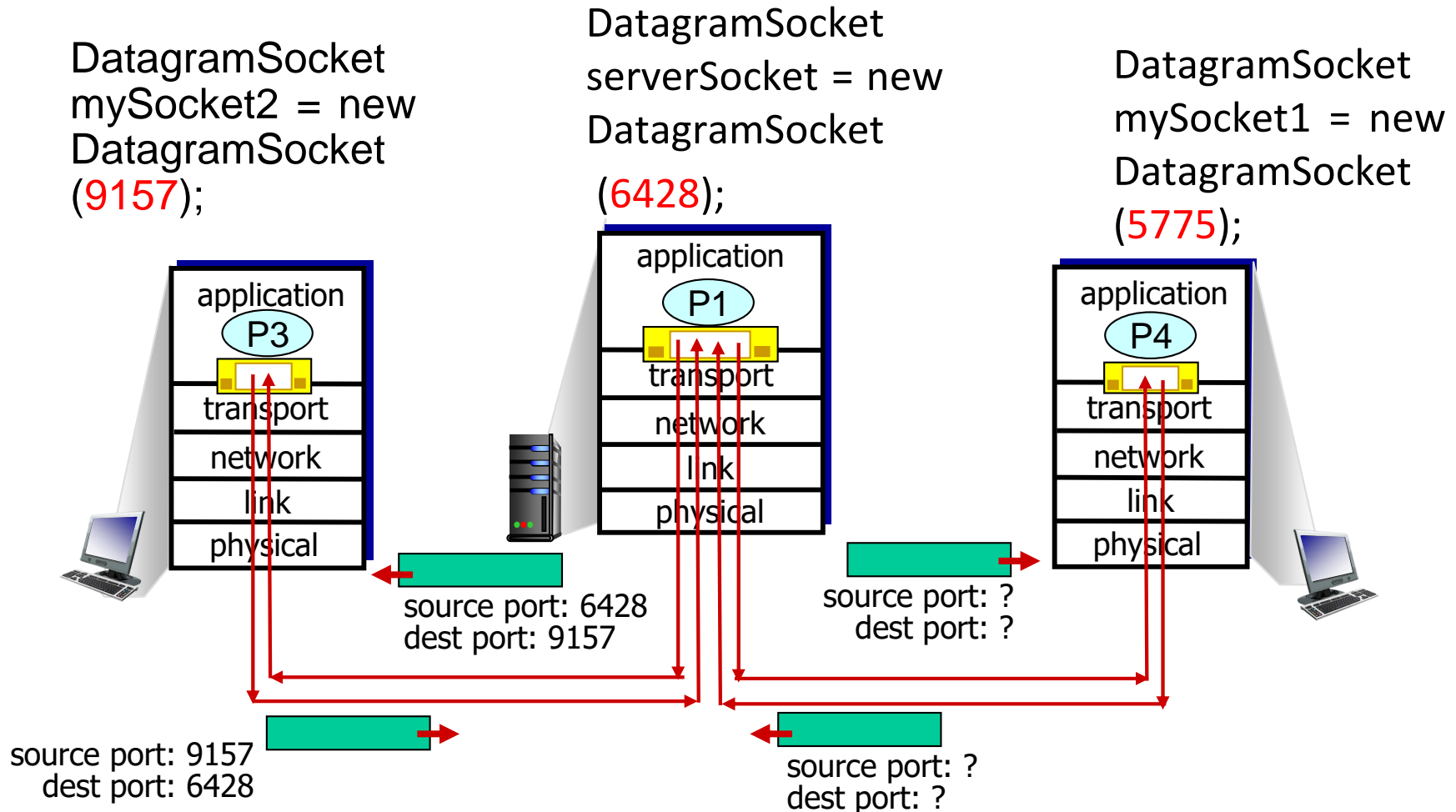
When host receives UDP segment:

- Checks destination port # in segment
- Directs UDP segment to socket with that port #



IP datagrams with **same dest. port #**, but different source IP addresses and/or source port numbers will be directed to **same socket** at dest

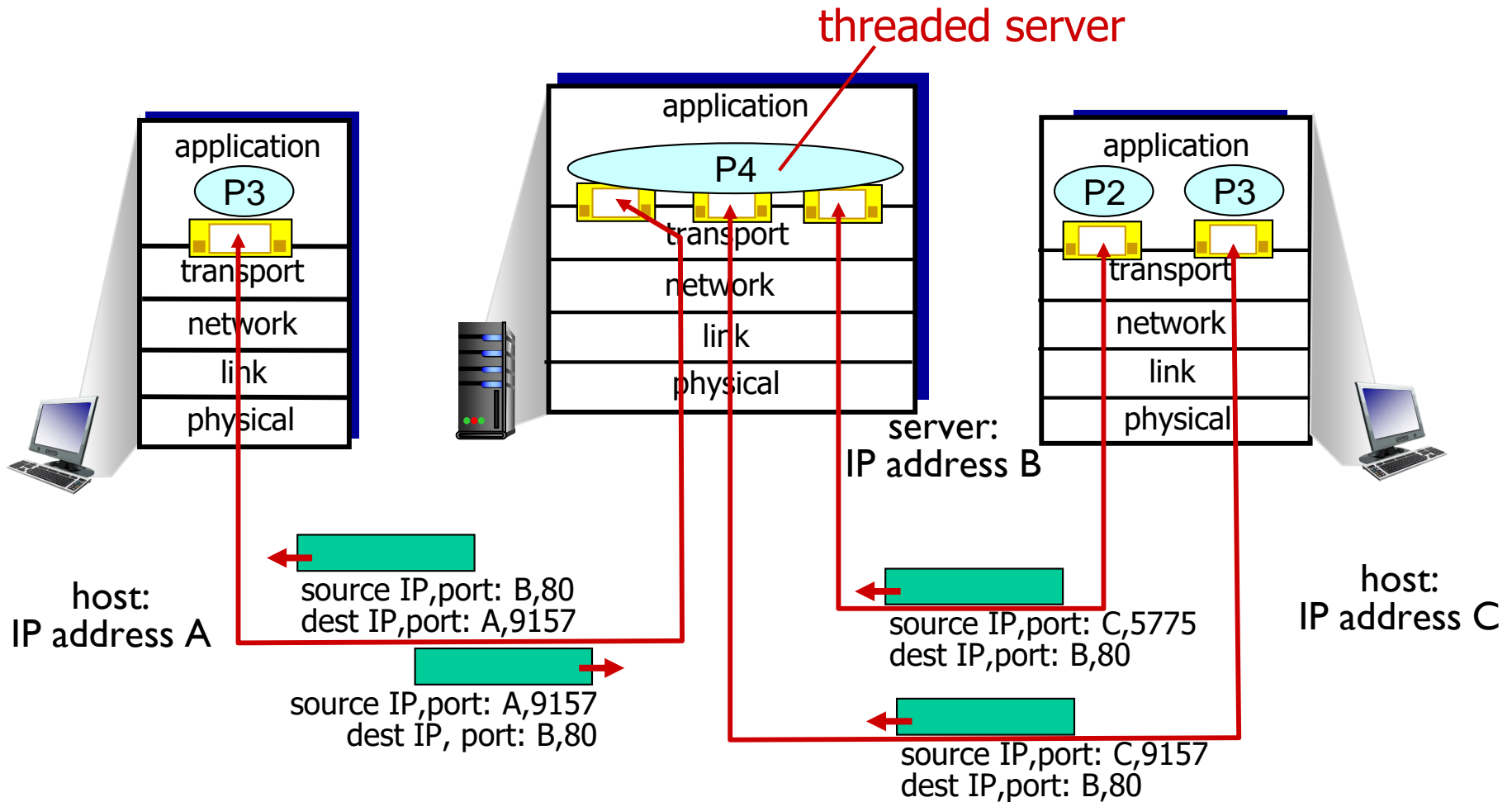
Connectionless demux: example



Connection-oriented demux

- TCP socket identified by 4-tuple:
 - Source IP address
 - Source port number
 - Dest IP address
 - Dest port number
- Demux: receiver uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
 - Each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
 - Non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



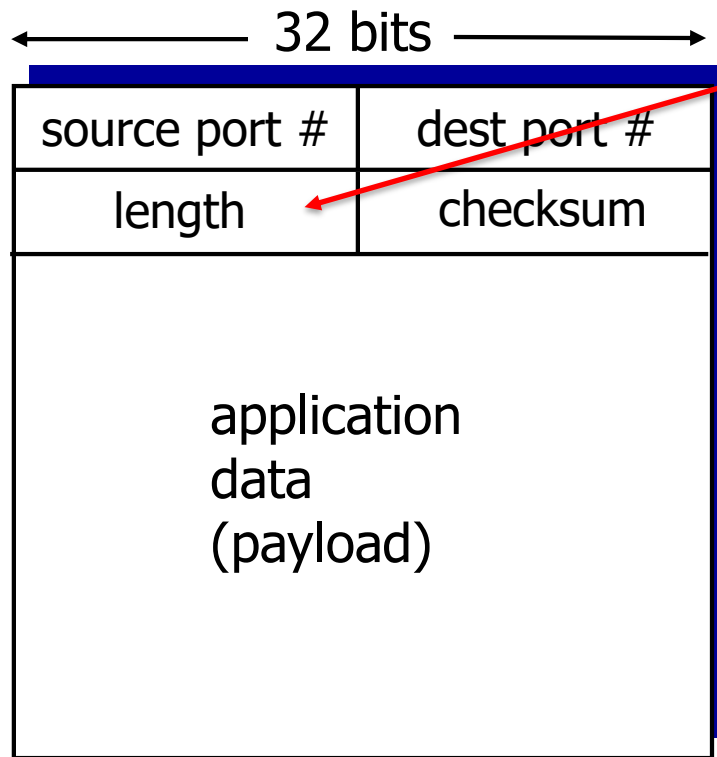
Chapter 3: Roadmap

- 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

UDP: User Datagram Protocol [RFC 768]

- “No frills,” “bare bones” Internet transport protocol
- “Best effort” service, UDP segments may be:
 - Lost
 - Delivered out-of-order to app
- **Connectionless:**
 - No handshaking between UDP sender, receiver
 - Each UDP segment handled independently of others
- 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



length, in bytes of UDP segment, including header

- **Why is there a 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 segment format

UDP checksum

- Goal: detect “errors” (e.g., flipped bits) in transmitted segment
- Sender:
 - Treat segment contents, including header fields, as sequence of 16-bit integers
 - Checksum: addition (one’s complement sum) of segment contents
 - Sender puts checksum value into **UDP checksum field**
- Receiver:
 - Add up all 16-Bit integers, including checksum
 - If result is 1111 1111 1111 1111
 - NO - error detected
 - Otherwise
 - YES - error detected. But maybe errors nonetheless? More later
 -

Internet checksum: example

example: add two 16-bit integers

	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
	<hr/>															
	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
	<hr/>															
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

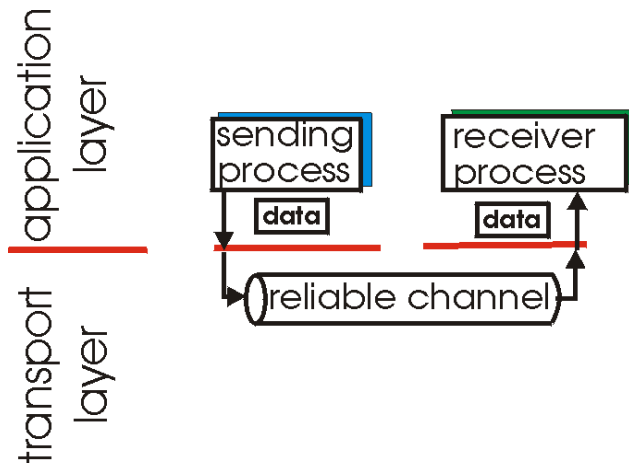
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

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

Principles of reliable data transfer

- Important in application, transport, link layers
 - Top-10 list of important networking topics!

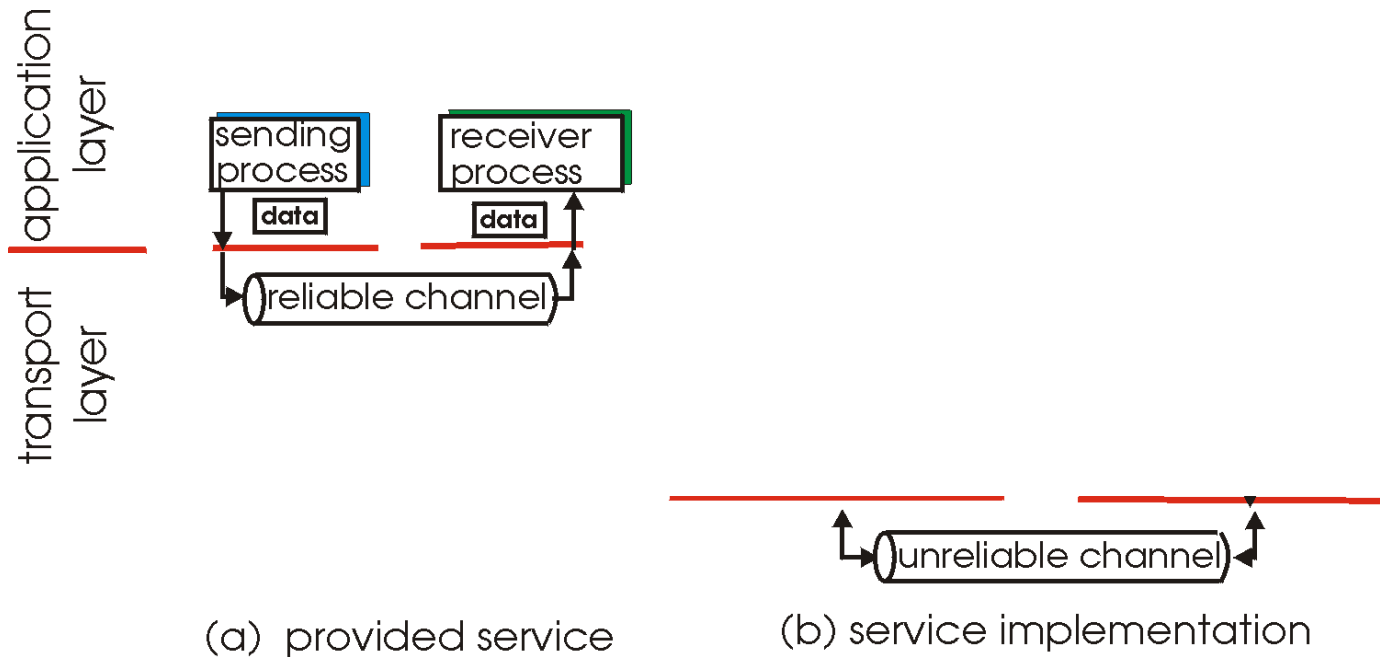


(a) provided service

- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

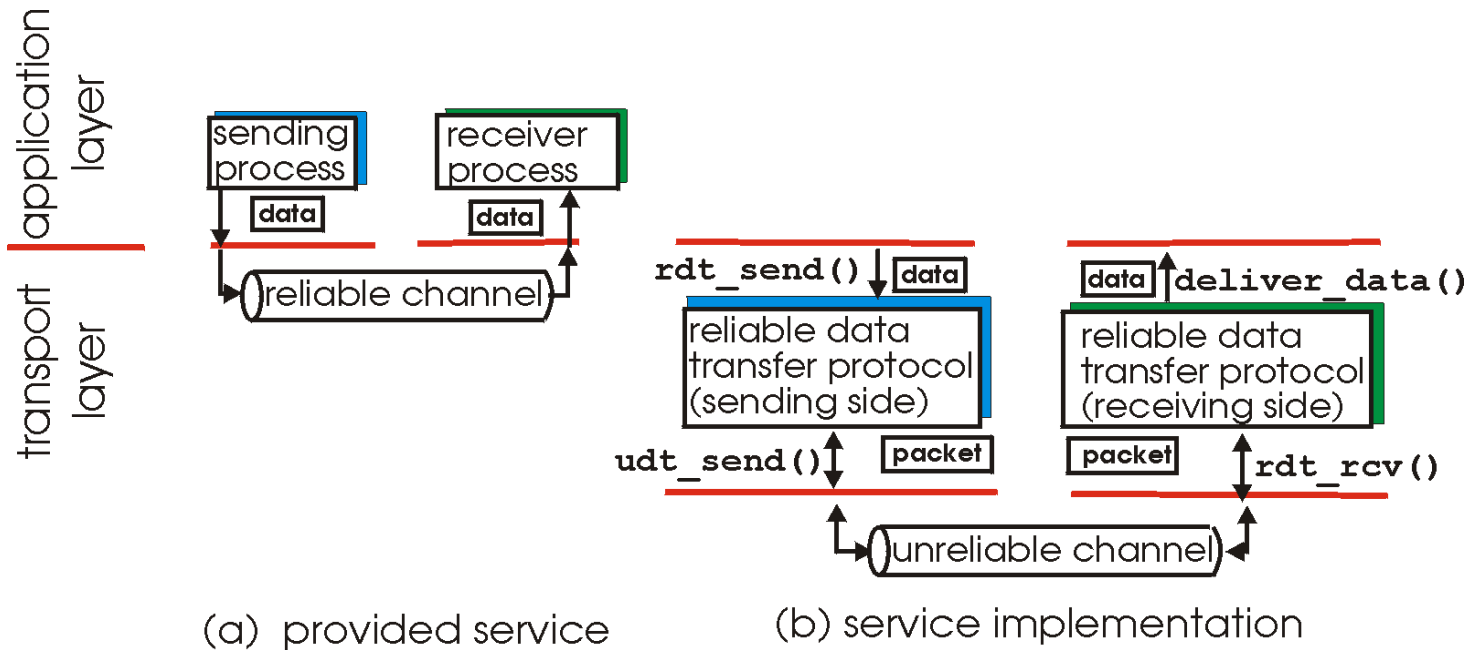
- Important in application, transport, link layers
 - Top-10 list of important networking topics!



- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

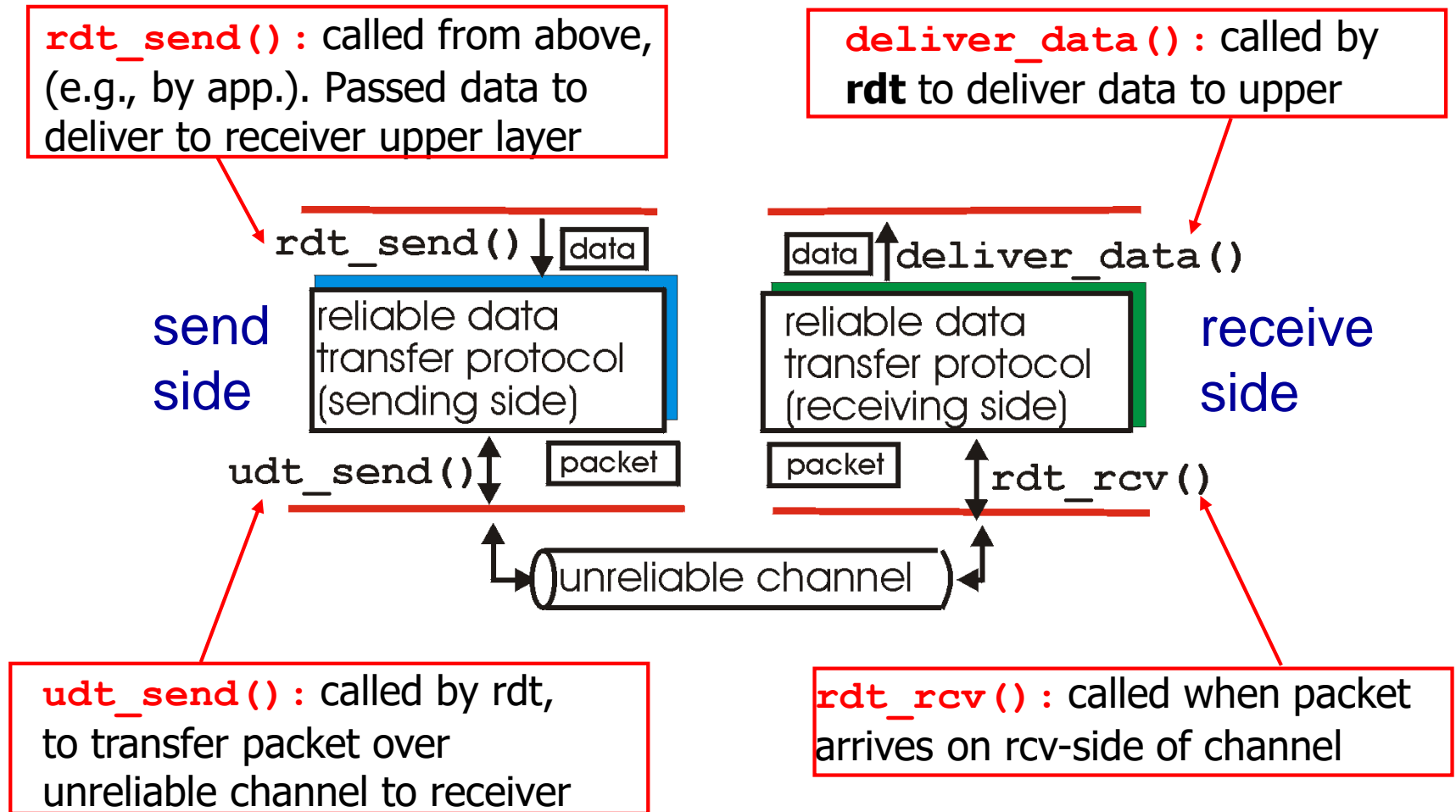
Principles of reliable data transfer

- Important in application, transport, link layers
 - Top-10 list of important networking topics!



- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started



Reliable data transfer: getting started

- We'll:
 - Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
 - Consider only unidirectional data transfer
 - But control info will flow on both directions!

RDT 1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
 - No bit errors
 - No loss of packets
- Simple operation
 - Sender sends data into underlying channel
 - Receiver reads data from underlying channel

RDT 2.0: channel with bit errors

- Underlying channel may flip bits in packet
 - Checksum to detect bit errors
- The question: how to recover from errors:

How do humans recover from “errors” during conversation?

RDT 2.0: channel with bit errors

- Underlying channel may flip bits in packet
 - Checksum to detect bit errors
- The question: how to recover from errors:
 - Acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
 - Negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
 - Sender retransmits pkt on receipt of NAK
- New mechanisms in rdt2.0 (beyond rdt1.0):
 - Error detection
 - Feedback: control msgs (ACK,NAK) from receiver to sender

RDT 2.0 has a fatal flaw!

- What happens if ACK/NAK corrupted?
 - Sender doesn't know what happened at receiver!
 - Can't just retransmit: possible duplicate
- Handling duplicates:
 - Sender retransmits current pkt if ACK/NAK corrupted
 - Sender adds sequence number to each pkt
 - Receiver discards (doesn't deliver up) duplicate pkt
- Stop and wait
 - Sender sends one packet,
 - Then waits for receiver
 - Response

RDT 2.1: discussion

- Sender:
 - Seq # added to pkt
 - Two seq. #'s (0,1) will suffice. Why?
 - Must check if received ACK/NAK corrupted
 - Twice as many states
 - State must “remember” whether “expected” pkt should have seq # of 0 or 1
- Receiver:
 - Must check if received packet is duplicate
 - State indicates whether 0 or 1 is expected pkt seq #
 - Note: receiver can not know if its last ACK/NAK received OK at sender

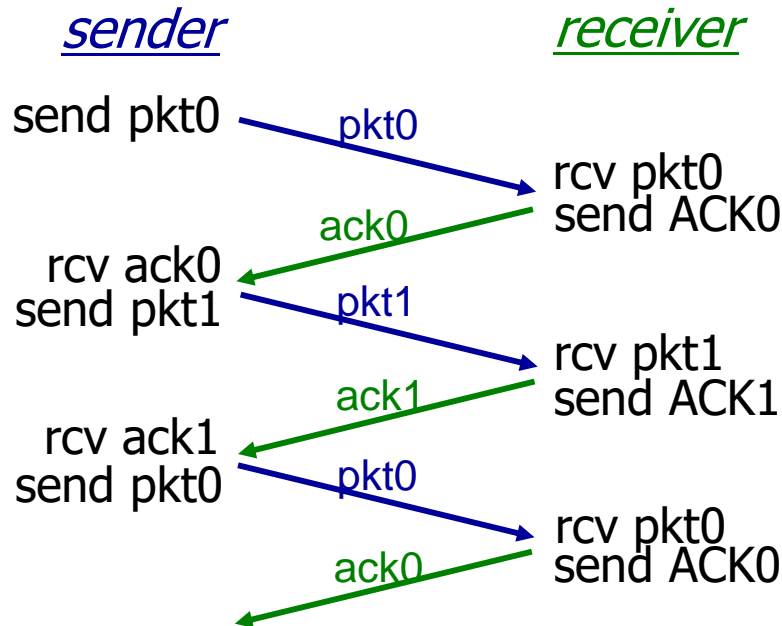
RDT 2.2: a NAK-free protocol

- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
 - Receiver must explicitly include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: retransmit current pkt

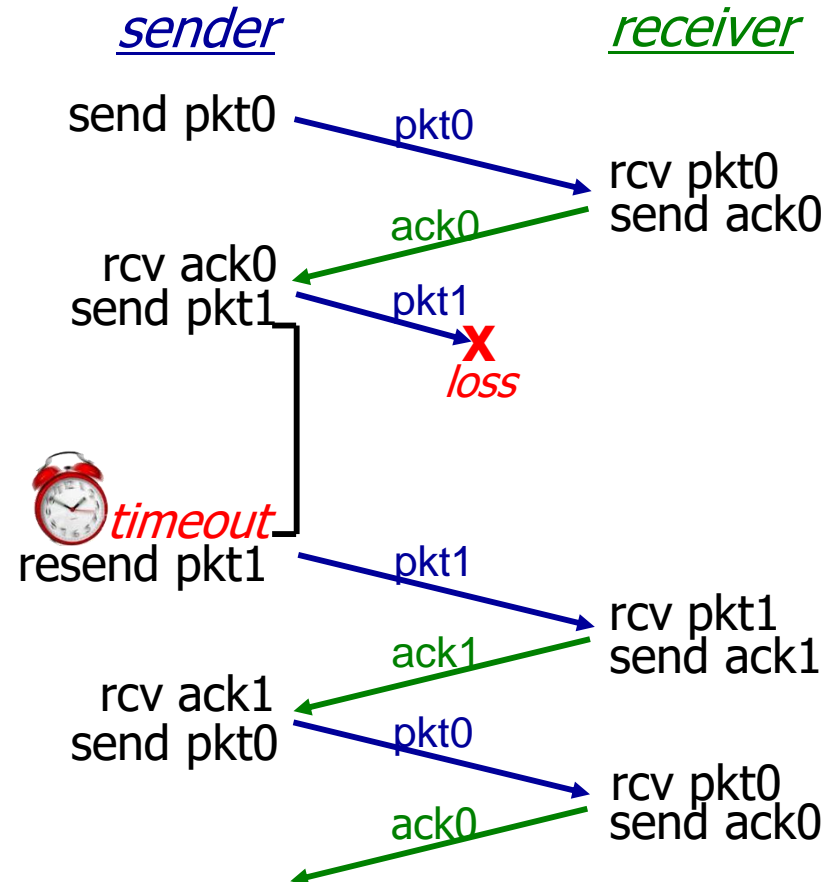
RDT 3.0: channels with errors and loss

- New assumption:
 - Underlying channel can also lose packets (data, ACKs)
 - Checksum, seq. #, ACKs, retransmissions will be of help ... but not enough
- Approach:
 - Sender waits “reasonable” amount of time for ACK
 - Retransmits if no ACK received in this time
 - If pkt (or ACK) just delayed (not lost):
 - Retransmission will be duplicate, but seq. #'s already handles this
 - Receiver must specify seq # of pkt being ACKed
 - Requires countdown timer

RDT 3.0 in action

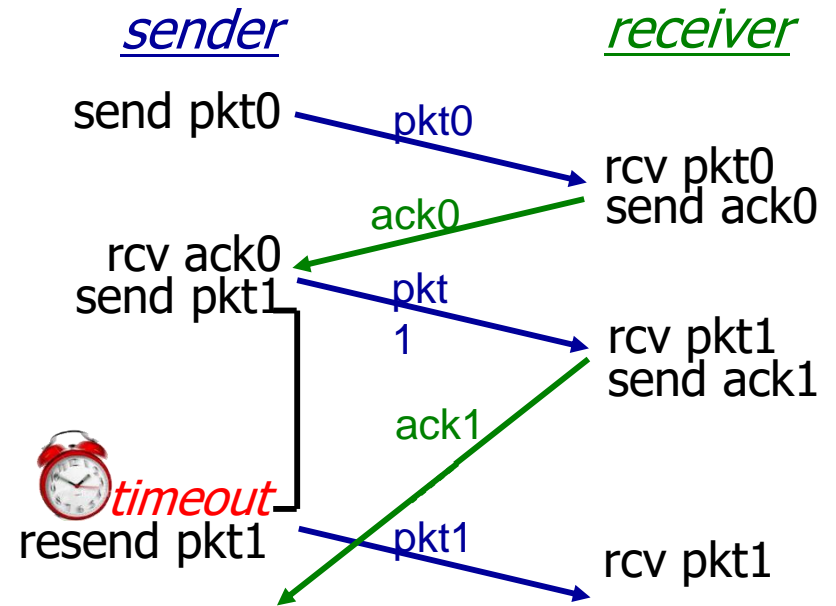
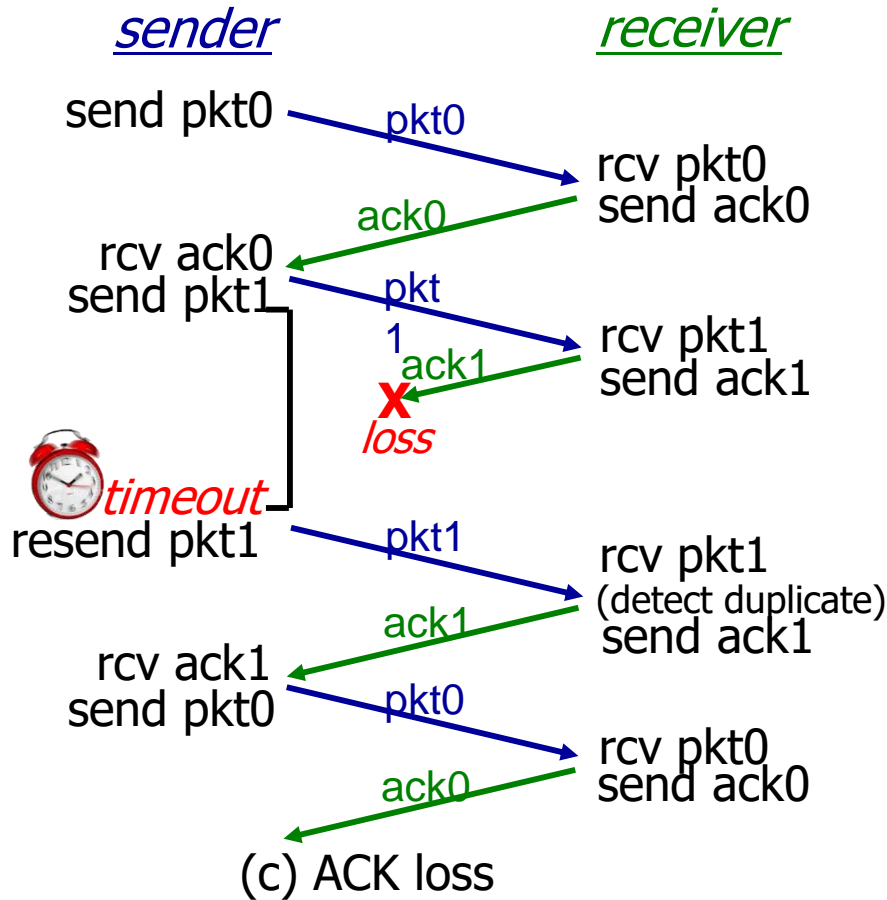


(a) no loss



(b) packet loss

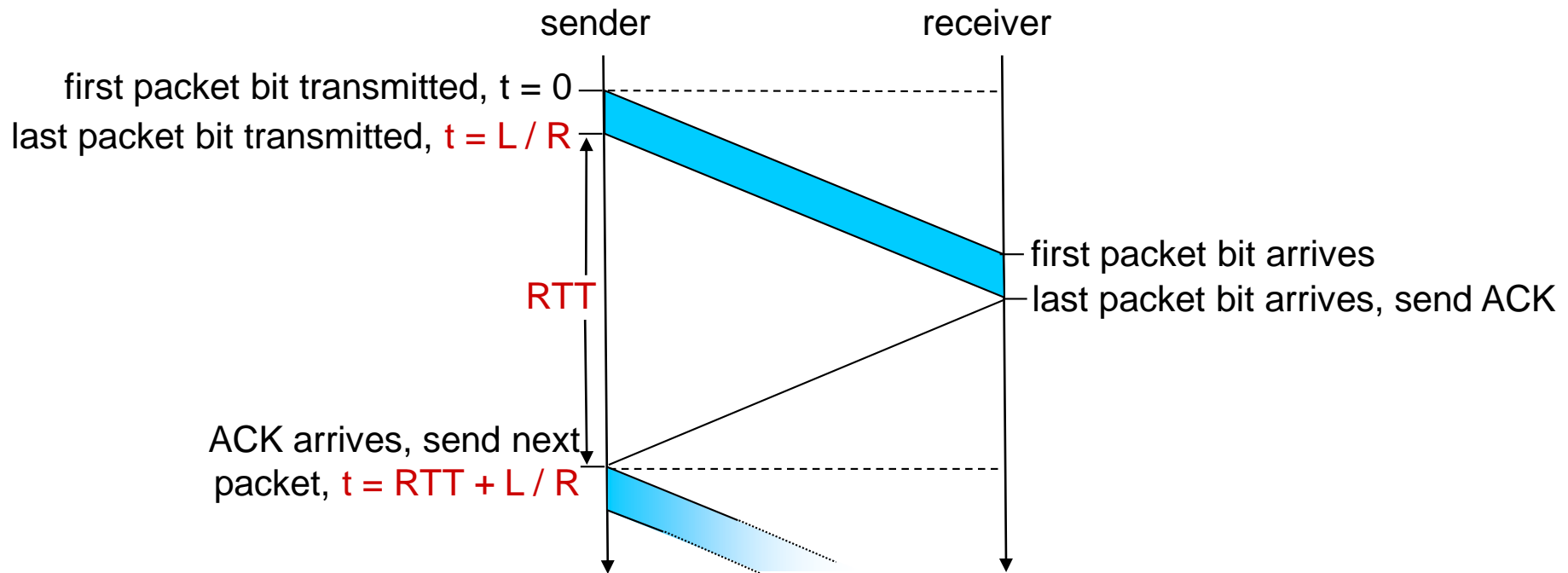
RDT 3.0 in action



Performance of rdt3.0

- RDT 3.0 is correct, but performance stinks
- E.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:
 - $D_{trans} = \frac{L}{R} = \frac{8.000 \text{ bits}}{10^9 \frac{\text{bits}}{\text{sec}}} = 8 \text{ microseconds}$
- U sender: utilization – fraction of time sender busy sending
 - $U_{sender} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{0.008}{30.008} = 0.00027$
 - If RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec through over 1 Gbps link
- Network protocol limits use of physical resources!

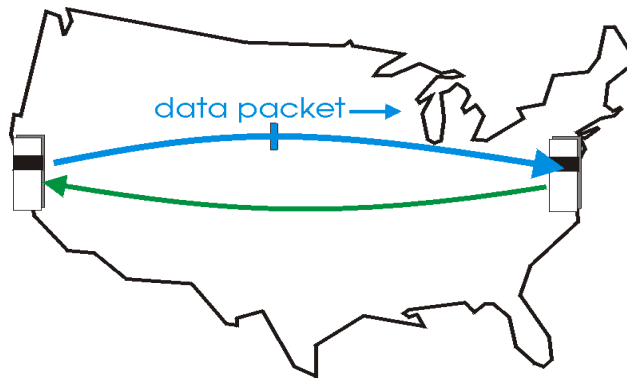
RDT 3.0: stop-and-wait operation



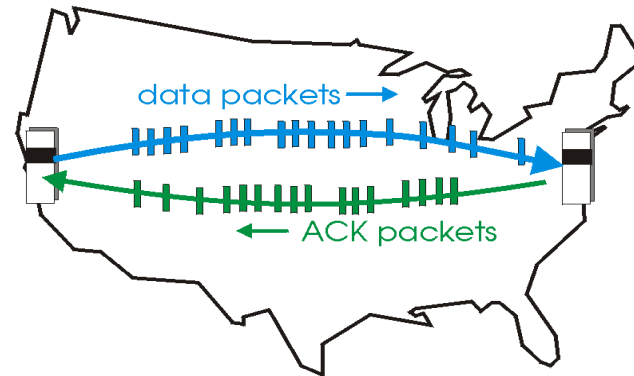
$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027$$

Pipelined protocols

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



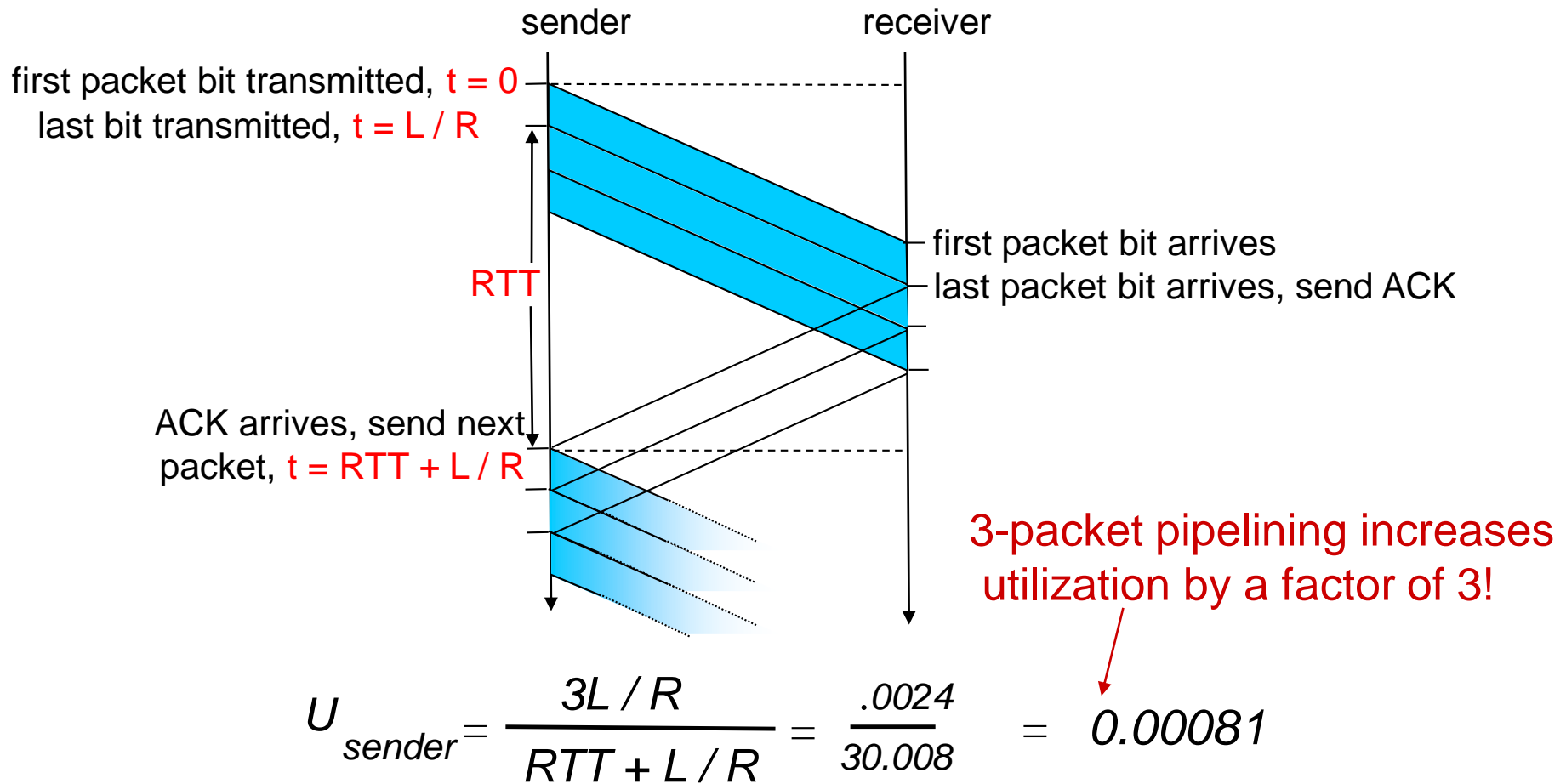
(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

- Two generic forms of pipelined protocols:
 - Go-Back-N,
 - Selective repeat

Pipelining: increased utilization

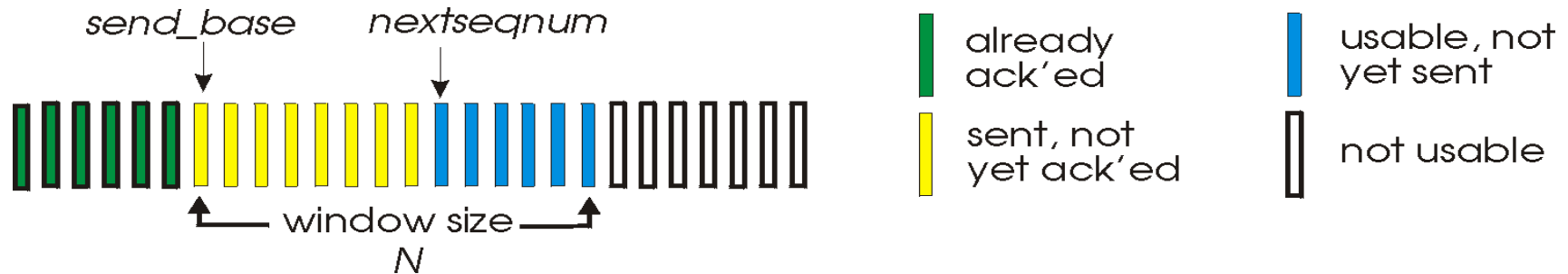


Pipelined protocols: overview

- Go-back-N:
 - Sender can have up to N unACKed packets in pipeline
 - Receiver only sends **cumulative ACK**
 - Doesn't ACK packet if there's a gap
 - 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
 - RCVR sends **individual ACK** for each packet
 - Sender maintains timer for each unACKed packet
 - When timer expires, retransmit only that unACKed packet

Go-Back-N: Sender

- K-bit seq # in pkt header
- “Window” of up to N, consecutive unACKed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
 - may receive duplicate ACKs (see receiver)
- Timer for oldest in-flight pkt
- Timeout(n): Retransmit packet n and all higher seq # pkts in window

GBN: Receiver

- ACK-only: always send ACK for correctly-received pkt with highest **in-order** seq #
 - May generate duplicate ACKs
 - Need only remember expectedseqnum
- Out-of-order pkt:
 - Discard (don't buffer): **no receiver buffering!**
 - Re-ACK pkt with highest in-order seq #

GBN in action

sender window (N=4)

0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8

sender

send pkt0
 send pkt1
 send pkt2
 send pkt3
 (wait)

rcv ack0, send pkt4
 rcv ack1, send pkt5

ignore duplicate ACK



pkt 2 timeout

send pkt2
 send pkt3
 send pkt4
 send pkt5

receiver

receive pkt0, send ACK0
 receive pkt1, send ACK1

receive pkt3, discard,
 (re)send ACK1

receive pkt4, discard,
 (re)send ACK1

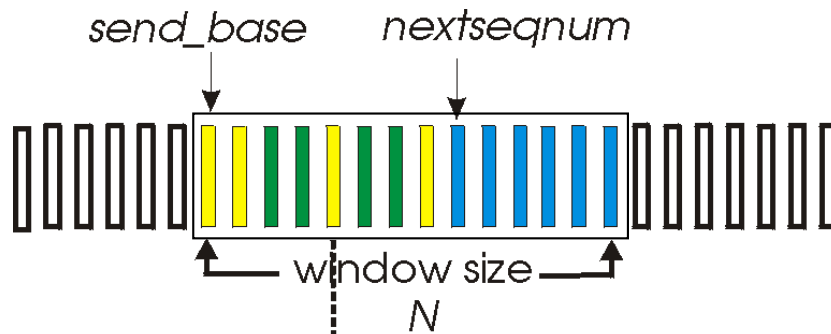
receive pkt5, discard,
 (re)send ACK1

rcv pkt2, deliver, send ACK2
 rcv pkt3, deliver, send ACK3
 rcv pkt4, deliver, send ACK4
 rcv pkt5, deliver, send ACK5

Selective repeat



- Receiver individually acknowledges all correctly received pkts
 - Buffers pkts, as needed, for eventual in-order delivery to upper layer
- Sender only resends pkts for which ACK not received
 - Sender timer for each unACKed pkt
- Sender window
 - N consecutive seq #'s
 - Limits seq #'s of sent, unACKed pkts

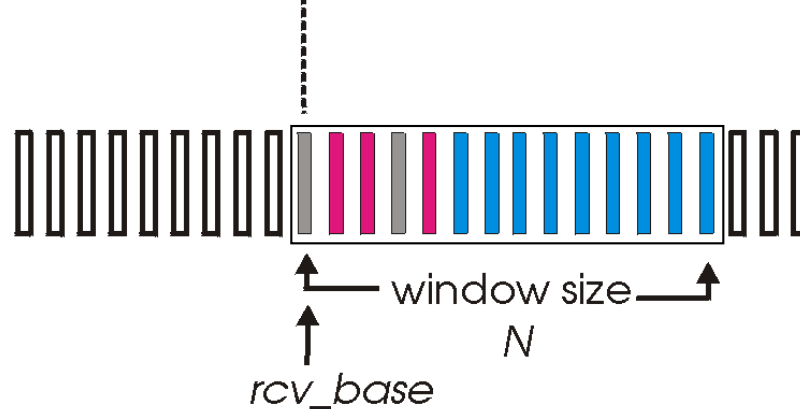
Selective repeat: sender, receiver windows





(a) sender view of sequence numbers



 already
ack'ed
 sent, not
yet ack'ed

 usable, not
yet sent
 not usable



(b) receiver view of sequence numbers

 out of order
(buffered) but
already ack'ed
 Expected, not
yet received

 acceptable
(within window)
 not usable

Selective repeat

Sender:

- Data from above:
 - If next available seq # in window, send pkt
- Timeout(n):
 - resend pkt n, restart timer
- ACK(n) in [sendbase, sendbase+N]:
 - Mark pkt n as received
 - If n smallest unACKed pkt, advance window base to next unACKed seq #

Receiver:

- Pkt n in [rcvbase, rcvbase+N-1]
 - Send ACK(n)
 - Out-of-order: buffer
 - In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
- Pkt n in [rcvbase-N, rcvbase-1]
 - ACK(n)
- Otherwise:
 - Ignore

Selective repeat in action

sender window (N=4)

0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 []

0 1 2 3 4 5 6 7 8 rcv ack0, send pkt4
 0 1 2 3 4 5 6 7 8 rcv ack1, send pkt5

0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8
 0 1 2 3 4 5 6 7 8

sender

send pkt0
 send pkt1
 send pkt2
 send pkt3
 (wait)

record ACK3 arrived



pkt 2 timeout

send pkt2

Record ACK4 arrived

record ACK5 arrived

receiver

receive pkt0, send ACK0
 receive pkt1, send ACK1

receive pkt3, buffer,
 send ACK3

receive pkt4, buffer,
 send ACK4

receive pkt5, buffer,
 send ACK5

rcv pkt2; deliver pkt2,
 pkt3, pkt4, pkt5; send ACK2

X/loss

Chapter 3: Roadmap

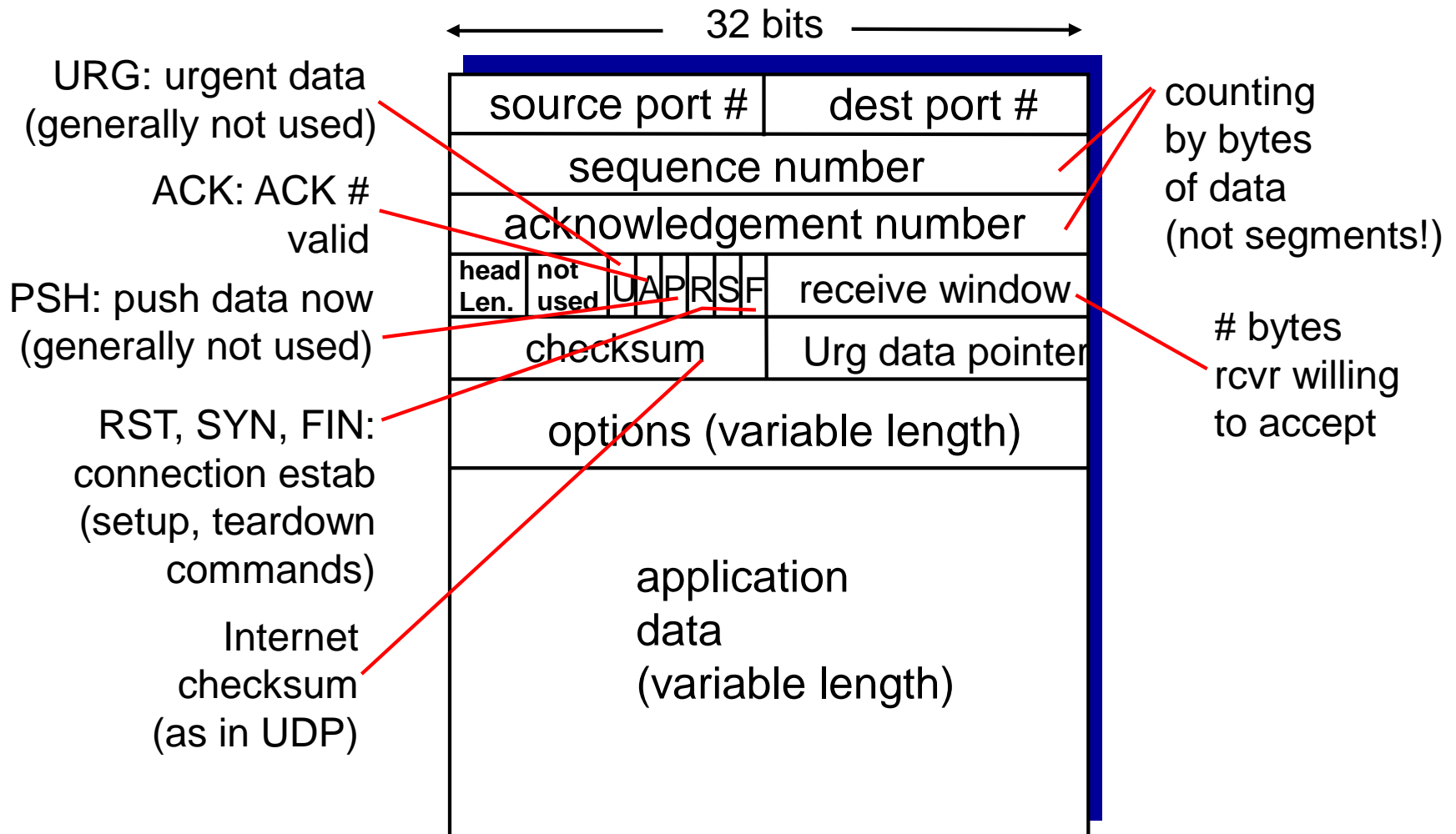
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
 - **Segment structure**
 - Reliable data transfer
 - Flow control
 - Connection management
- Principles of congestion control
- TCP congestion control

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point:
 - One sender, one receiver
- Reliable, in-order byte stream:
 - No “message boundaries”
- Pipelined:
 - TCP congestion and flow control set window size
- Full duplex data:
 - Bi-directional data flow in same connection
 - MSS: maximum segment size
- Connection-oriented:
 - Handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- Flow controlled:
 - Sender will not overwhelm receiver

TCP segment structure

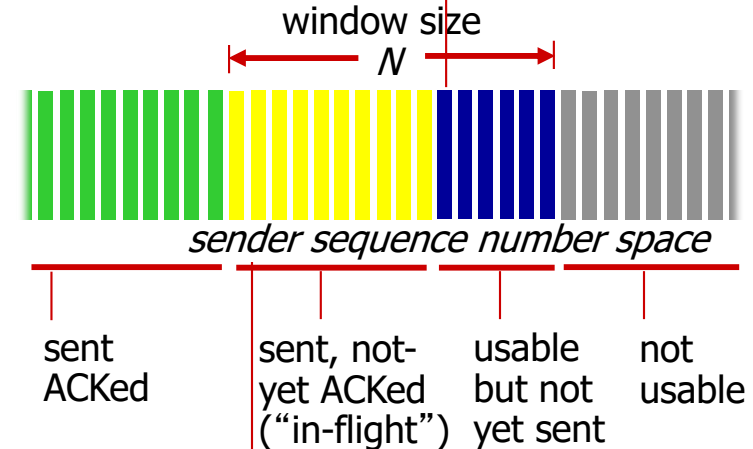


TCP seq. numbers, ACKs

- Sequence numbers:
 - Byte stream “number” of first byte in segment’s data
- Acknowledgements:
 - Seq # of next byte expected from other side
 - Cumulative ACK
- **Q:** How receiver handles out-of-order segments
 - **A:** TCP spec doesn’t say, - up to implementor

outgoing segment from sender

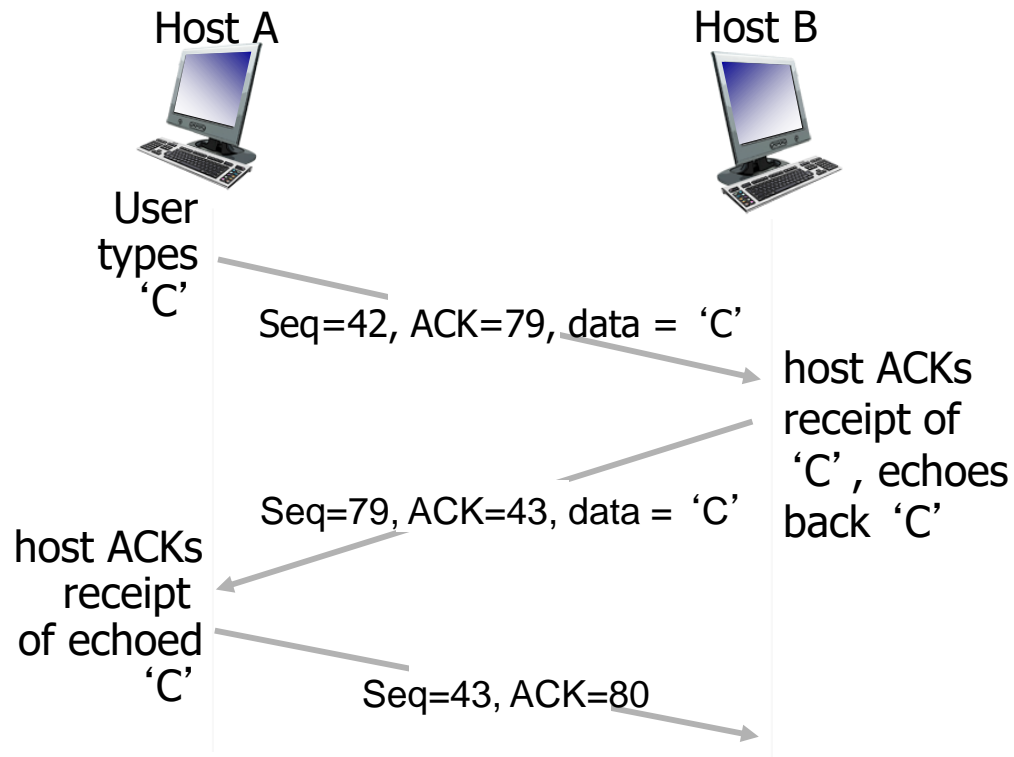
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



incoming segment to sender

source port #	dest port #
sequence number	
acknowledgement number	
	A
checksum	urg pointer

TCP seq. numbers, ACKs



simple telnet scenario

TCP round trip time, timeout

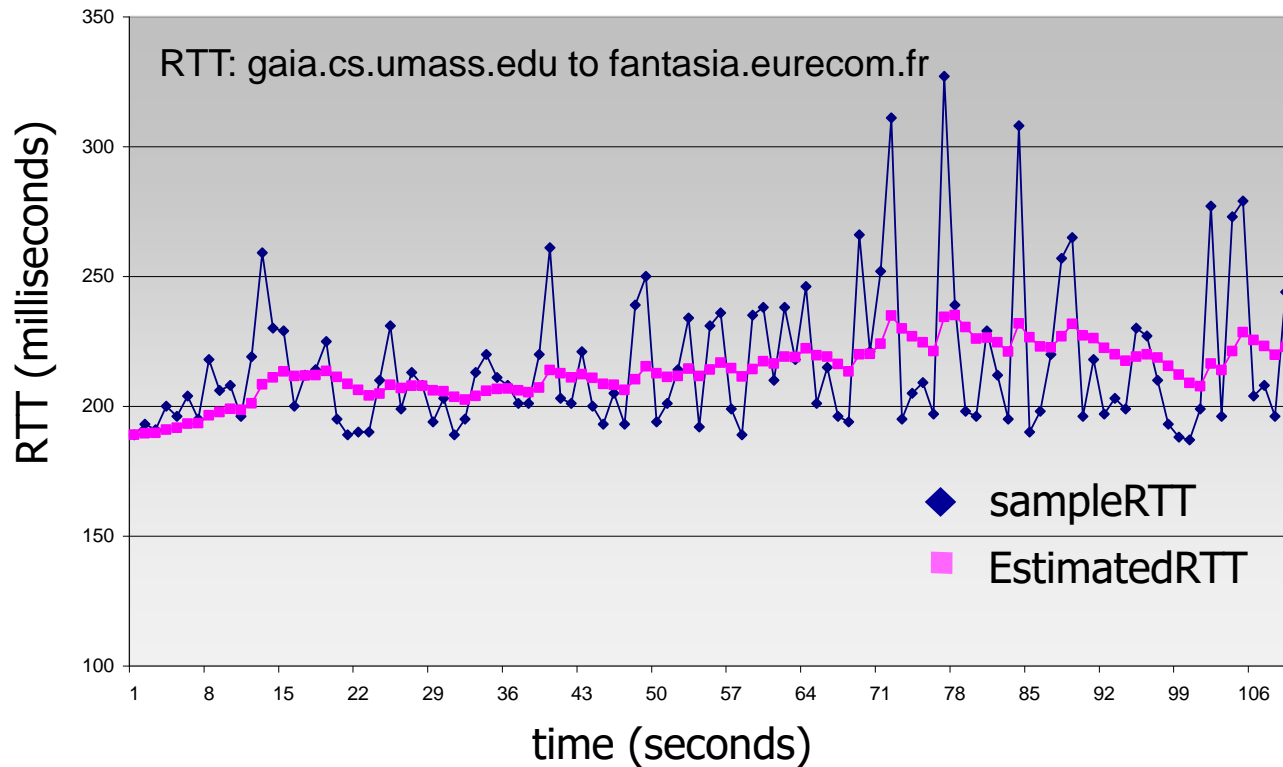
- **Q:** how to set TCP timeout value?
 - Longer than RTT
 - But RTT varies
 - Too short: premature timeout, unnecessary retransmissions
 - Too long: slow reaction to segment loss
- **Q:** How to estimate RTT?
 - SampleRTT: measured time from segment transmission until ACK receipt
 - Ignore retransmissions
 - SampleRTT will vary, want estimated RTT “smoother”
 - Average several recent measurements, not just current SampleRTT

TCP round trip time, timeout

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

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$

TCP round trip time, timeout



TCP round trip time, timeout

- Timeout interval: EstimatedRTT plus “safety margin”
 - Large variation in EstimatedRTT -> larger safety margin
- Estimate SampleRTT deviation from EstimatedRTT:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



↑
estimated RTT

↑
“safety margin”

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

Host A

Host B

timeout

Seq=92, 8 bytes of data

ACK=100

X

Seq=92, 8 bytes of data

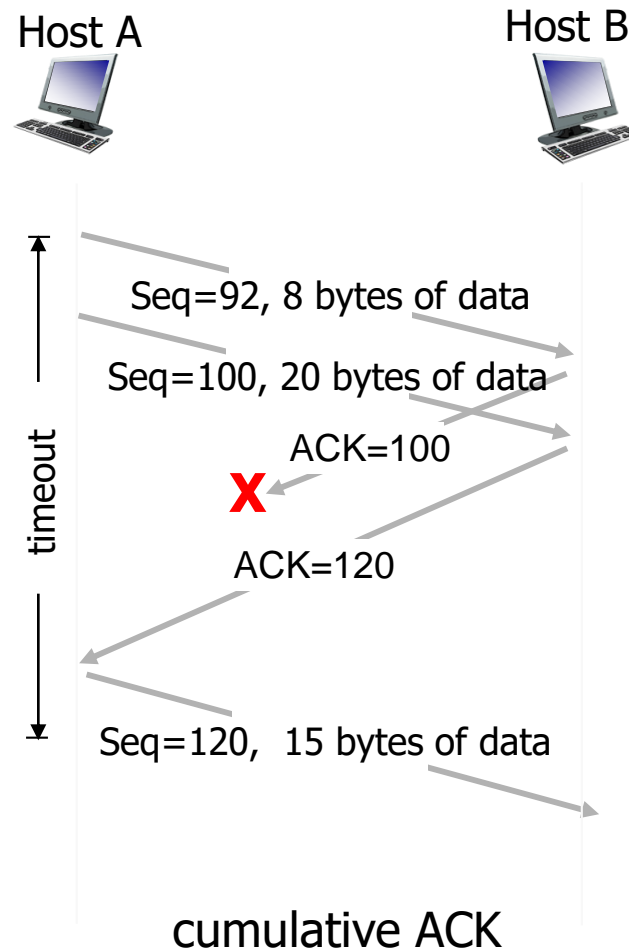
ACK=100

The diagram illustrates the Stop-and-Wait protocol between Host A and Host B. The sequence of events is as follows:

- Host A starts with **SendBase=92**.
- Host A sends **Seq=92, 8 bytes of data** to Host B.
- Host A sends **Seq=100, 20 bytes of data** to Host B.
- A **timeout** occurs at Host A.
- Host B receives the first packet and sends **ACK=100** to Host A.
- Host B receives the second packet and sends **ACK=120** to Host A.
- Host A receives **ACK=100** and updates **SendBase=100**.
- Host A receives **ACK=120** and updates **SendBase=120**.
- Host A sends **Seq=92, 8 bytes of data** to Host B.
- Host B receives the packet and sends **ACK=120** to Host A.
- Host A receives **ACK=120** and updates **SendBase=120**.

61

TCP: retransmission scenarios



TCP ACK generation

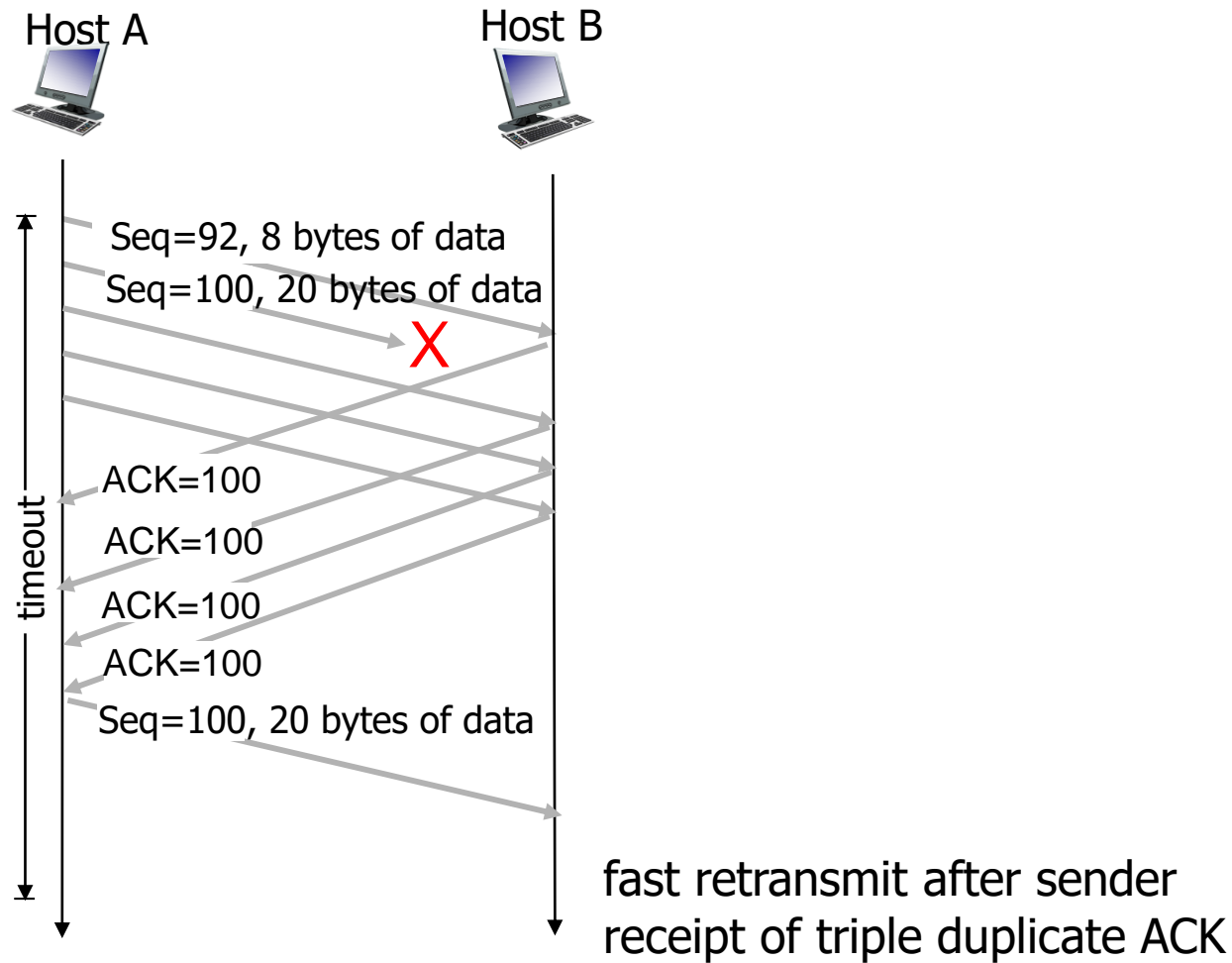
[RFC 1122, RFC 2581]

<i>Event at receiver</i>	<i>TCP receiver action</i>
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send <i>duplicate ACK</i> , indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

- Time-out period often relatively long:
 - Long delay before resending lost packet
- 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.
- TCP fast retransmit
 - If sender receives **3 duplicate 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



Chapter 3: Roadmap

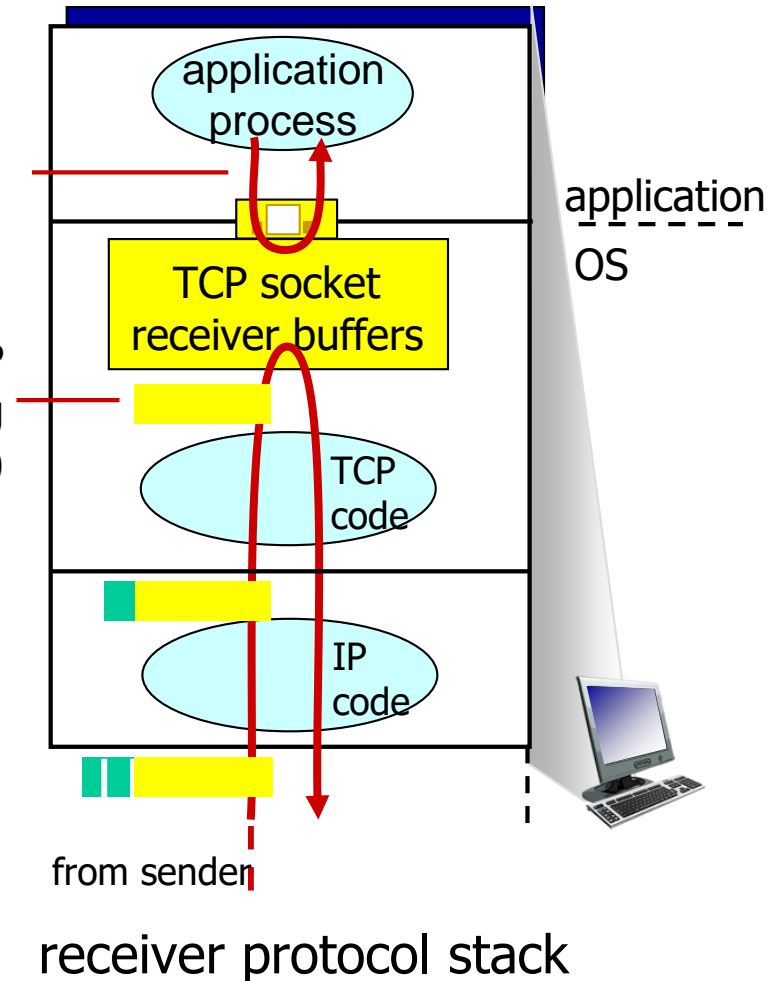
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
 - Segment structure
 - Reliable data transfer
 - **Flow control**
 - Connection management
- Principles of congestion control
- TCP congestion control

TCP flow control

... slower than TCP receiver is delivering (sender is sending)

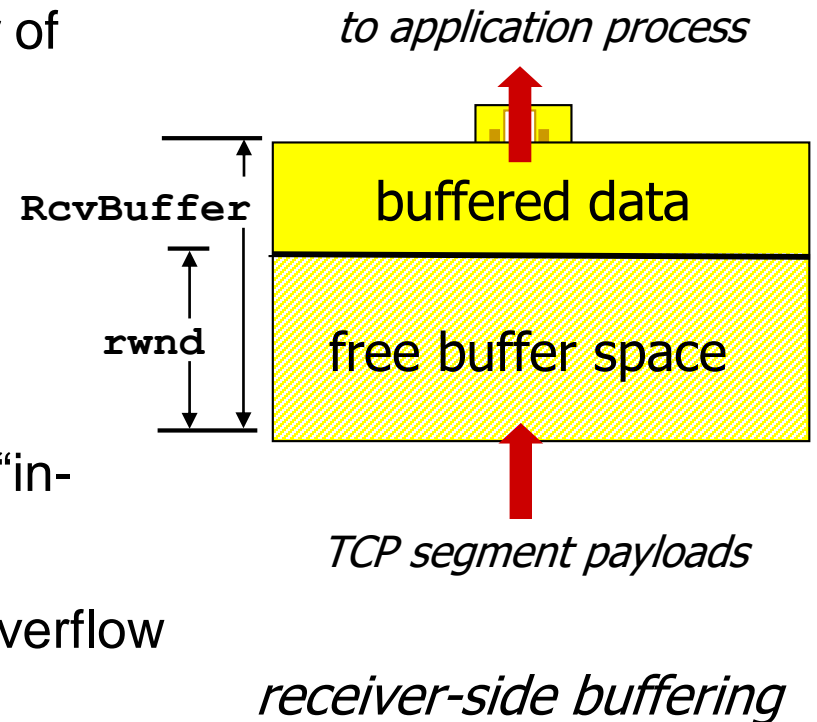
flow control

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

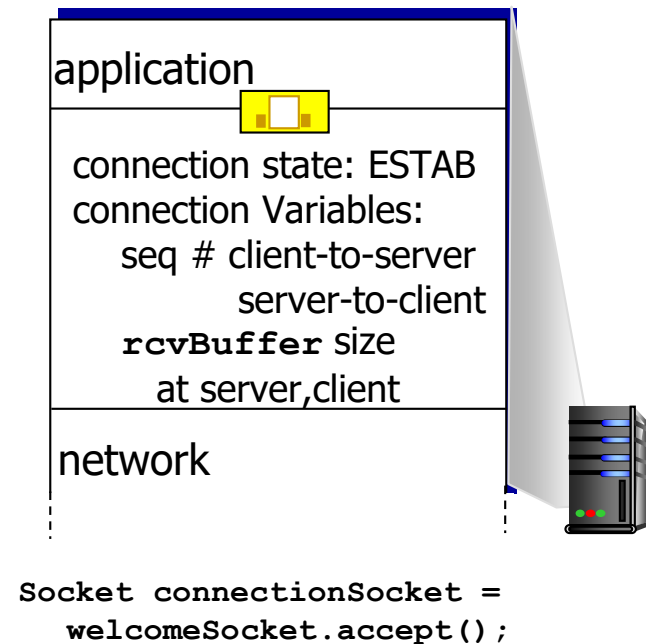
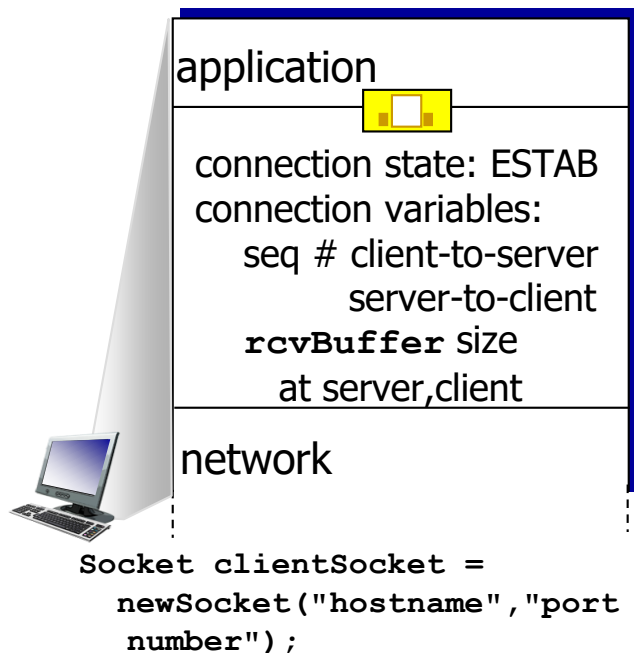


Chapter 3: Roadmap

- 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

Connection Management

- Before exchanging data, sender/receiver “handshake”:
 - Agree to establish connection (each knowing the other willing to establish connection)
 - Agree on connection parameters



TCP 3-way handshake

client state

LISTEN

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

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



SYNbit=1, Seq=x

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

ACKbit=1, ACKnum=y+1

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

received ACK(y)
indicates client is live

server state

LISTEN

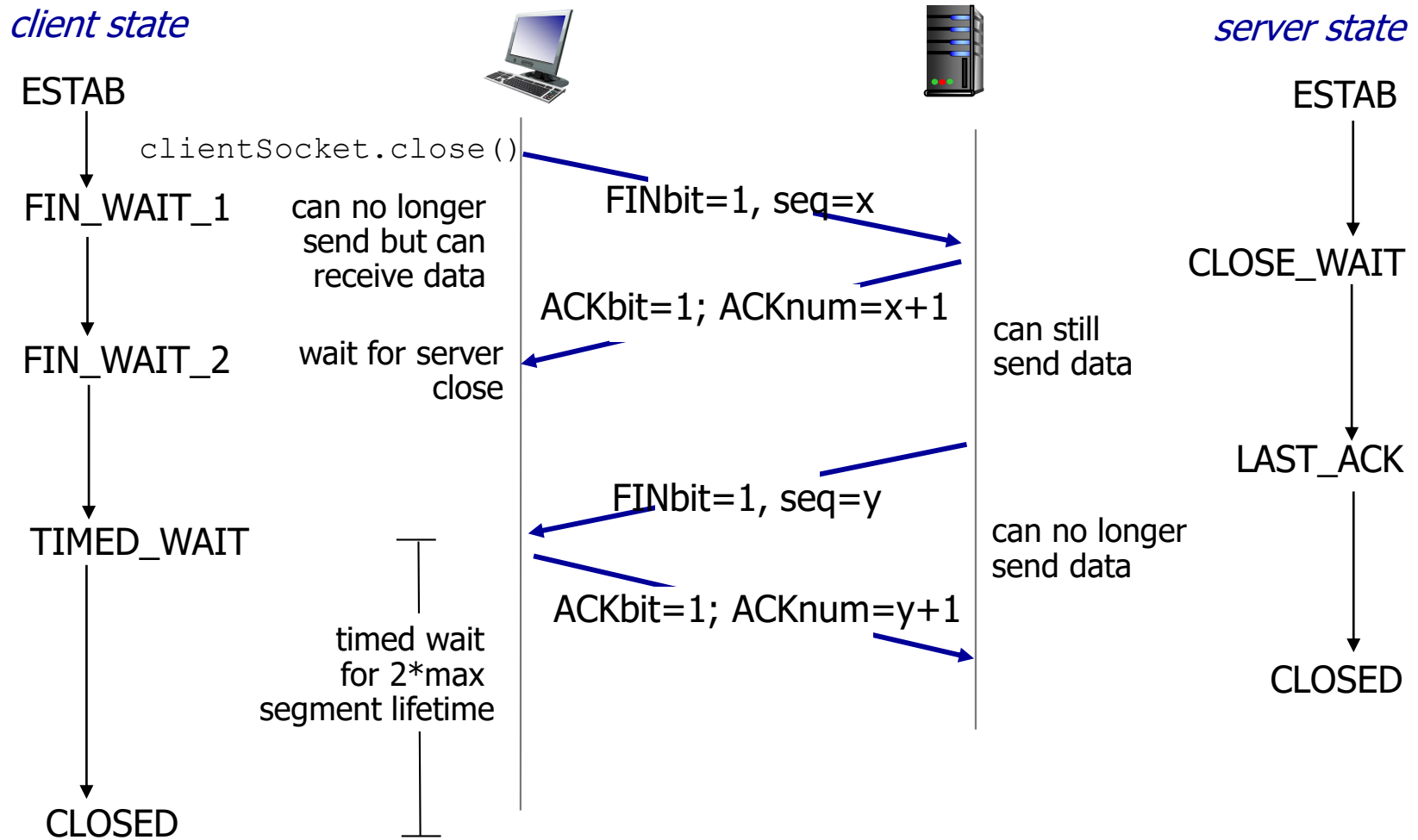
SYN RCVD

ESTAB

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



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

Principles of congestion control

- Congestion:
 - Informally: “too many sources sending too much data too fast for network to handle”
 - Different from flow control!
 - Manifestations:
 - Lost packets (buffer overflow at routers)
 - Long delays (queueing in router buffers)
 - A top-10 problem!

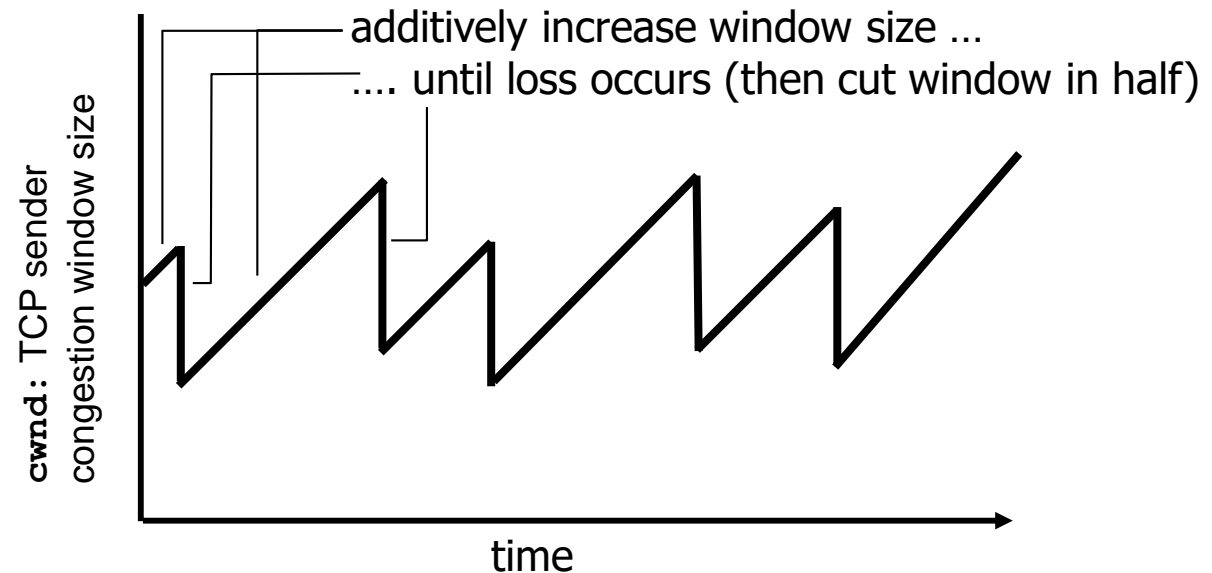
Approaches towards congestion control

- Two broad approaches towards congestion control:
 - End-end congestion control:
 - No explicit feedback from network
 - Congestion inferred from end-system observed loss, delay
 - Approach taken by TCP
 - Network-assisted congestion control
 - Routers provide feedback to end systems
 - Single bit indicating congestion
 - Explicit rate for sender to send at

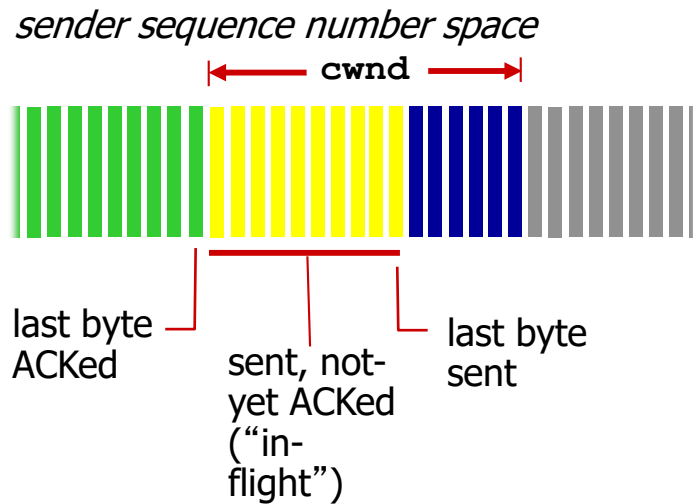
TCP congestion control: additive increase multiplicative decrease

- 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: details



- Sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

- cwnd** is dynamic, function of perceived network congestion

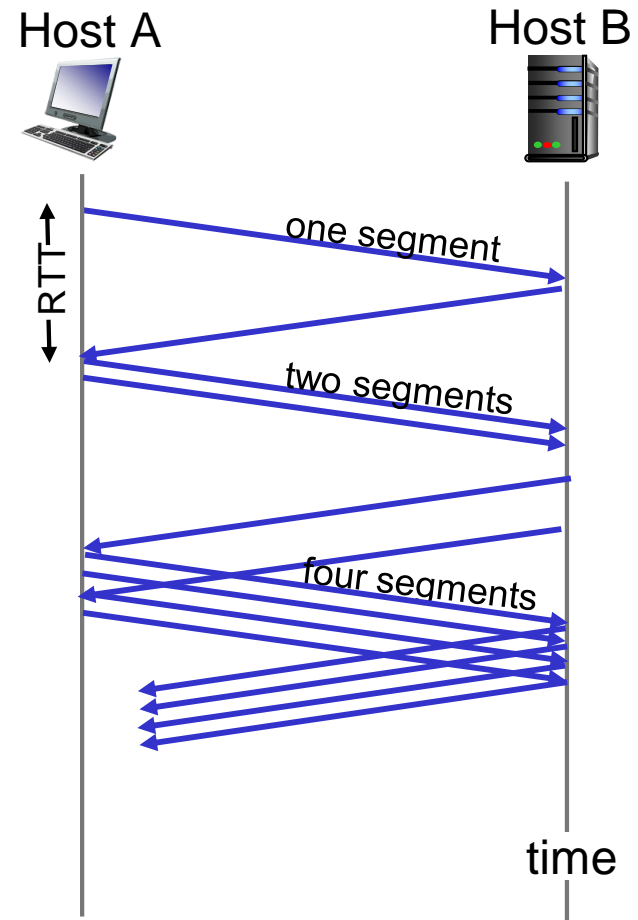
TCP sending rate:

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

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

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

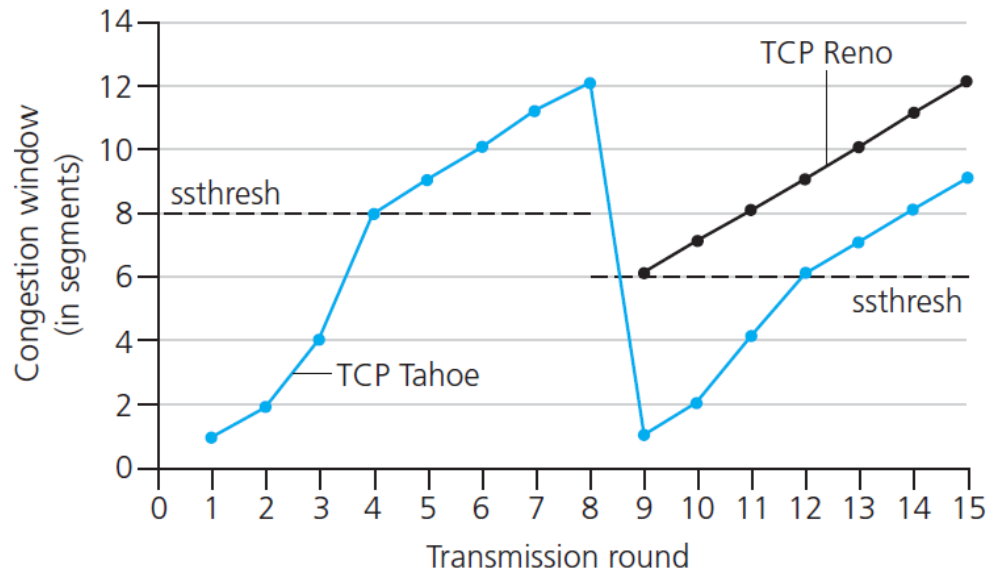


TCP: detecting, reacting to loss

- Loss indicated by timeout:
 - **Cwnd** set to 1 MSS;
 - Window then grows exponentially (as in slow start) to threshold, then grows linearly
- Loss indicated by 3 duplicate ACKs: **TCP RENO**
 - Dup ACKs indicate network capable of delivering some segments
 - **Cwnd** is cut in half window then grows linearly
- **TCP Tahoe** always sets **cwnd** to 1 (timeout or 3 duplicate ACKs)
- Linear growth of cwnd: **TCP Congestion Avoidance**

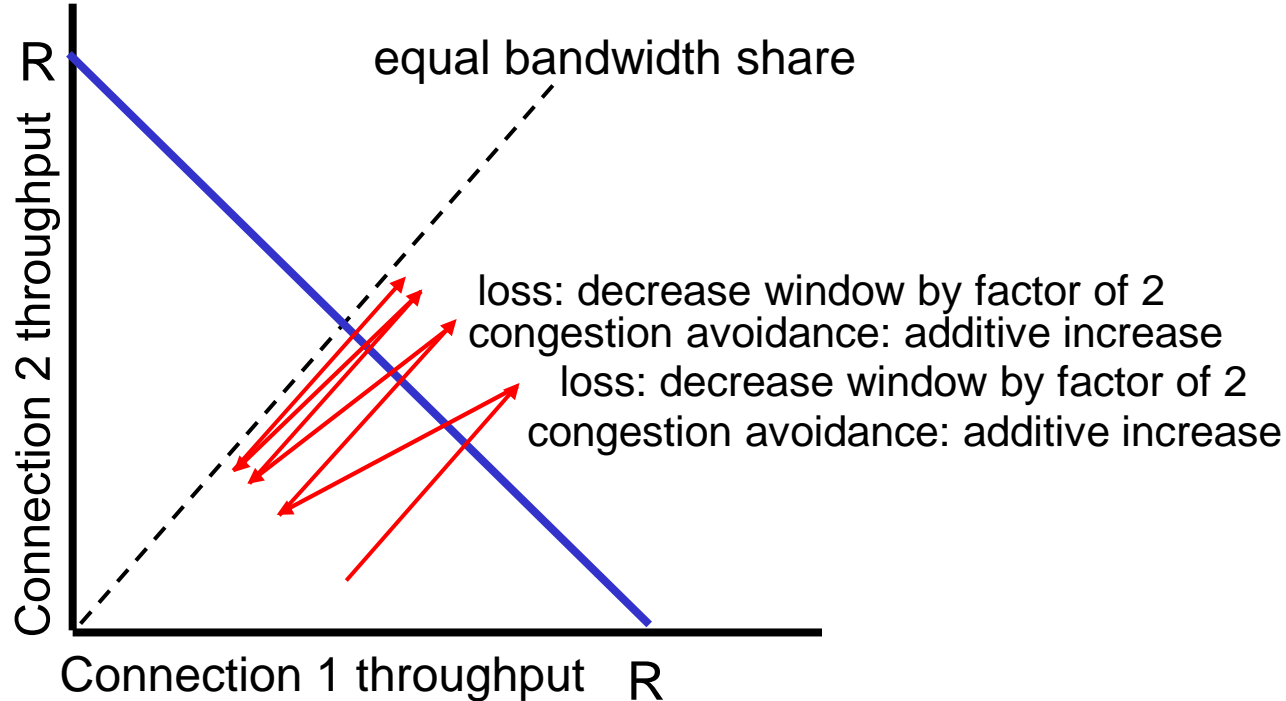
TCP: switching from Slow Start to Congestion Avoidance

- **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



Why is TCP fair?

- Two competing sessions:
 - Additive increase gives slope of 1, as throughput increases
 - Multiplicative decrease decreases throughput proportionally



Fairness (more)

- Fairness and UDP
 - 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
- Fairness, parallel TCP connections
 - 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 10 TCPs, gets $R/2$

Chapter 3: Summary

- Principles behind transport layer services:
 - Multiplexing, demultiplexing
 - Reliable data transfer
 - Flow control
 - Congestion control
- Instantiation, implementation in the Internet
 - UDP
 - TCP

Next:

- Leaving the network “edge” (application, transport layers)
- Into the network “core”