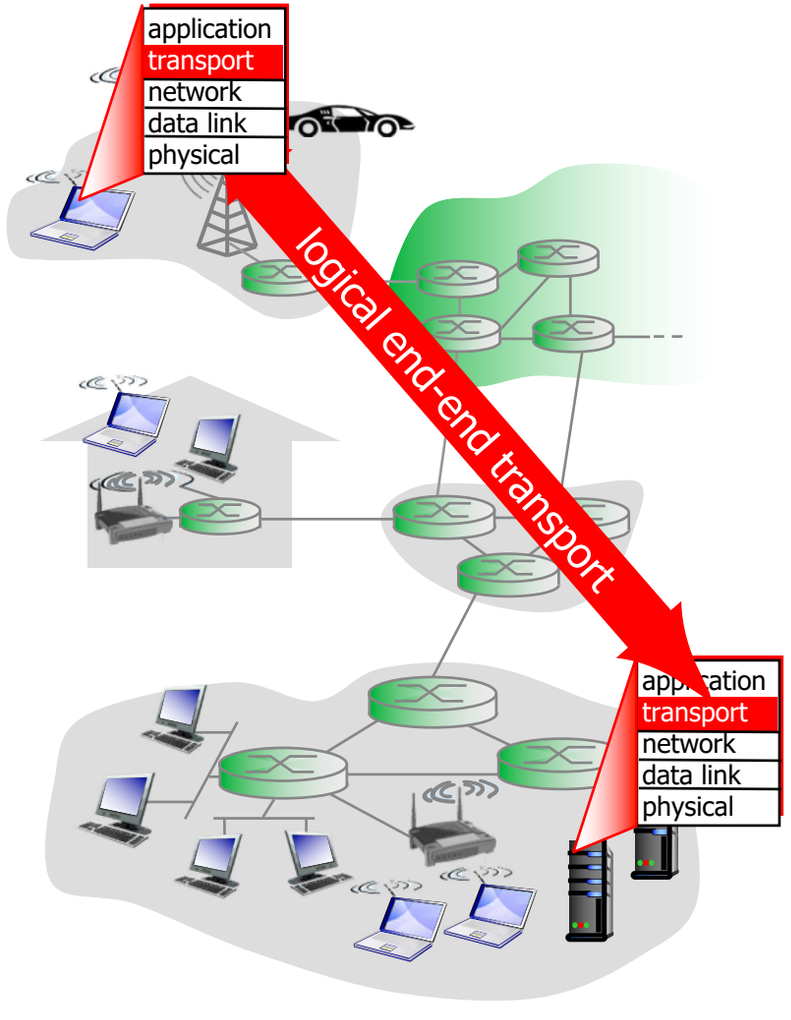
University of Duisburg-Essen
Networked Embedded Systems Group
Institute for Computer Science and
Business Information Systems (ICB)
Schützenbahn 70
D-45127 Essen, Germany

Kommunikationsnetze 2
4 – Transport

Prof. Dr. Pedro José Marrón

Transport Services and Protocols

- Goal: provide logical communication between application processes running on different hosts
- Transport protocols run in end systems
  - Sender side: breaks application messages into segments and passes them to the network layer
  - Receiver side: reassembles segments into messages and passes them to application layer
- More than one transport protocol available to applications
  - Internet: TCP and UDP

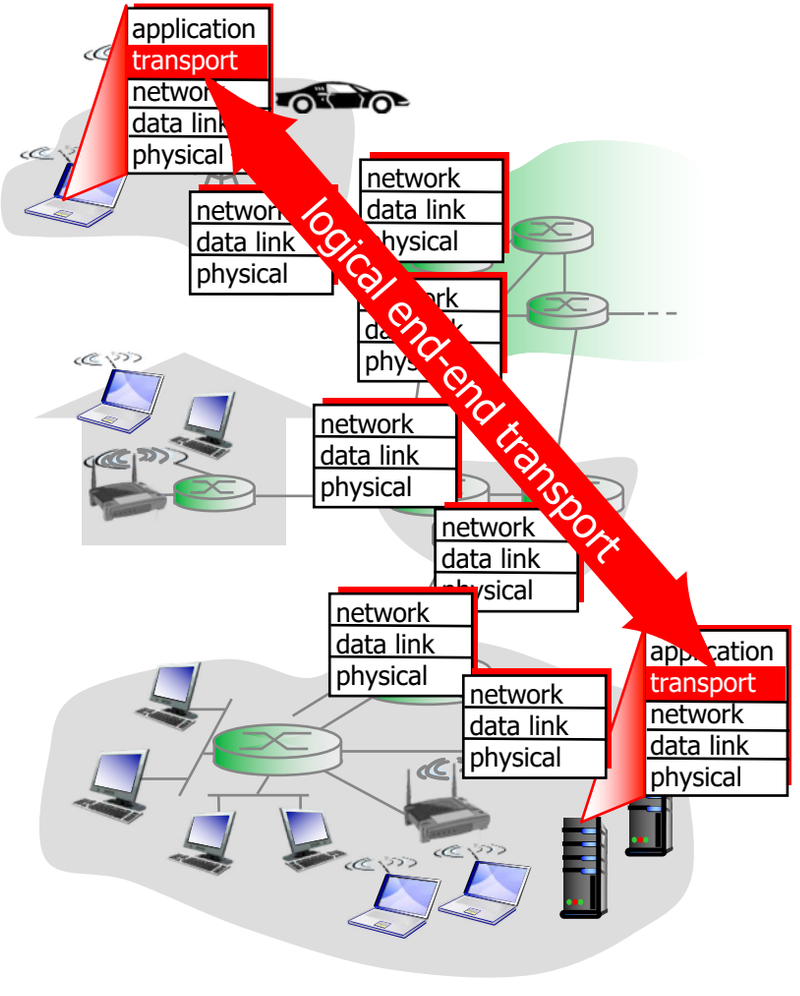


Transport vs. Network Layer

- Network layer: logical communication between hosts
  - Exists in hosts and routers
  - Routes data through the network
  - IP addresses identify the destination host
- Transport layer: logical communication between processes
  - Exists only in hosts
  - Relies on network layer services for reaching the destination host
  - Port numbers identify the destination process on the destination host

Internet Transport-Layer Protocols

- Reliable, in-order delivery (TCP)
  - Congestion control
  - Flow control
  - Connection setup
- Unreliable, unordered delivery (UDP)
  - Basic “best-effort” IP
- Services not available
  - Delay guarantees
  - Bandwidth guarantees



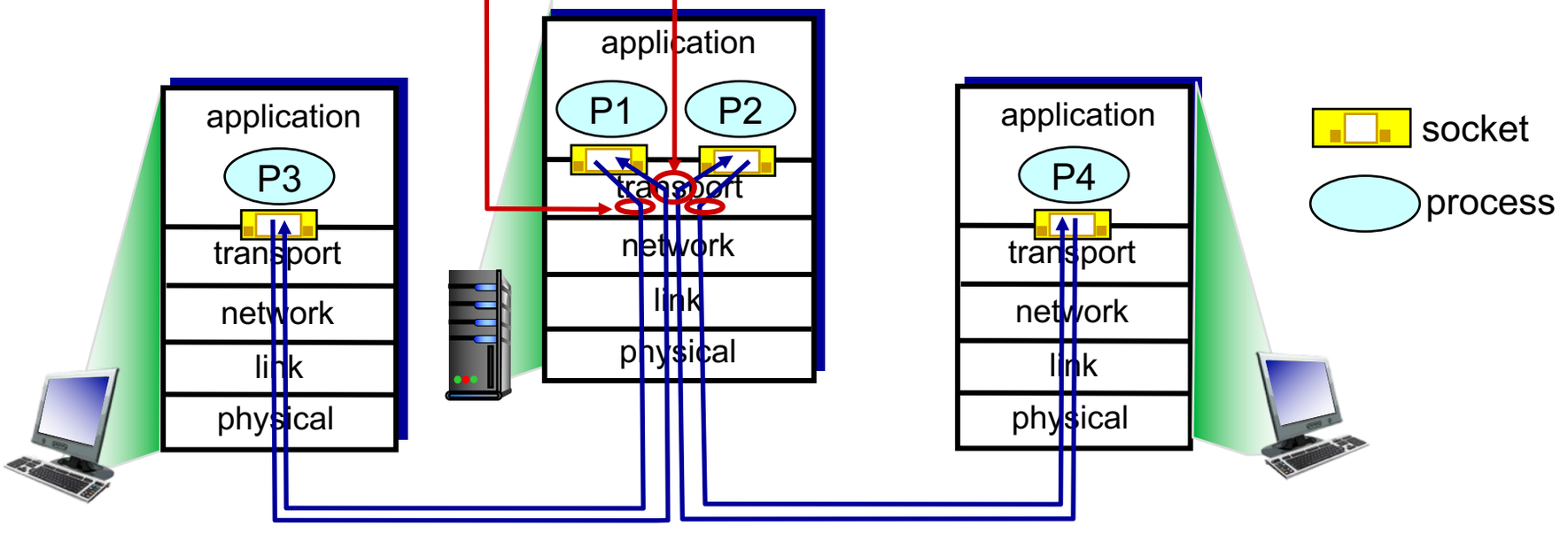
Multiplexing/Demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

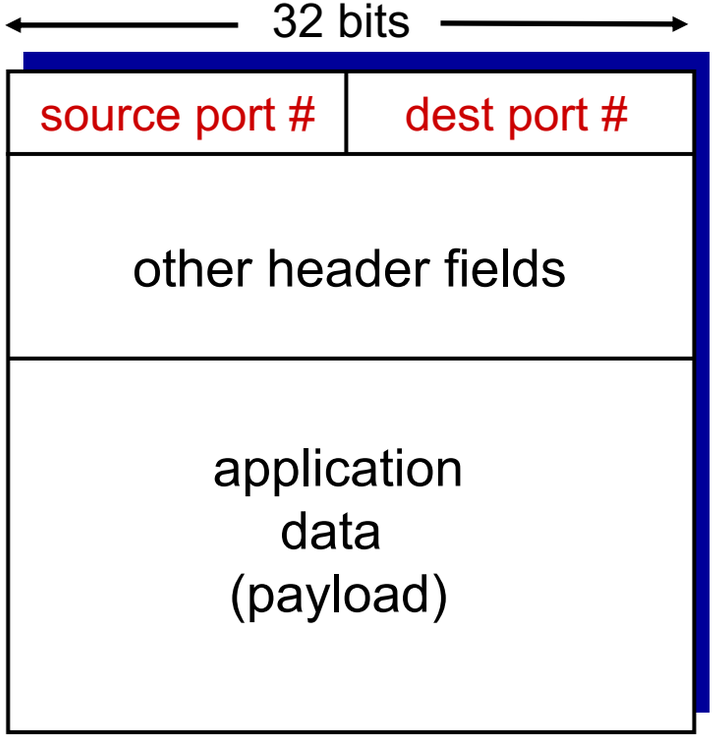
demultiplexing at receiver:

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 and destination port number
- Host uses IP addresses and port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless Demultiplexing

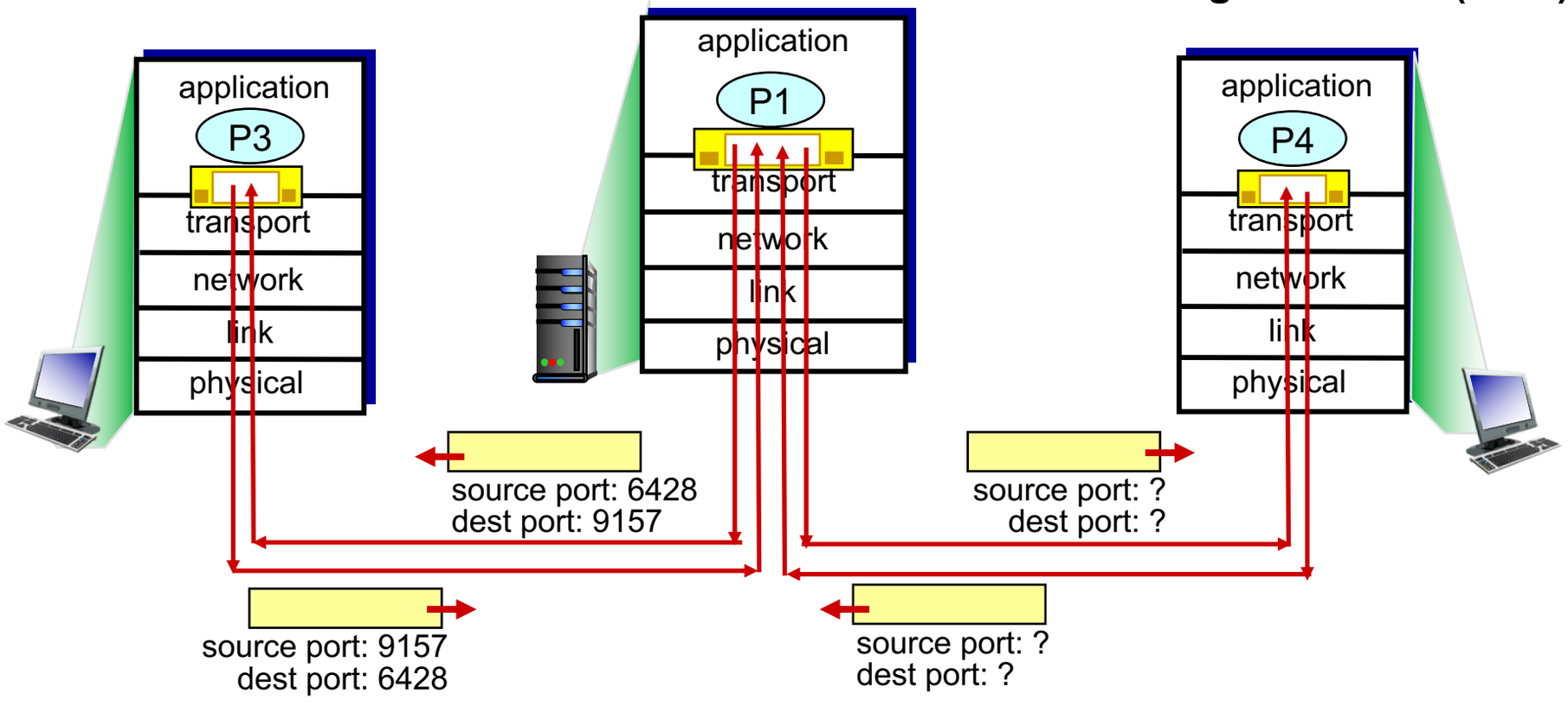
- Created socket has host-local port number:
  - DatagramSocket mySocket1 = new DatagramSocket(12543);
- When creating datagram to send into UDP socket, must specify
  - Destination IP address and destination port number
- When host receives UDP segment:
  - Checks destination port number in segment
  - Directs UDP segment to socket with that port number
  - IP datagrams with same destination port number but different source IP addresses and/or source port numbers will be directed to same socket at the destination

Connectionless Demultiplexing: Example

DatagramSocket mySocket2 = new DatagramSocket (9157);

DatagramSocket serverSocket = new DatagramSocket (6428);

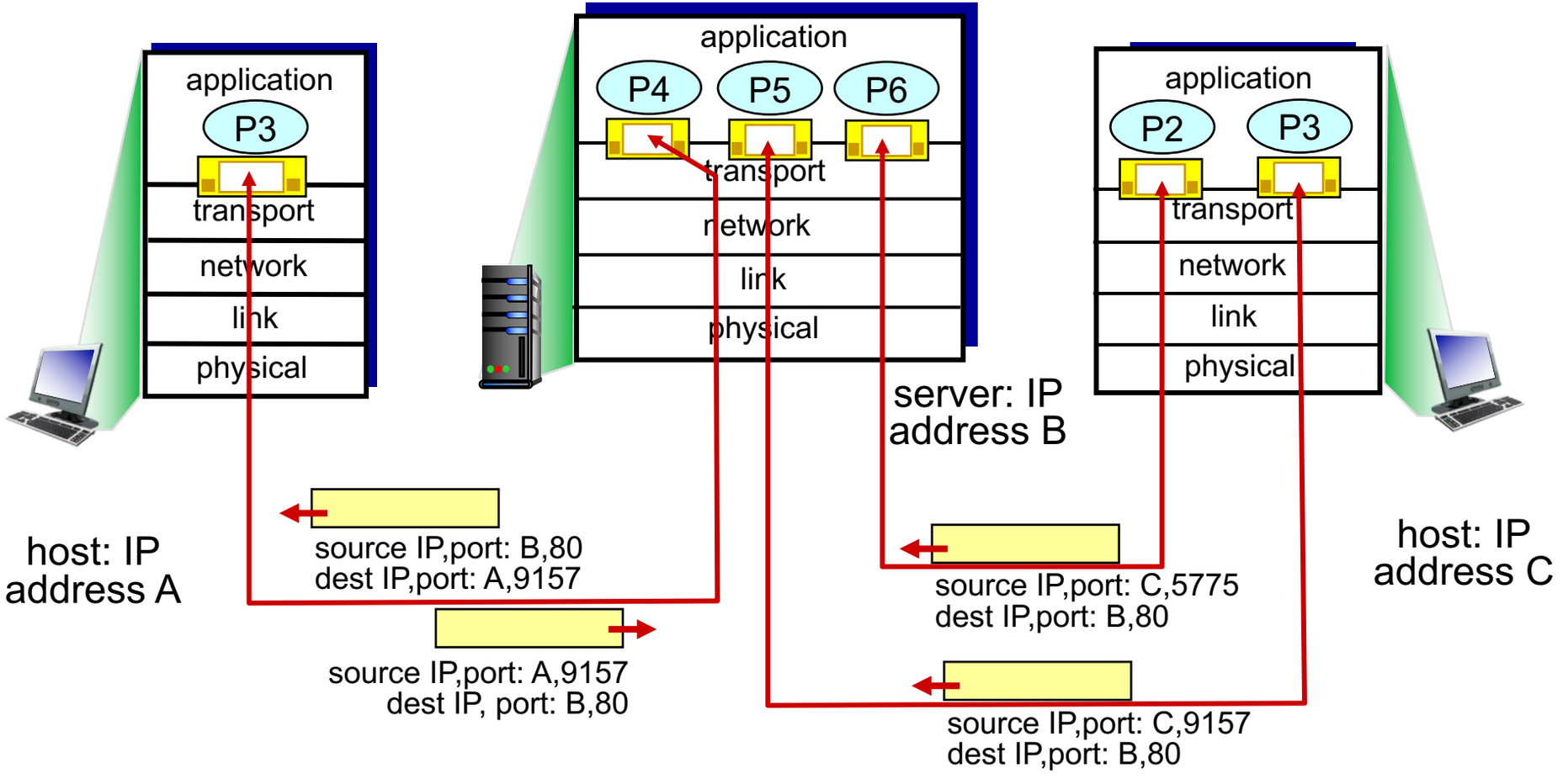
DatagramSocket mySocket1 = new DatagramSocket (5775);



Connection-Oriented Demultiplexing

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

Connection-Oriented Demultiplexing: Example

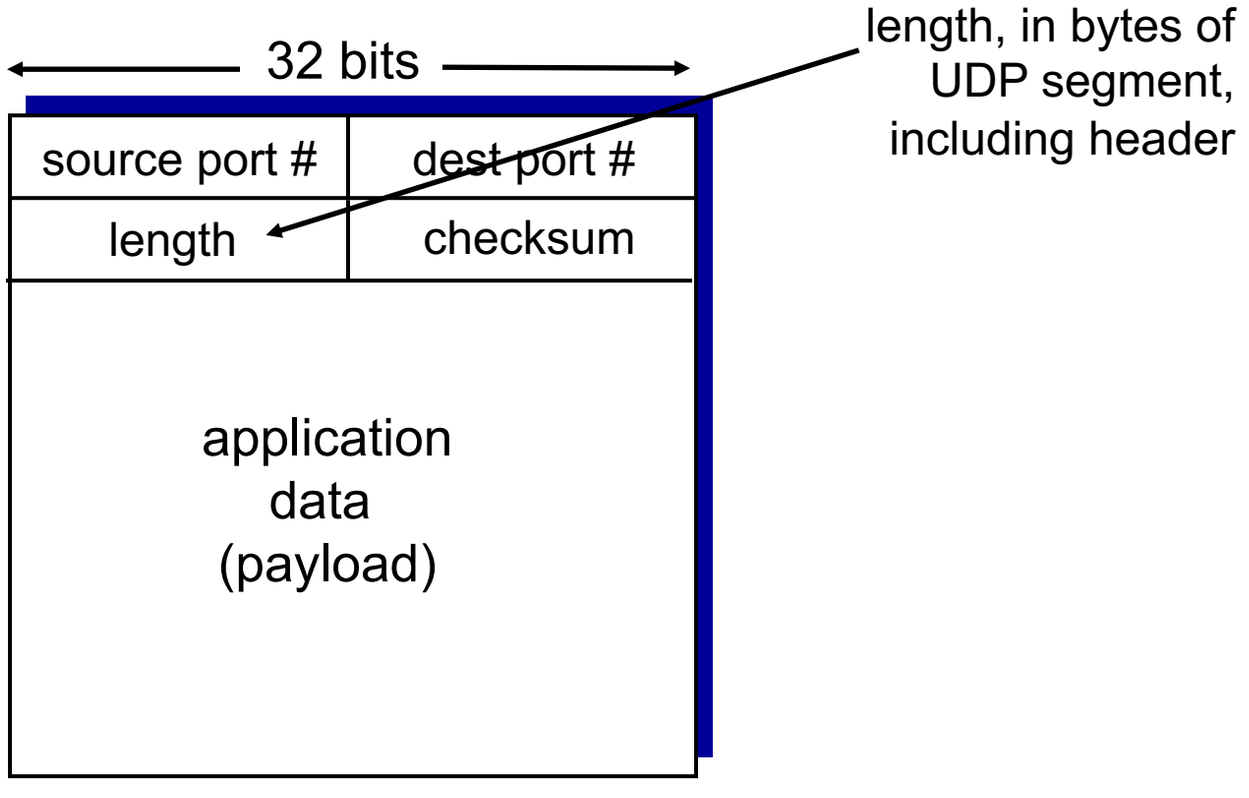


three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

UDP: User Datagram Protocol [RFC 768]

- "Bare bones" Internet transport protocol
- "Best effort" service:
  - Segments may be lost
  - Segments may be delivered out-of-order
- Connectionless
  - No handshaking between UDP sender and receiver
  - Each UDP segment handled independently
- UDP uses
  - Multimedia streaming (not always)
  - DNS
  - SNMP
- Reliability can still be added over UDP at the application layer

UDP Segment Header



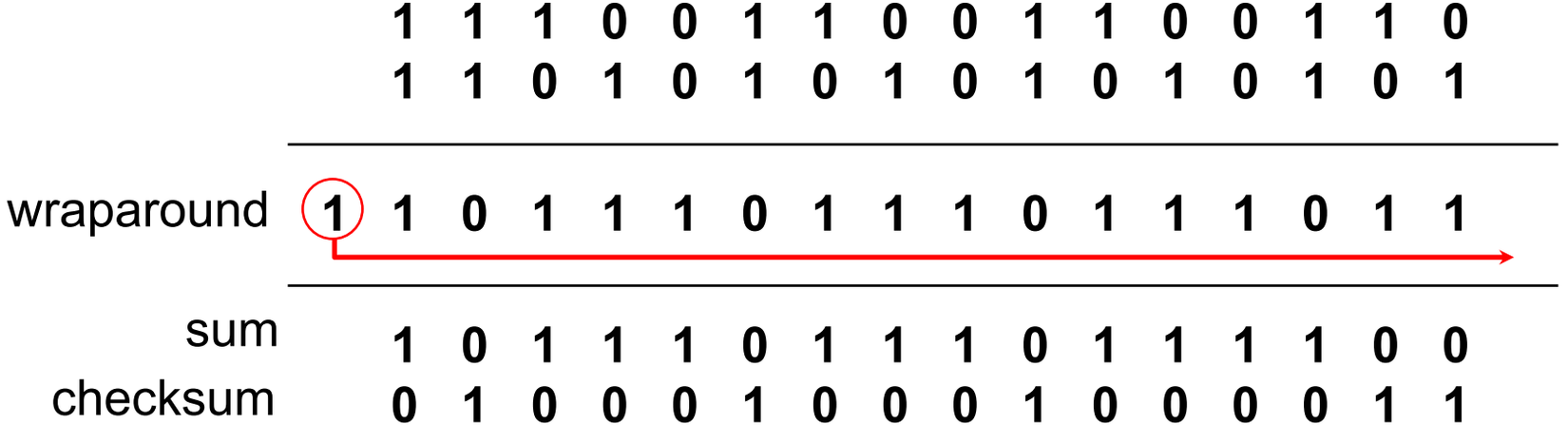
UDP segment format

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
  - Compute checksum of received segment
  - Check if computed checksum equals checksum field value:
    - NO – error detected
    - YES – no error detected… but still possible

Checksum Example

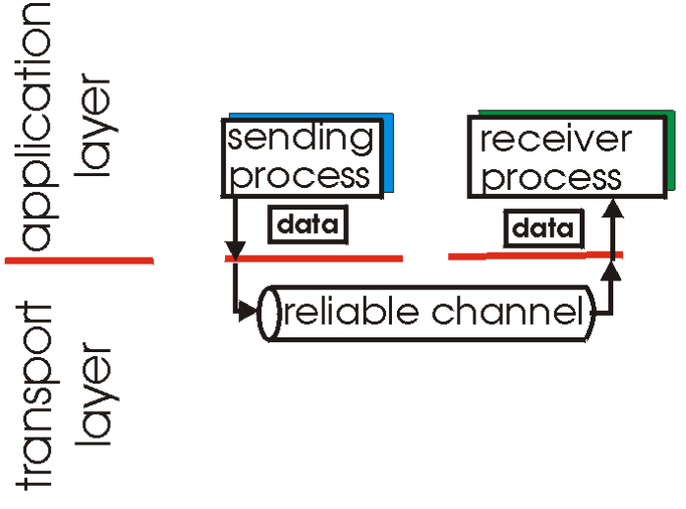
example: add two 16-bit integers



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

Principles of Reliable Data Transfer

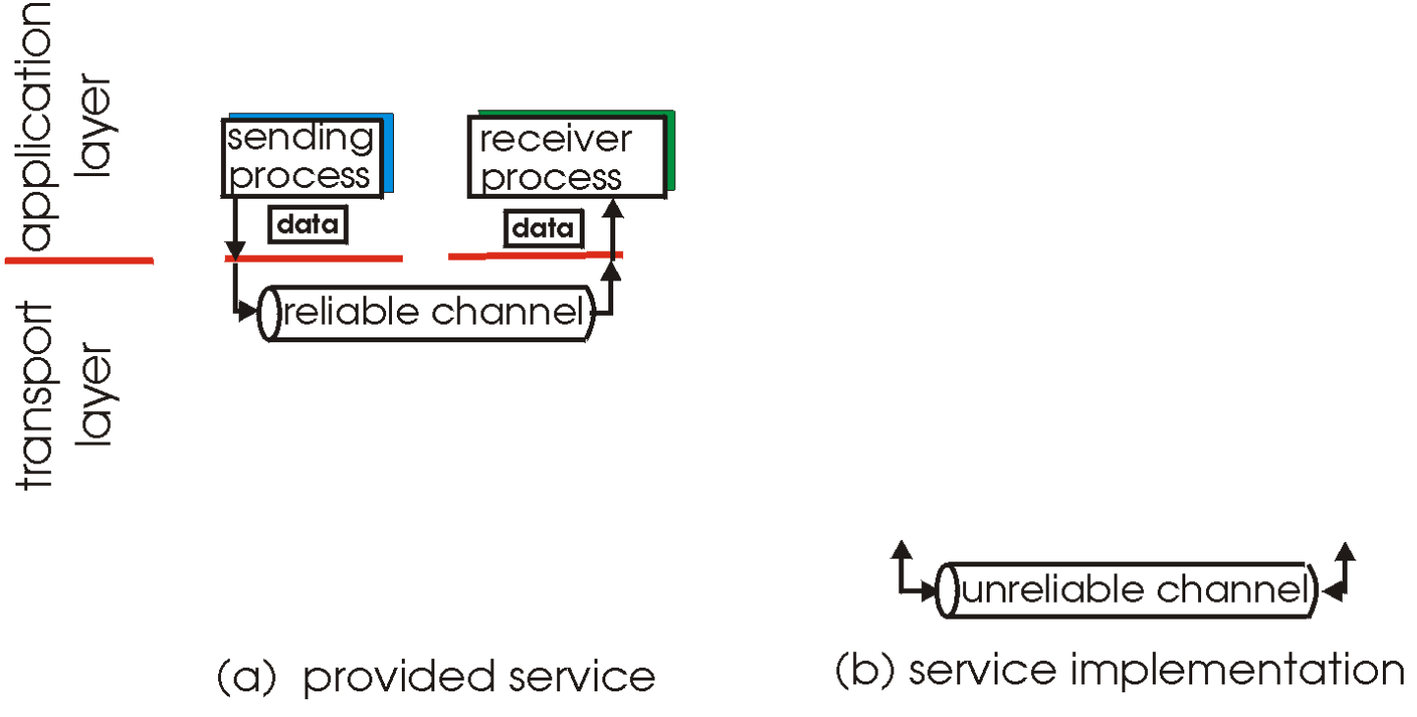
- Important in application, transport, link layers
- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)



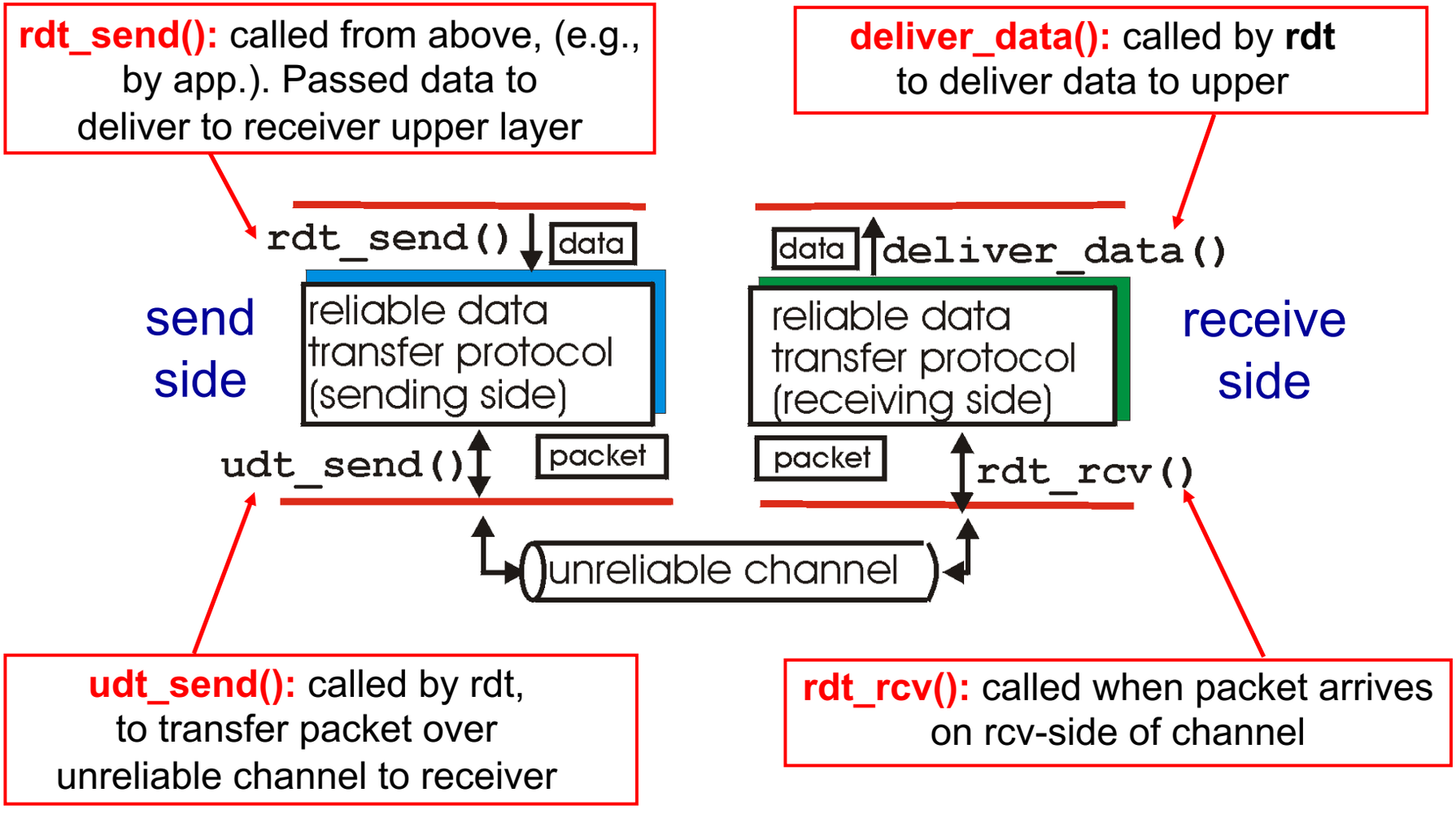
(a) provided service

Principles of Reliable Data Transfer

- Important in application, transport, link layers
- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

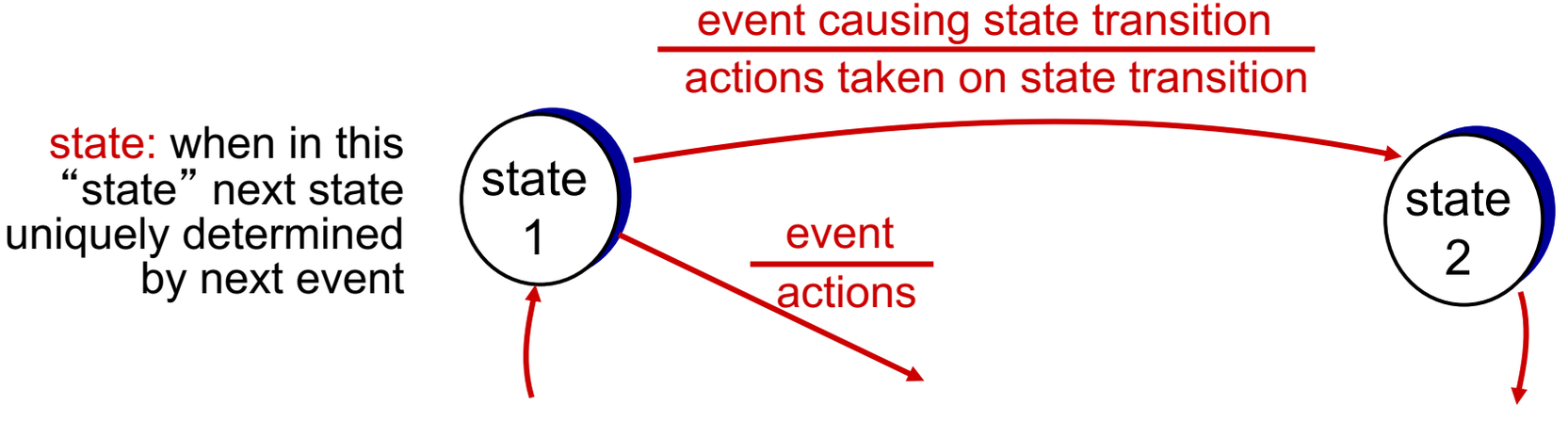


Reliable Data Transfer: Getting Started



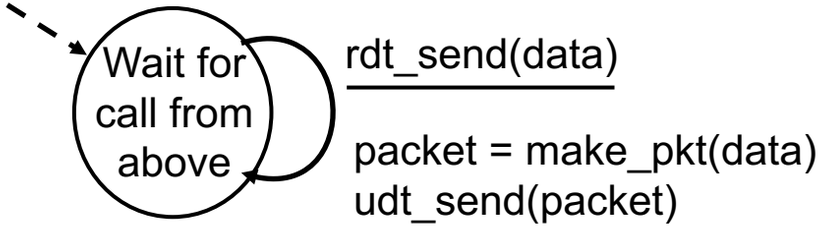
Reliable Data Transfer: Approach

- We are going to
  - Incrementally develop sender and receiver sides of a reliable data transfer protocol (rdt)
  - Consider only unidirectional data transfer
    - But control info will flow on both directions!
  - Use finite state machines (FSM) to specify sender and receiver

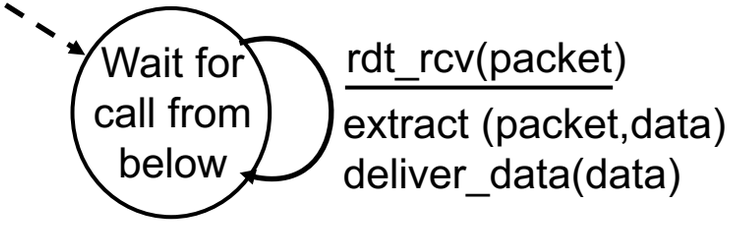


rdt 1.0: Reliable Transfer over Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- Separate FSMs for sender and receiver
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



sender



receiver

rdt 2.0: Channel with Bit Errors

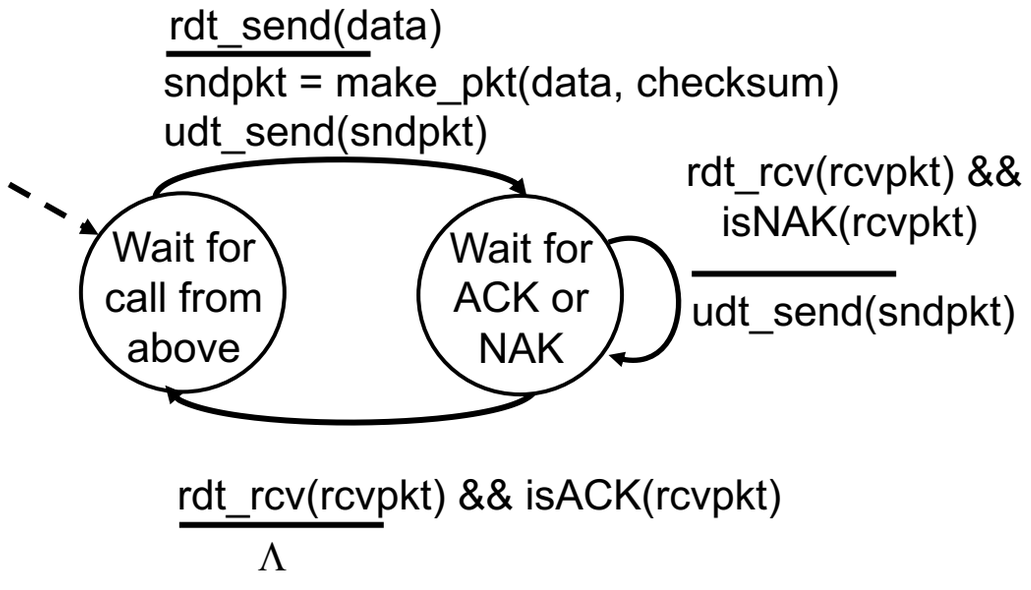
- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- How to recover from errors?

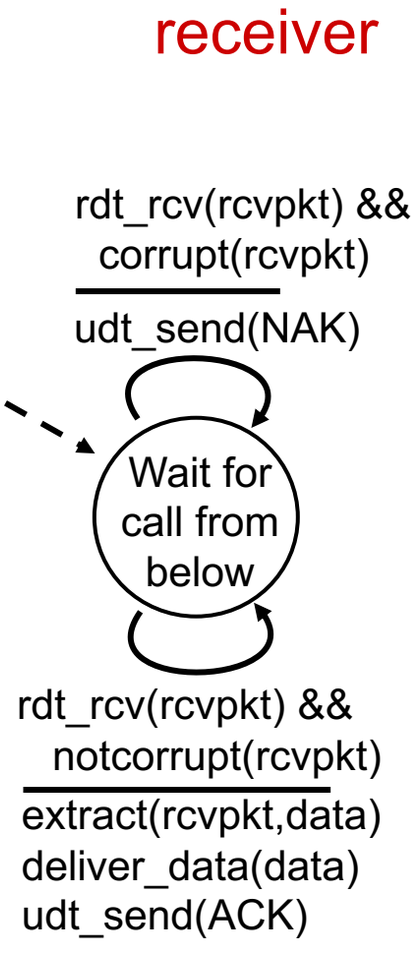
How do humans recover from "errors" during conversation?

rdt 2.0: Channel with Bit Errors

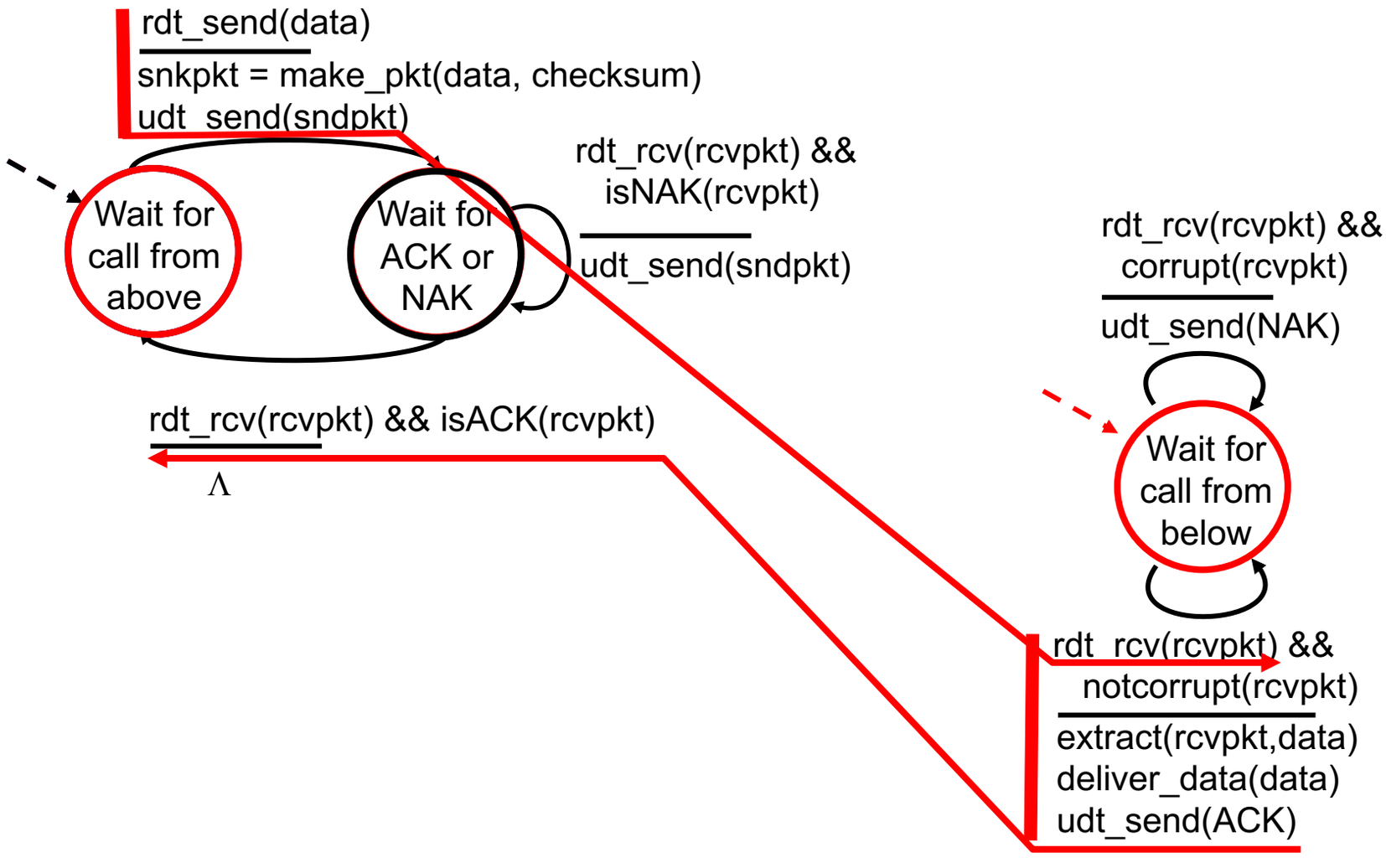
- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- How to recover from errors?
  - Acknowledgements (ACKs): receiver explicitly tells sender that a packet was received OK
  - Negative acknowledgments (NAKs): receiver explicitly tells sender that a packet had errors
- New mechanisms in rtd 2.0 (vs. rtd 1.0):
  - Error detection
  - Feedback: control messages (ACK, NAK) from receiver to sender

rdt 2.0: FSM Specification

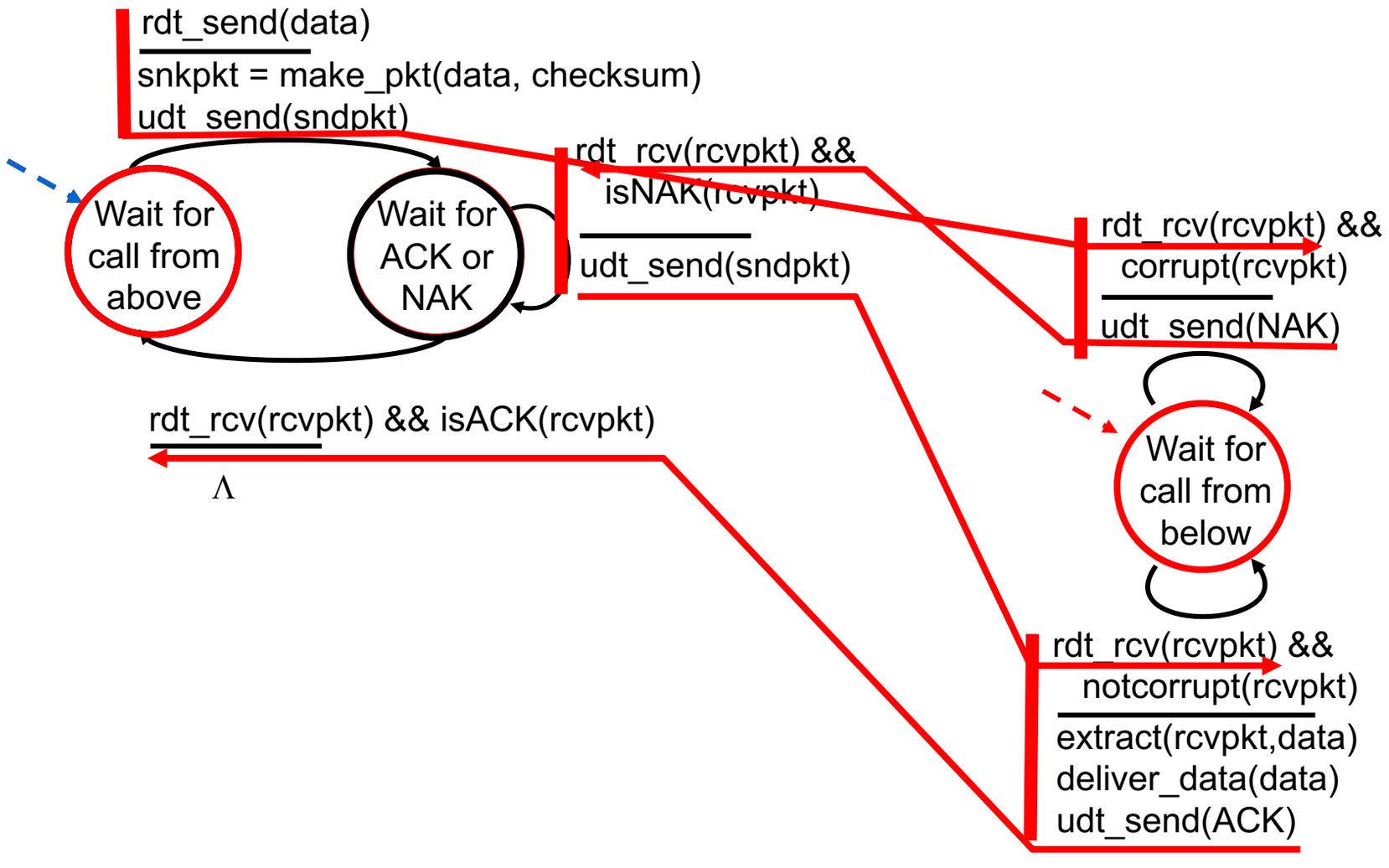




rdt 2.0: Operation without Errors



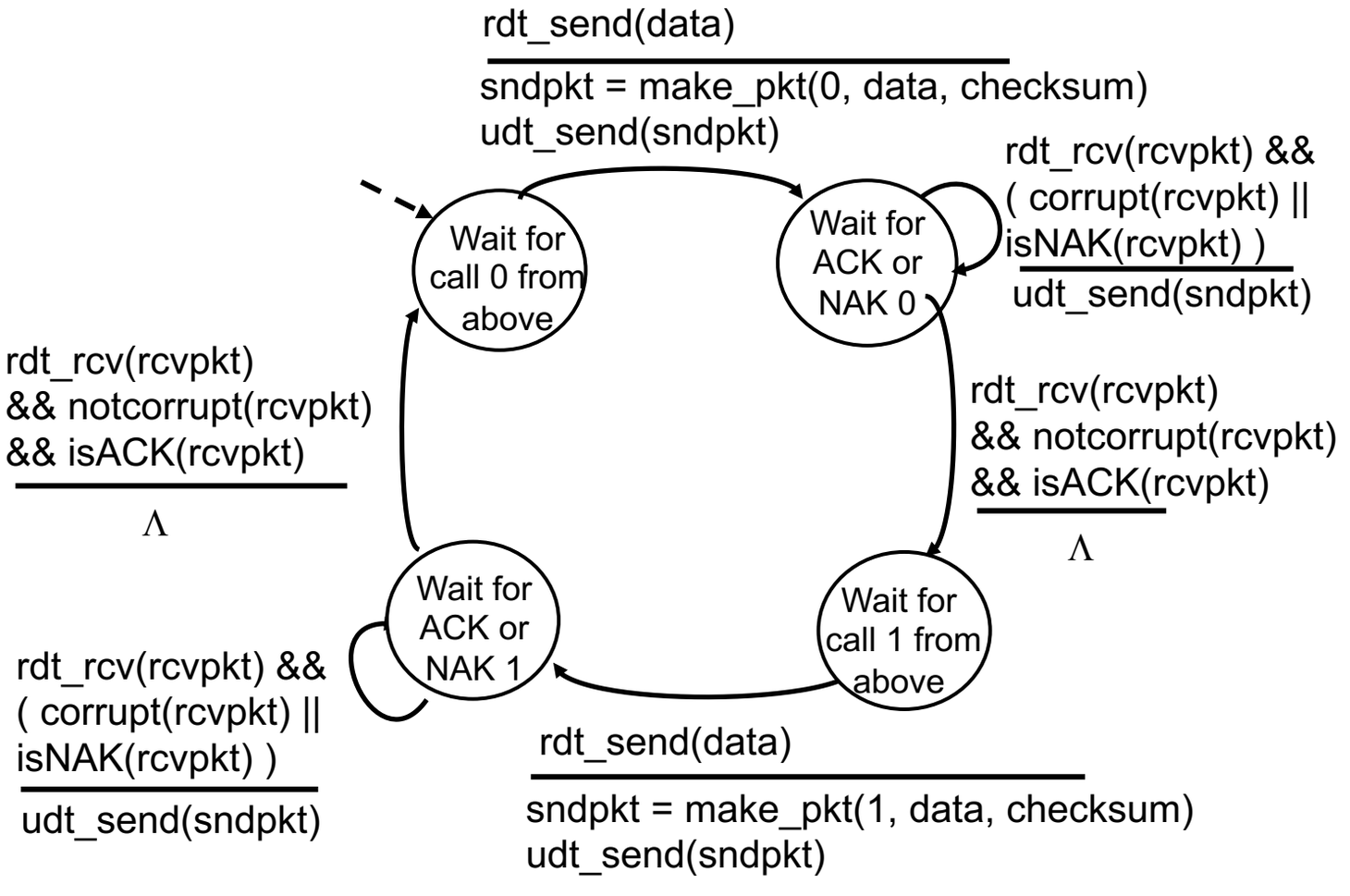
rdt 2.0: Error Scenario



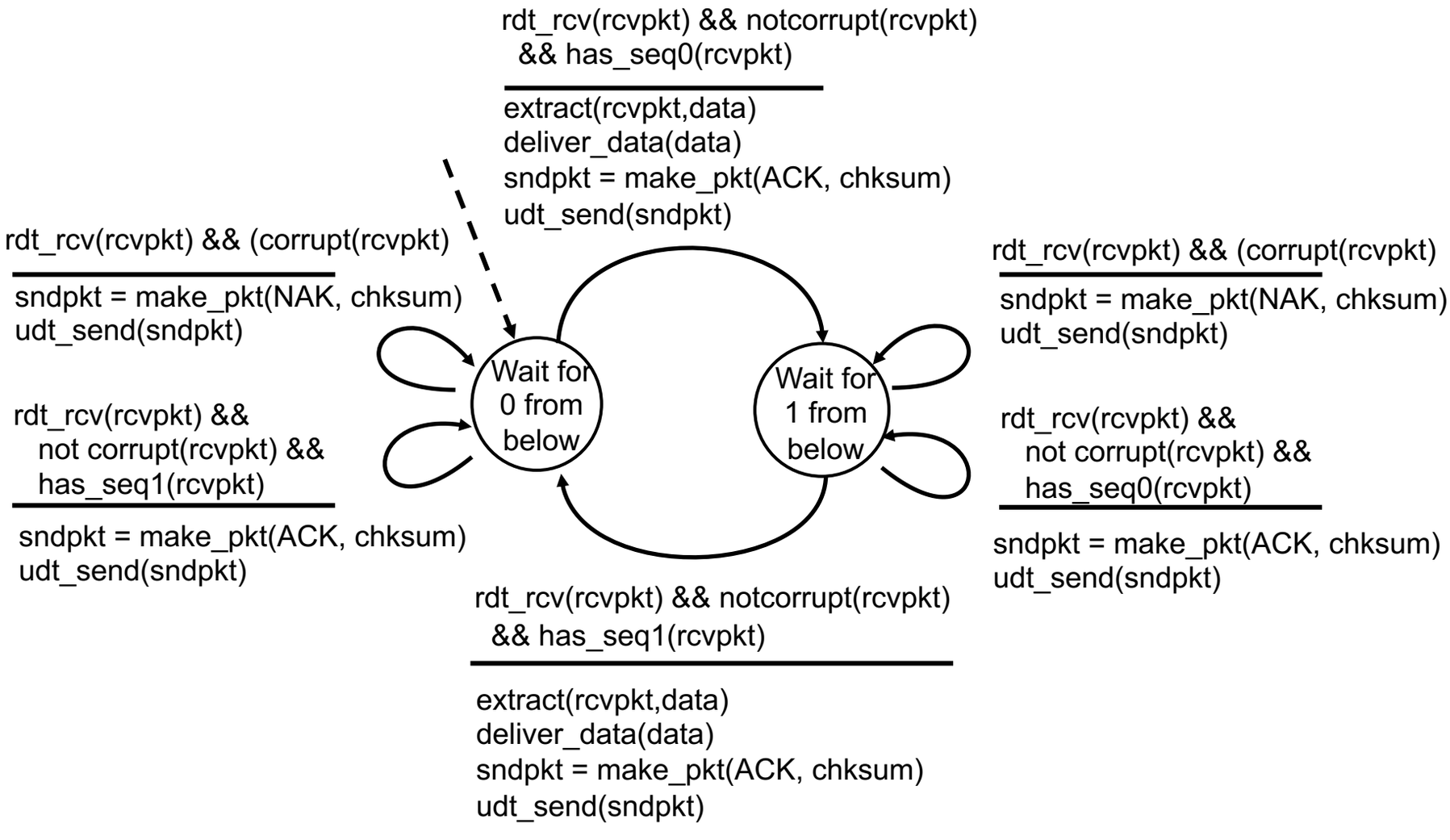
rdt 2.0: A Flaw!

- What happens if ACK/NAK corrupted?
  - Sender does not know what happened at receiver
  - Cannot just retransmit: possible duplicate
- Handling duplicates
  - Sender retransmits current packet if ACK/NAK corrupted
  - Sender adds a sequence number to each packet
  - Receiver discards duplicate packets
- Stop and wait
  - Sender sends one packet
  - Waits for receiver response

rdt 2.1: Sender Handling Corrupted ACK/NAKs



rdt 2.1: Receiver Handling Corrupted ACK/NAKs



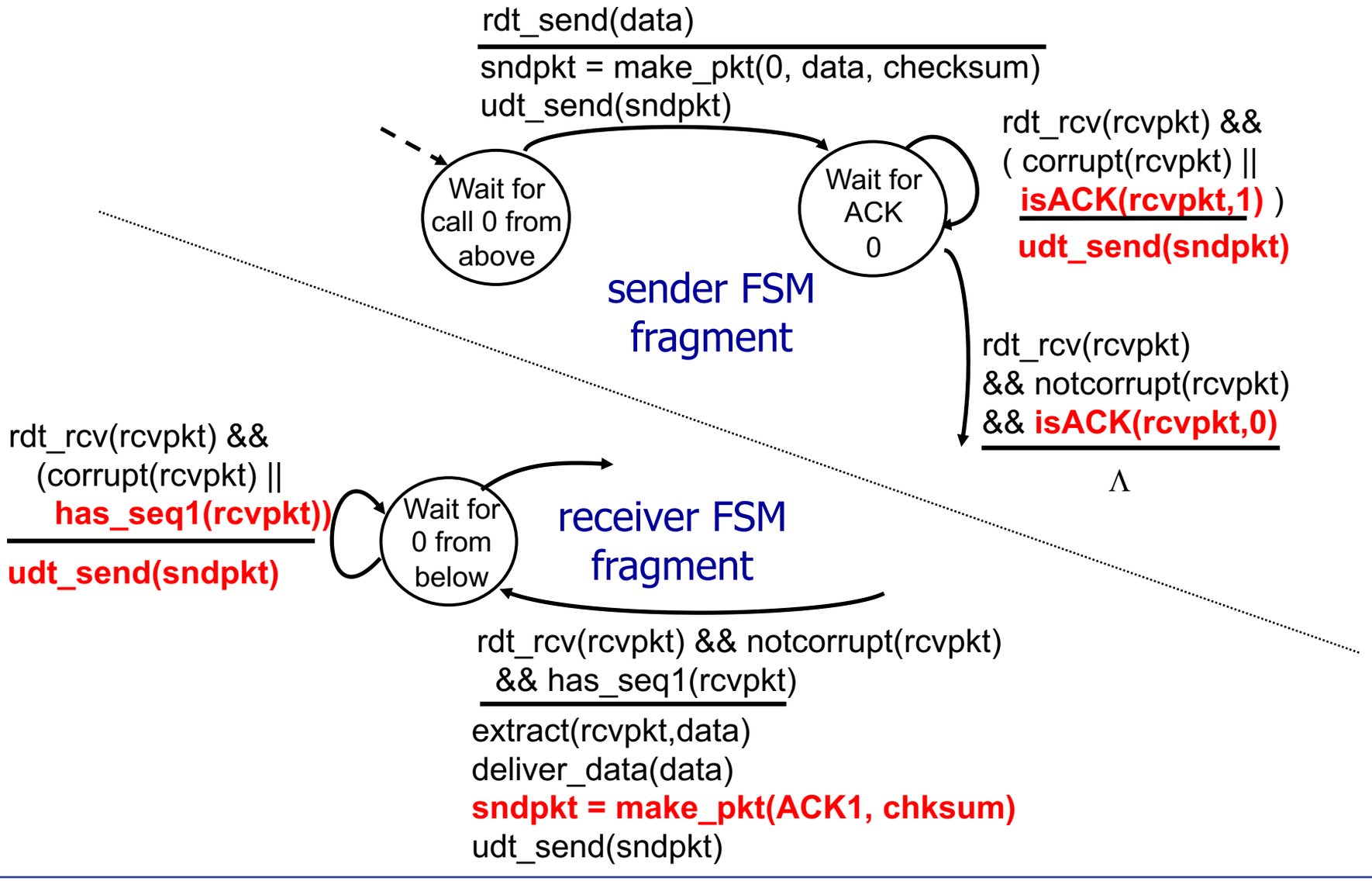
rdt 2.1: Discussion

- Sender
  - Sequence number added to packet
  - Two sequence numbers (0,1) suffice… why?
  - Must check if received ACK/NAK corrupted
  - Twice as many states
    - Need to remember which sequence number expected
- Receiver
  - Must check if received packet is duplicate
    - Need to remember which sequence number expected
  - Note: Receiver cannot know if last ACK/NAK was received at sender

rdt 2.2: A NAK-free Protocol

- Same functionality as rtd 2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last packet received OK
  - Receiver must explicitly include sequence number of packet being ACKed
- Duplicate ACK at sender results in same action as NAK: retransmit current packet

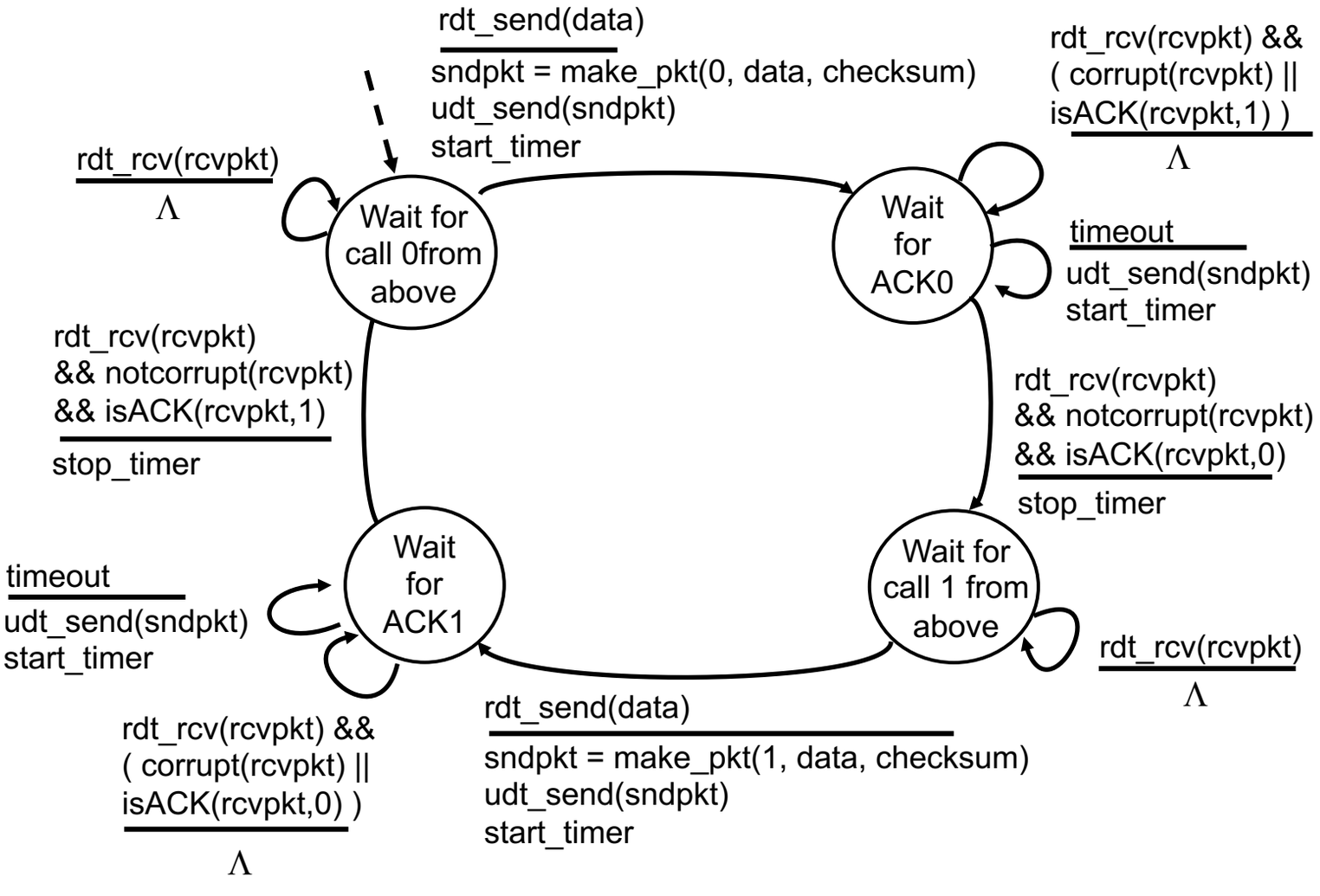
rdt 2.2: Sender and Receiver Excerpts



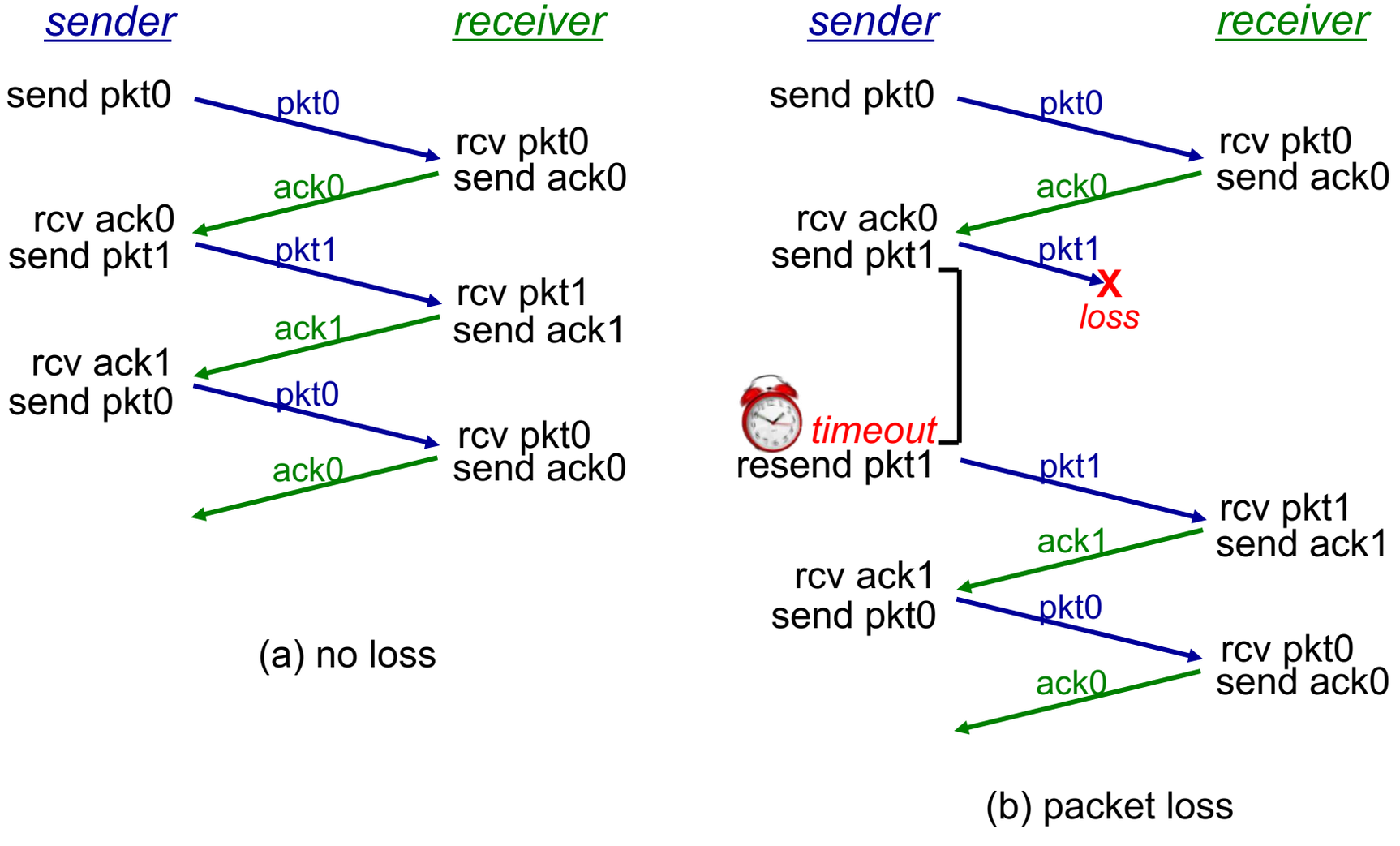
rdt 3.0: Channels with Errors and Loss

- New assumption:
  - Underlying channel can also lose packets (data or ACKs)
    - Checksum, sequence numbers, ACKs, retransmissions can help
- Approach:
  - Sender waits “reasonable” amount of time for ACK
    - Retransmits if no ACK received in time
    - If packet (or ACK) just delayed (not lost)
      - Retransmission will be duplicate (but sequence numbers already handle it)
      - Receiver must specify sequence number of the packet being ACKed
    - Requires a countdown timer

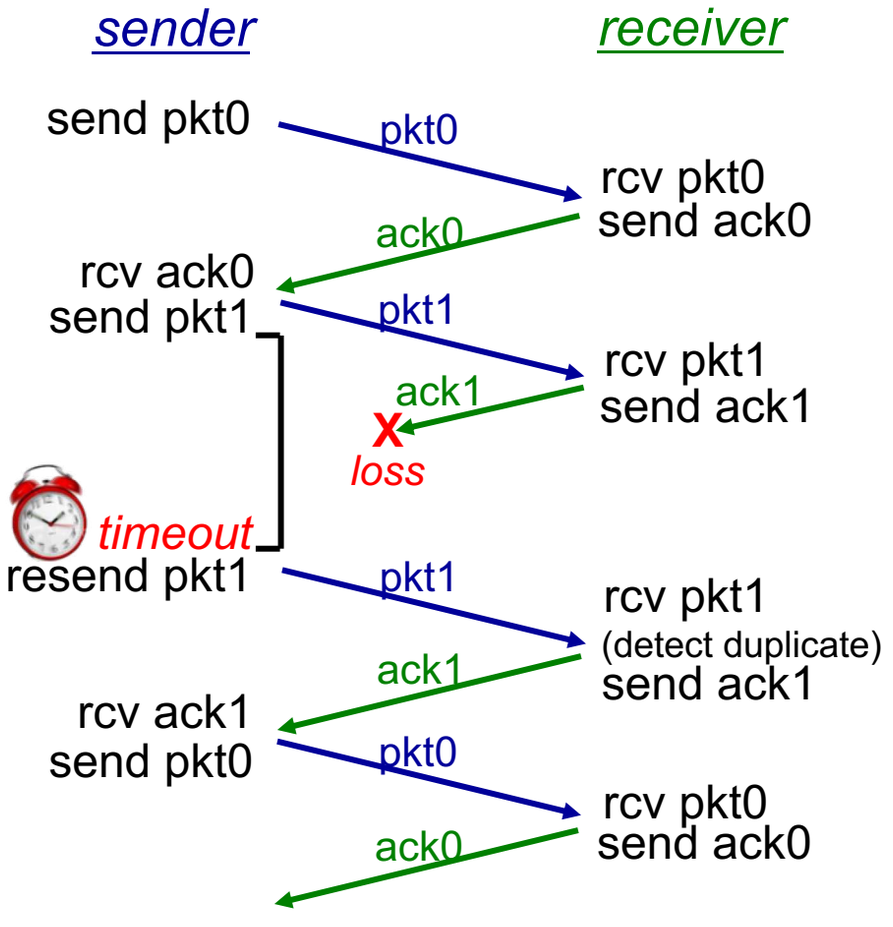
rdt 3.0: Sender



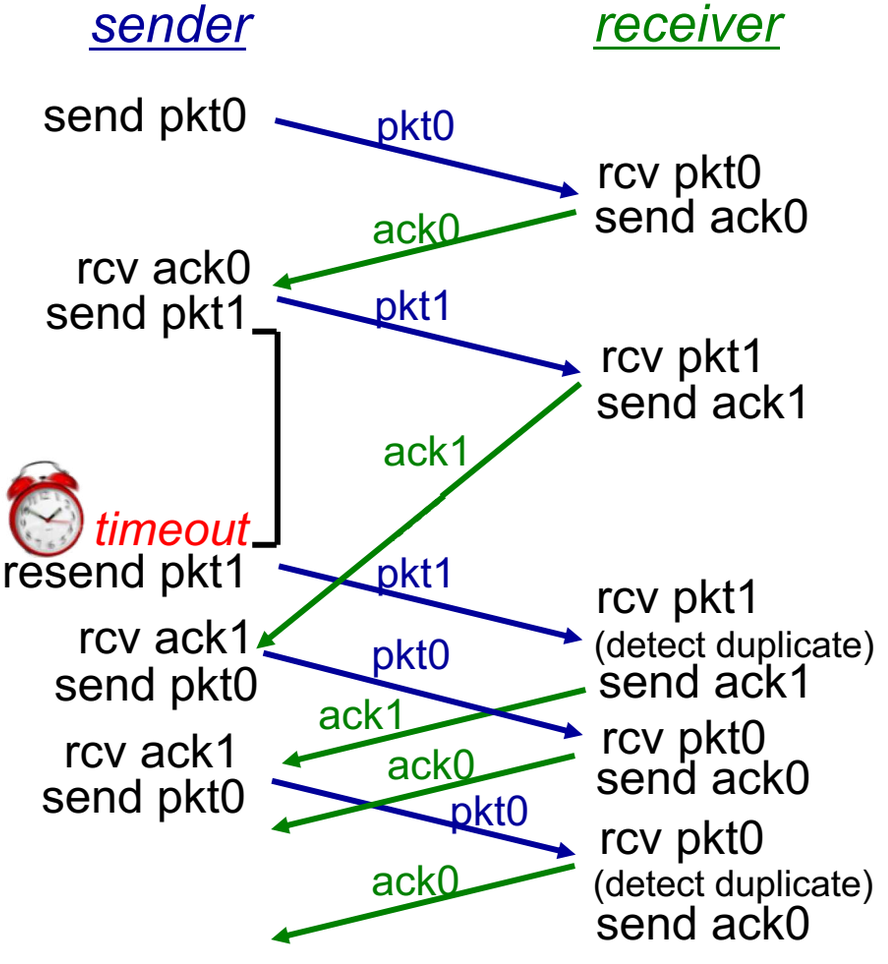
rdt 3.0: In Action (1/2)



rdt 3.0: In Action (2/2)



(c) ACK loss



(d) premature timeout/ delayed ACK

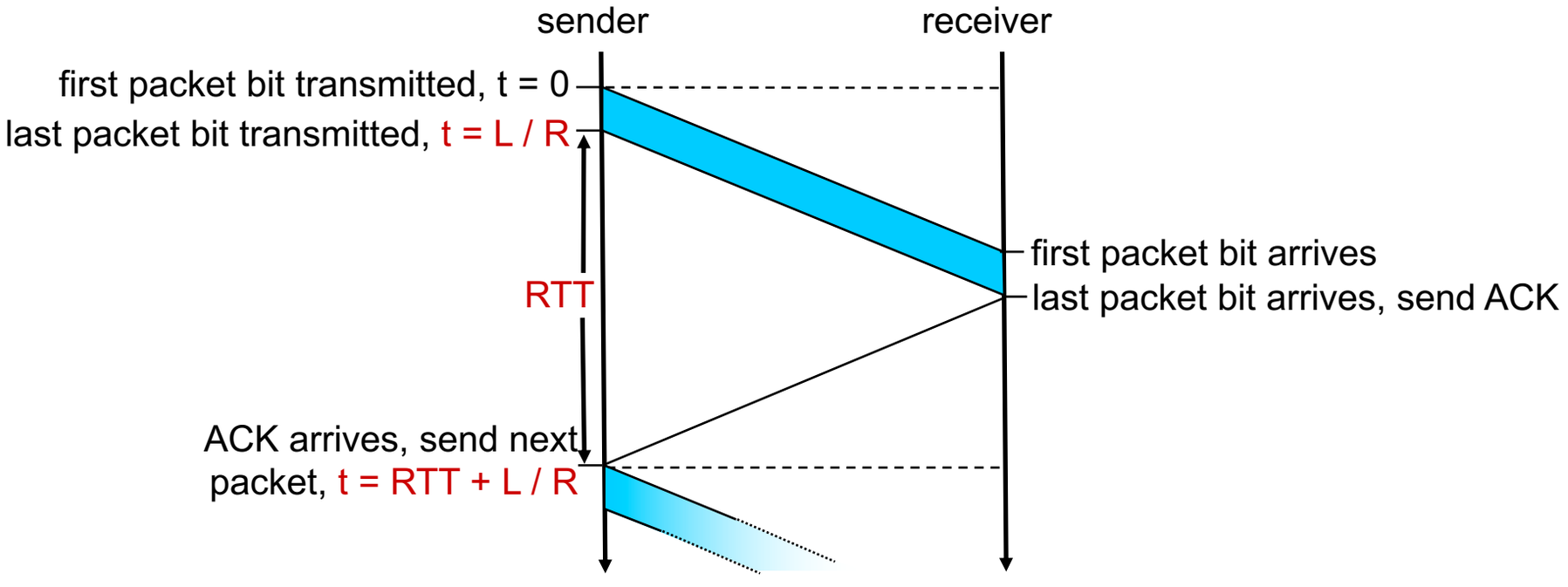
rdt 3.0: Performance Example

- E.g., 1 Gbps link, 15 ms delay and 8000 bit packet
  - D_{trans} = L / R = 8000 bits / 10^9 bits/sec = 8 microsecs

- Utilisation (U_{sender}): fraction of time sender busy sending
  - $U_{sender} = D_{trans} / (RTT + D_{trans}) = .008 / 30.008 = 0.00027$
  - If RTT = 30 ms, with 1 KB packet every 30 ms: 33kB/sec throughput over 1 Gbps link

- Network protocol limits use of physical resources!

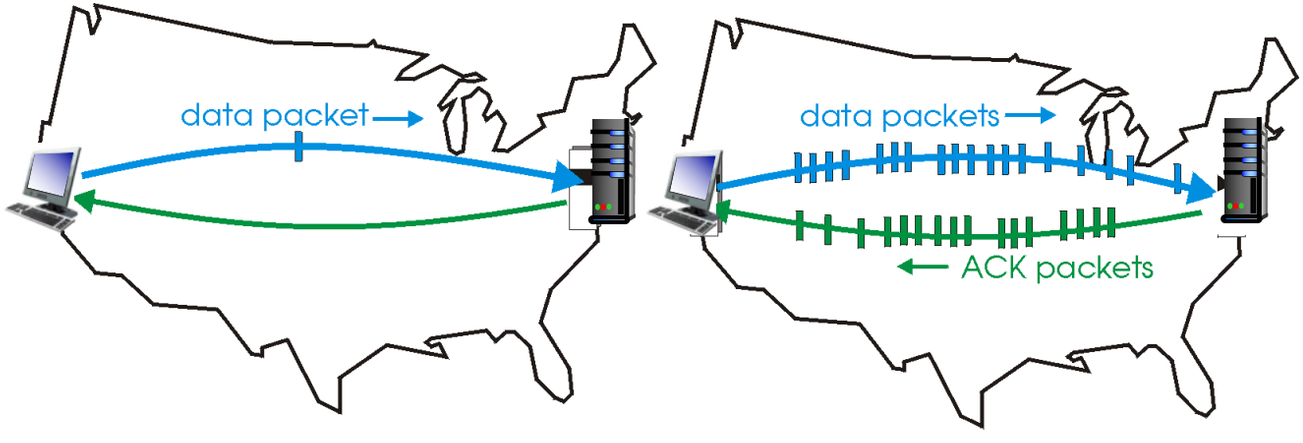
rdt 3.0: Stop-and-Wait Operation



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

Pipelined Protocols

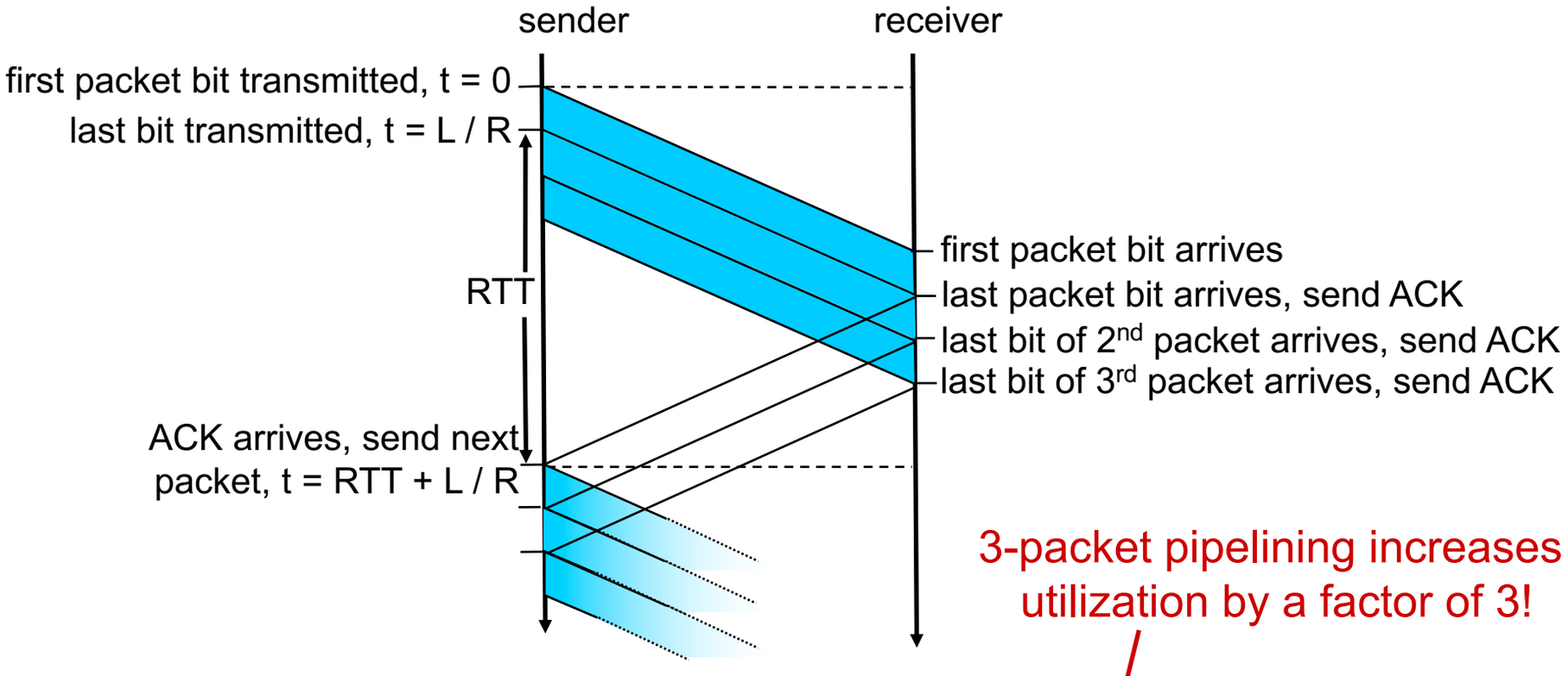
- Pipelining: sender allows multiple, yet-to-be-acknowledged packets
  - Range of sequence numbers must be increased
  - Buffering at sender and/or receiver



(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

Pipelining: Increased Utilisation



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

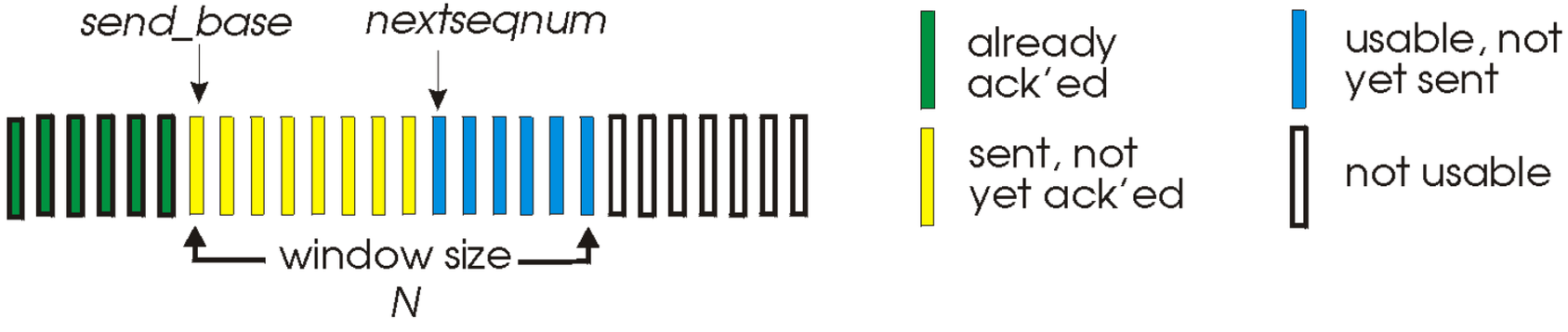
$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined Protocols: Overview

- Go-Back-N
  - Sender can have up to N unacked packets in pipeline
  - Receiver only sends cumulative ack
    - Does not ack packet if there is 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
  - Receiver sends individual ack for each packet
  - Sender maintains timer for each unacked packet
    - When timer expires, retransmit only that unacked packet

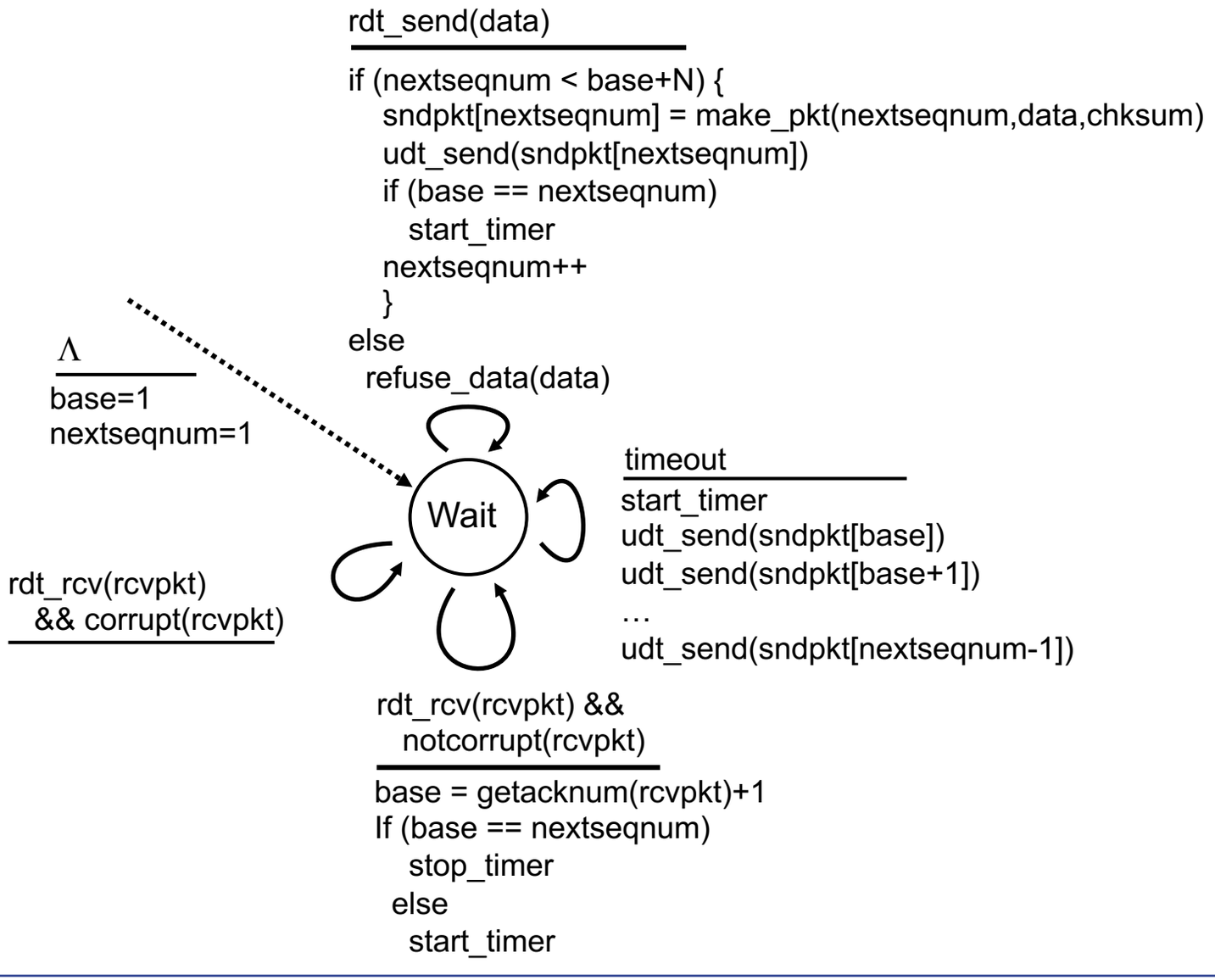
Go-Back-N: Sender

- k-bit sequence number in packet header
- Window of up to N, consecutive unacked packets allowed

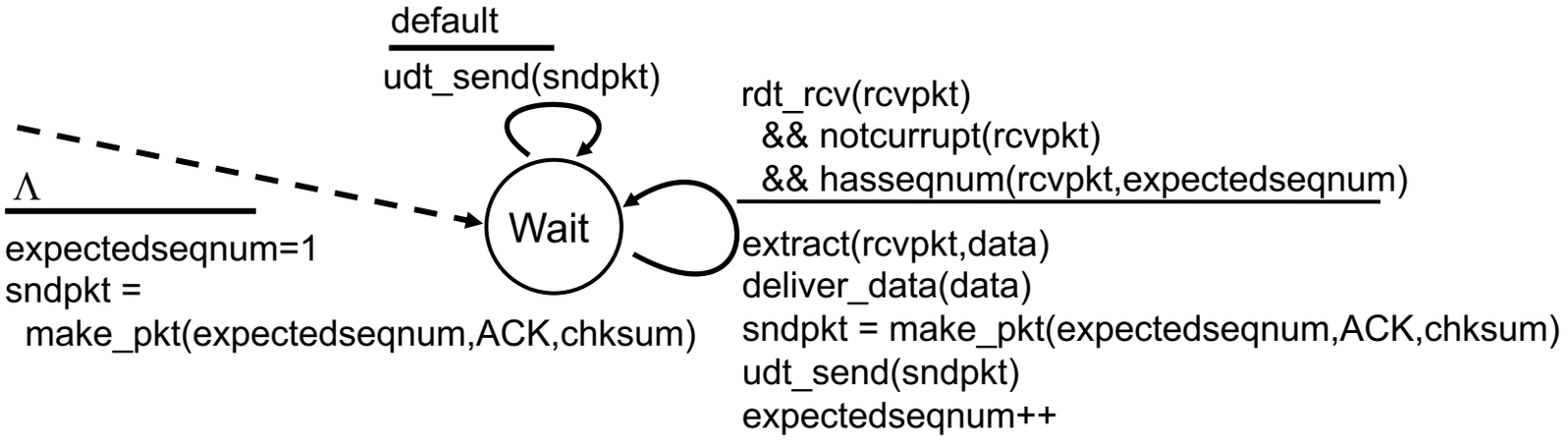


- ACK(n): ACKs all packets up to, including sequence number n
  - "Cumulative ACK"
  - May receive duplicate ACKs
- Timer for older in-flight packet
- Timeout(n): Retransmit packet n and all higher sequence number packets in window

Go-Back-N: Sender Extended FSM

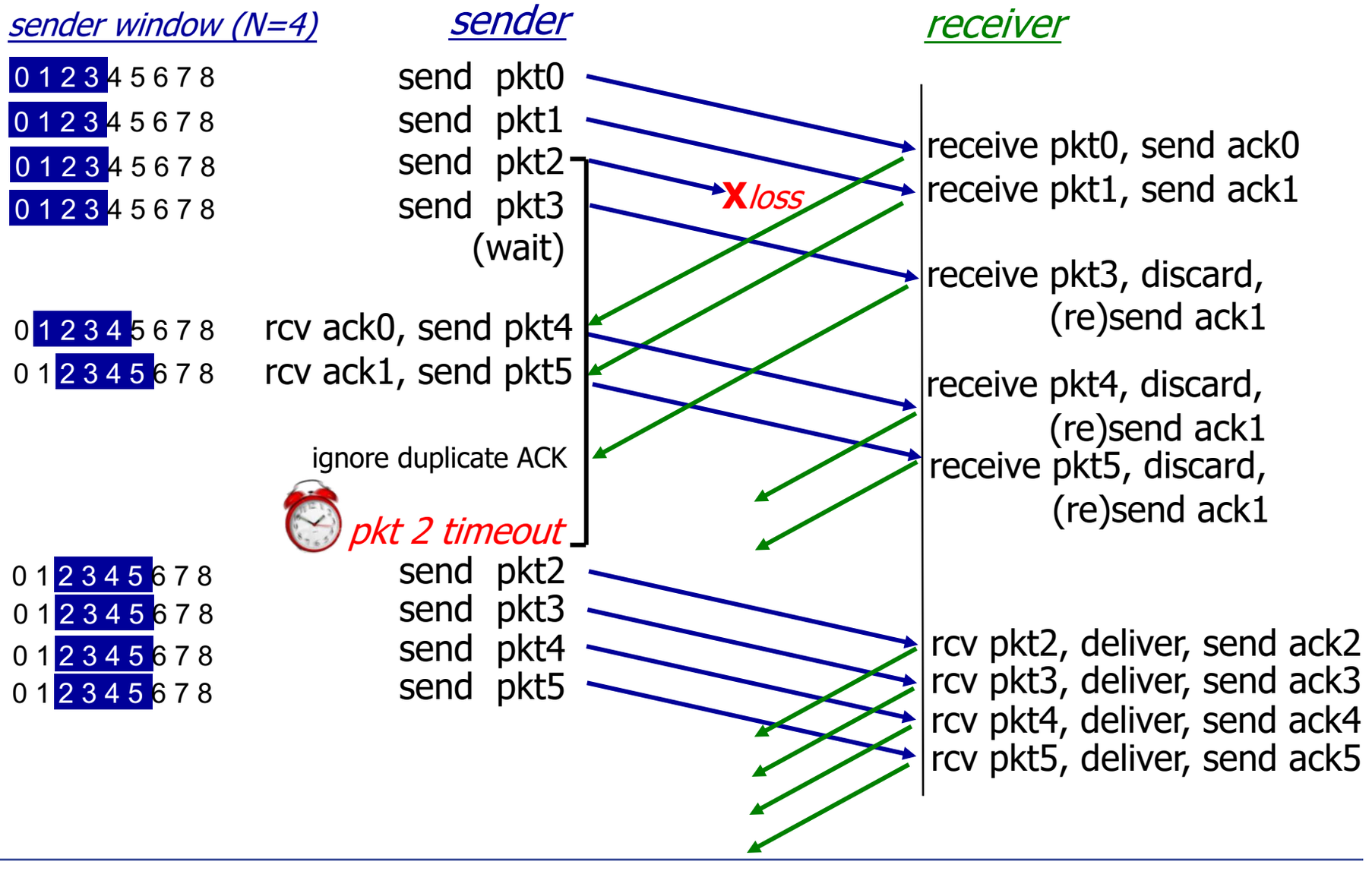


Go-Back-N: Receiver Extended FSM



- ACK-only: always send ACK for correctly-received packet with highest in-order sequence number
  - May generate duplicate ACKs
  - Need only to remember the expected sequence number
- Out of order packet
  - Discard: no receiving buffering!
  - Re-ACK packet with highest in-order sequence number

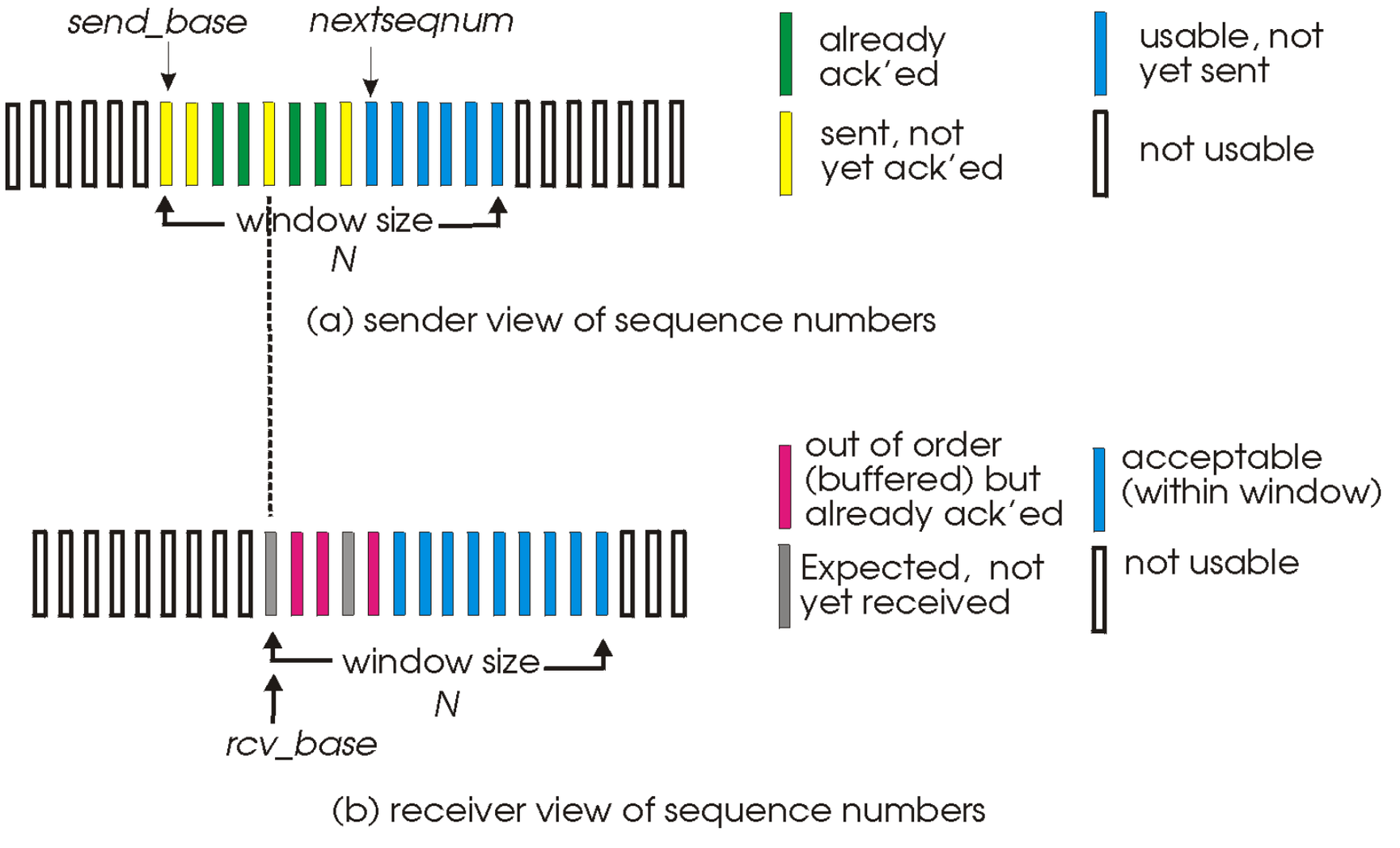
Go-Back-N: In Action



Selective Repeat

- Receiver individually acknowledges all correctly received packets
  - Buffers packets for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received
  - Sender timer for each unacked packet
- Sender window
  - N consecutive sequence numbers
  - Limits the sequence numbers of sent, unacked packets

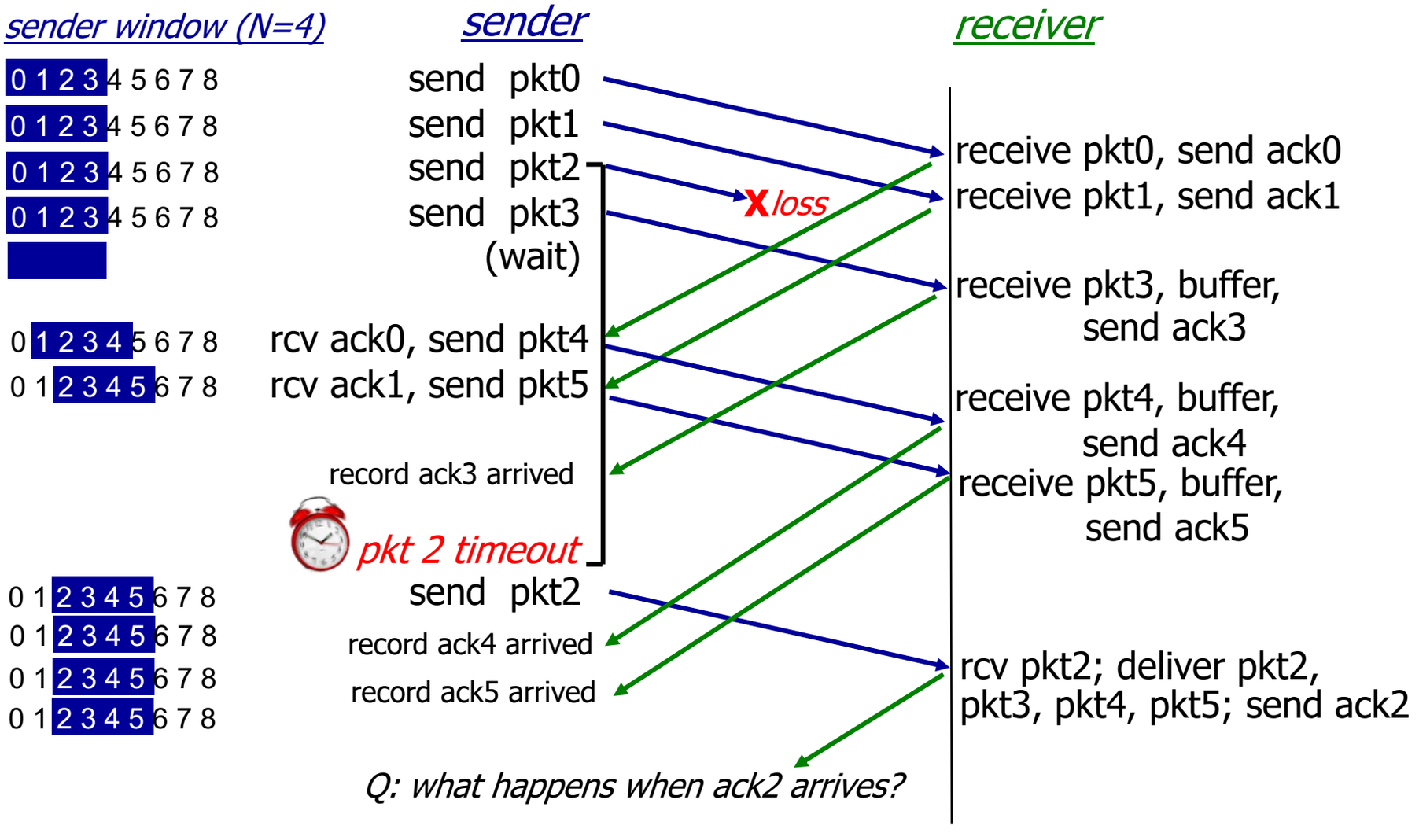
Selective Repeat: Windows



Selective Repeat

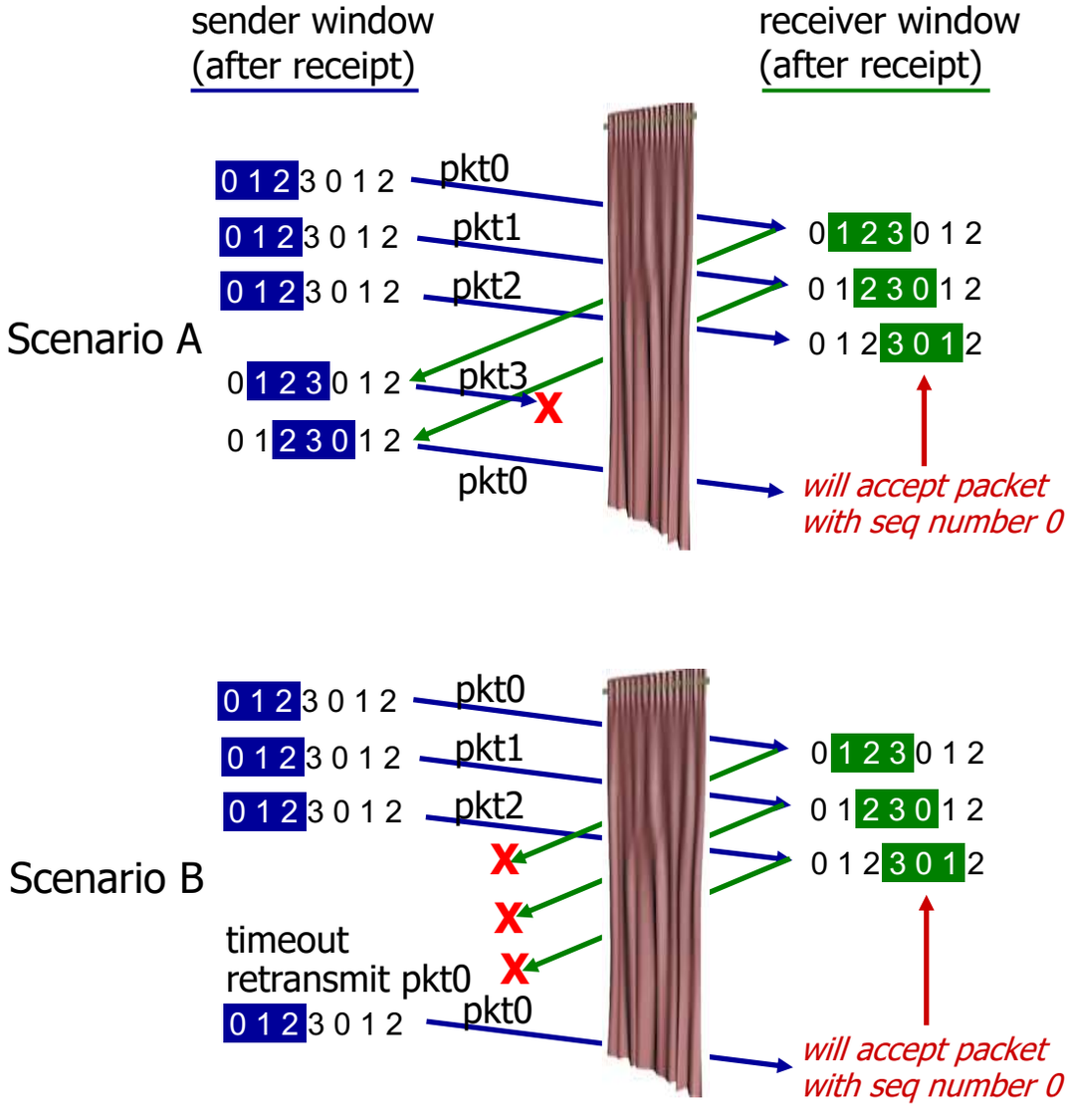
- Sender
  - Data from above
    - If next available sequence number in window, send packet
  - Timeout(n)
    - Resend packet n, restart timer
  - ACK(n) in [sendbase, sendbase+N]
    - Mark packet n as received
    - If n smallest unacked packet, advance window base to next unacked sequence number
- Receiver
  - Packet n in [rcvbase, rcvbase+N-1]
    - Send ACK(n)
    - Out-of-order: buffer
    - In-order: deliver (with buffered, in-order packets), advance window to next no-yet-received packet
  - Packet n in [rcvbase-N, rcvbase-1]
    - ACK(n)
  - Otherwise
    - Ignore

Selective Repeat: In Action



Selective Repeat: Dilemma

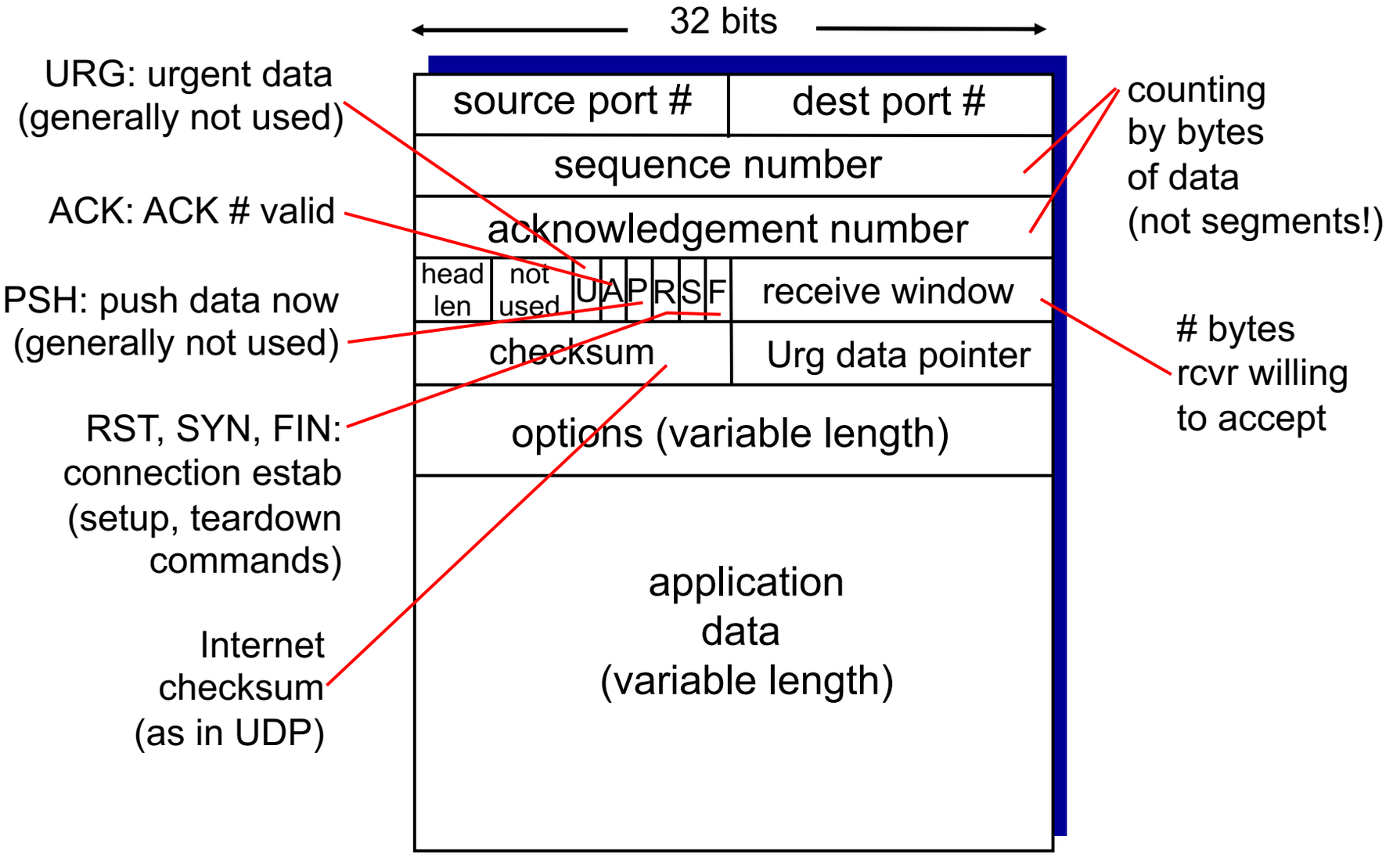
- Example:
  - Sequence numbers; 0, 1, 2, 3
  - Window size = 3
  - Receiver sees no difference in two scenarios!
  - Duplicate date accepted as new in (b)



TCP: Overview [RFC 793, 1122, 1323, 2018, 2581]

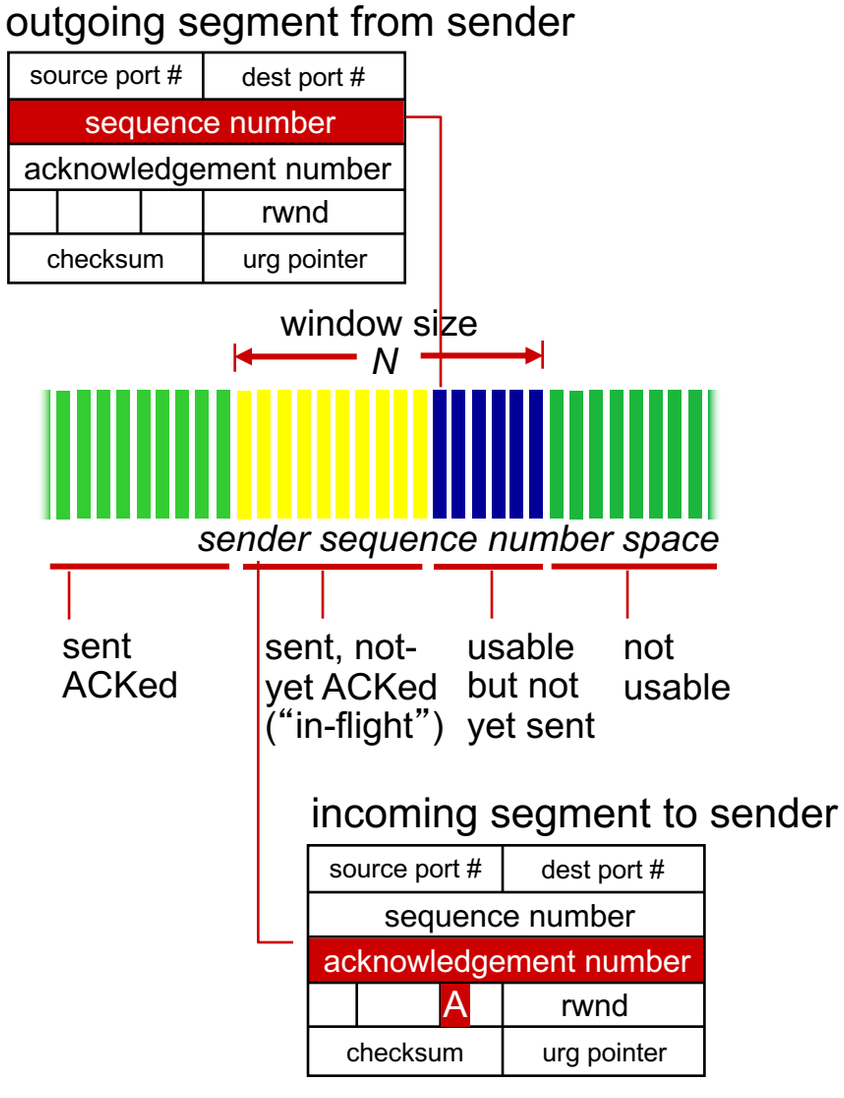
- Point-to-point
  - One sender, one receiver
- Reliable, in-order byte stream
  - No “message boundaries”
- Pipelined
  - TCP congestion and flow control set window size
- Full duplex data
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- Connection-oriented
  - Handshaking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

TCP: Segment Structure

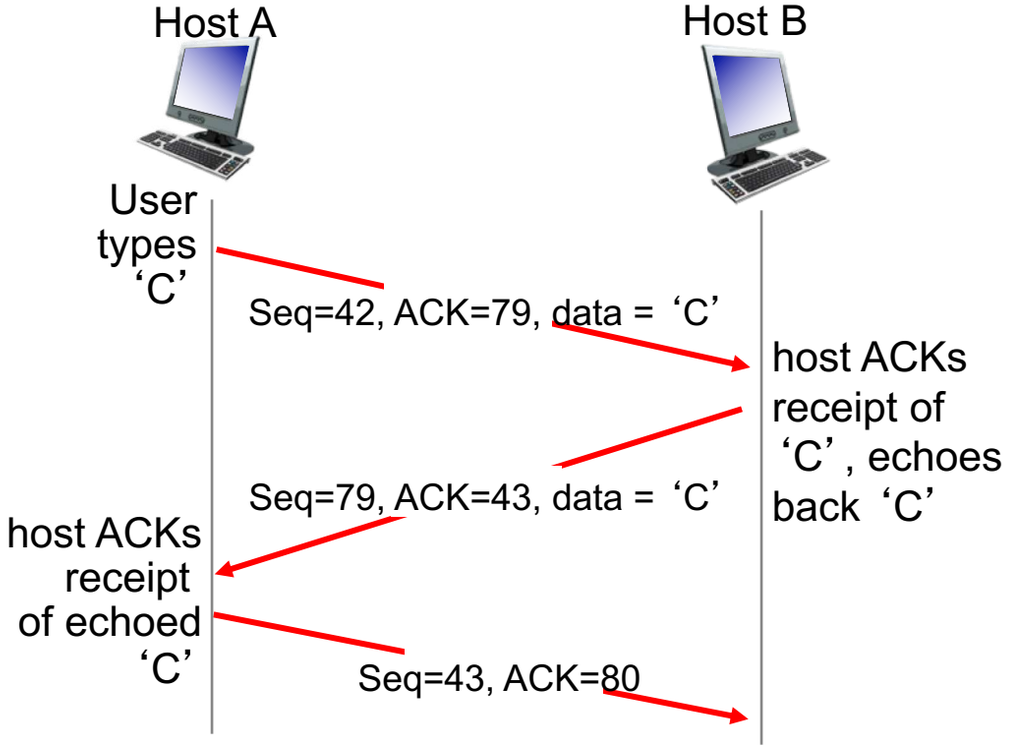


TCP: Sequence Numbers and ACKs

- Sequence numbers
  - Byte stream “number” of first byte in segment’s data
- Acknowledgements
  - Sequence number of next byte expected from other side
  - Cumulative ACK
- How does receiver handle out-of-order segments?
  - Specifications do not say, up to the implementor



TCP: Sequence Numbers and ACKs



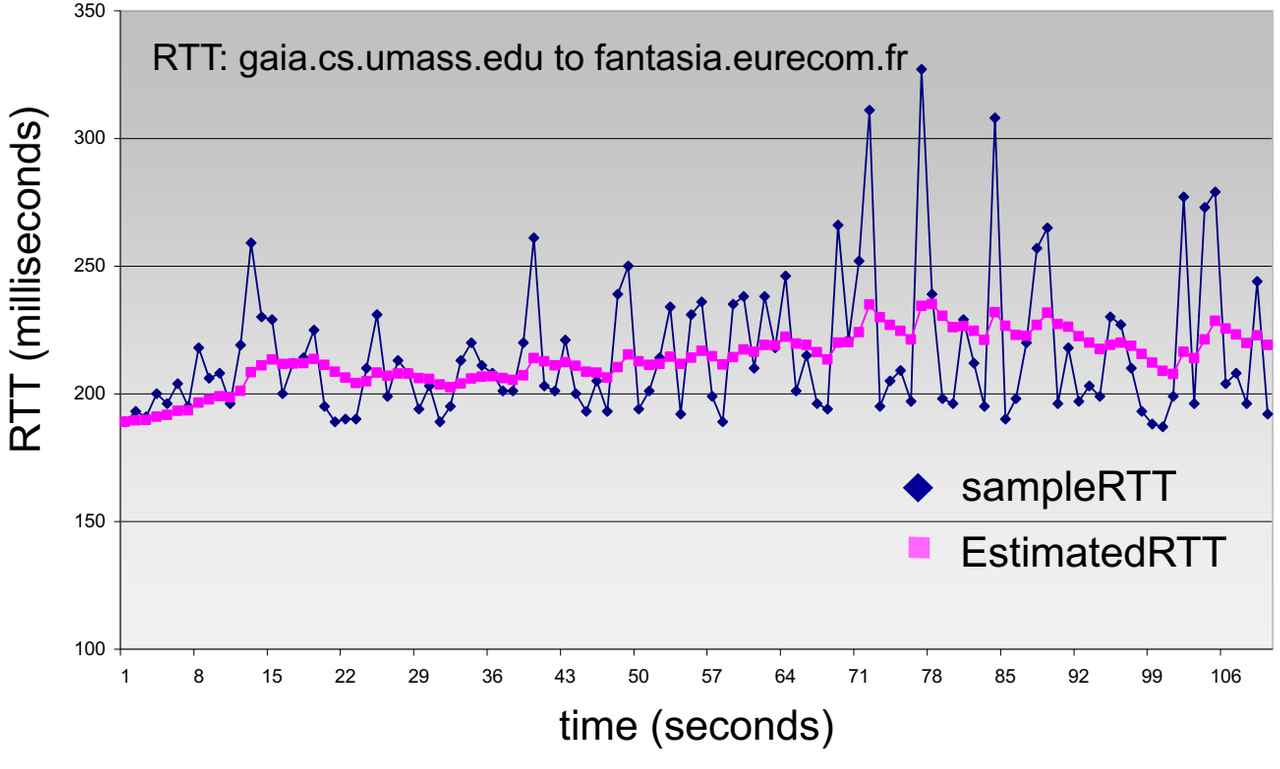
simple telnet scenario

TCP: Round Trip Time, Timeout

- 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
- How to estimate RTT?
  - SampleRTT: measured time from segment transmission until ACK reception
    - Ignore retransmissions
  - SampleRTT varies, therefore “smooth” RTT estimation
    - Average several recent measurements

TCP: Round Trip Time, Timeout

- EstimatedRTT = (1 - α) * EstimatedRTT + α * SampleRTT
- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: α = 0.125



TCP: Round Trip Time, Timeout

- Timeout interval
  - Estimated RTT plus “safety margin”
  - Large variation in the estimated RTT → larger safety margin
- Estimate deviation of the sampled RTT from the estimated one
  - DevRTT = (1 - β) * DevRTT + β * | SampleRTT - EstimatedRTT |
  - Typically, β = 0.25

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

estimated RTT

“safety margin”

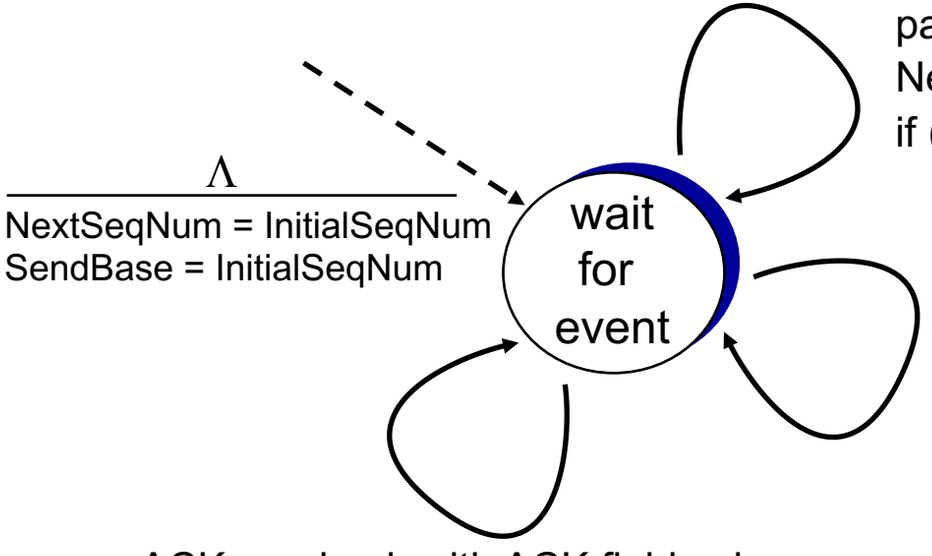
TCP: Reliable Data Transfer

- TCP creates rdt service on top of IP's unreliable service
  - Pipelined segments
  - Cumulative ACKs
  - Single retransmission timer
- Retransmissions triggered by
  - Timeout events
  - Duplicate ACKs
- Let's start with a simplified TCP sender
  - Ignore duplicate ACKs
  - Ignore flow control and congestion control

TCP: Sender Events

- Data received from app
  - Create segment with sequence number
  - Sequence number is byte-stream number of first data byte in segment
  - Start timer if not already running
    - Think of timer as for oldest unacked segment
    - Expiration interval: TimeoutInterval
- Timeout
  - Retransmit segment that caused timeout
  - Restart timer
- ACK received
  - If ack acknowledges previously unacked segment
    - Update what is known to be ACKed
    - Start timer if there are still unacked segments

TCP: Simplified Sender



data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer

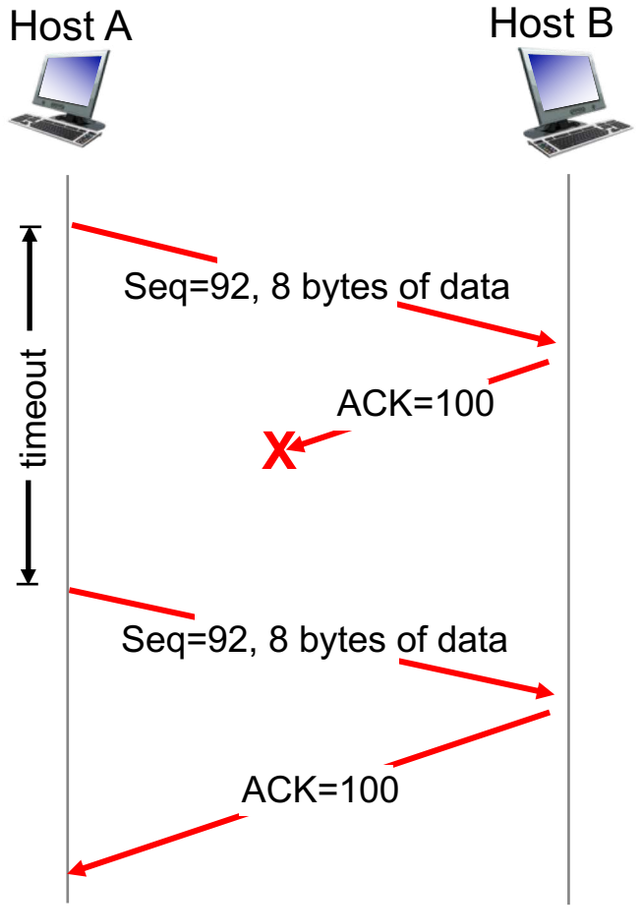
timeout

retransmit not-yet-acked segment
with smallest seq. #
start timer

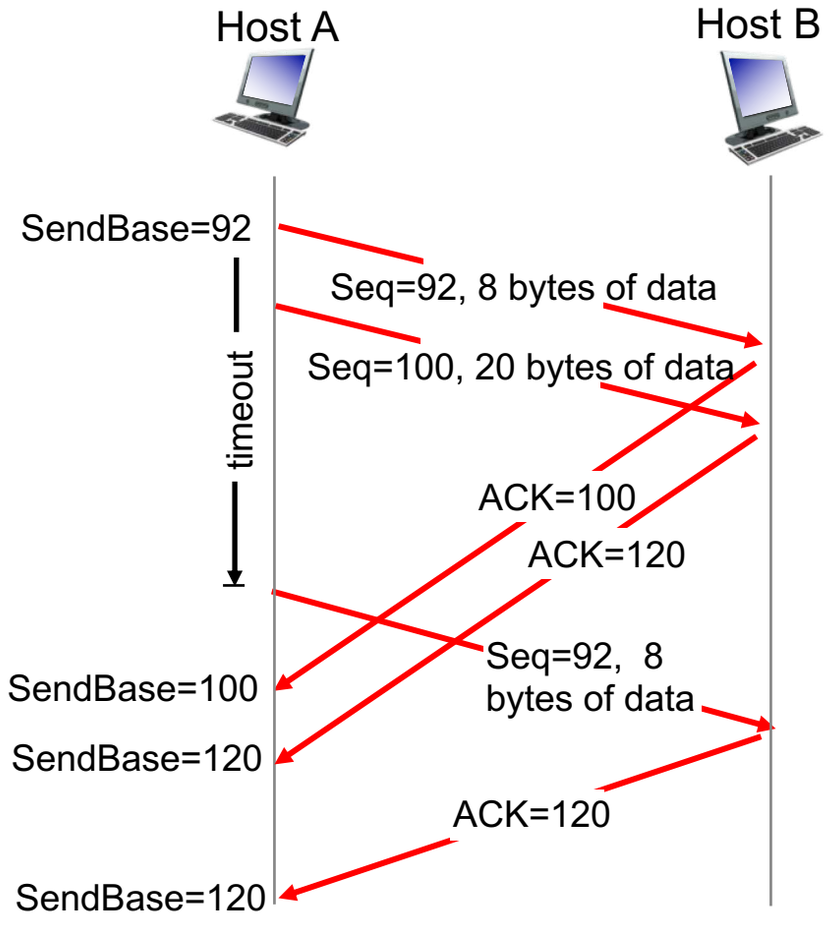
ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
        start timer
     else stop timer
    }
```

TCP: Retransmission Scenarios

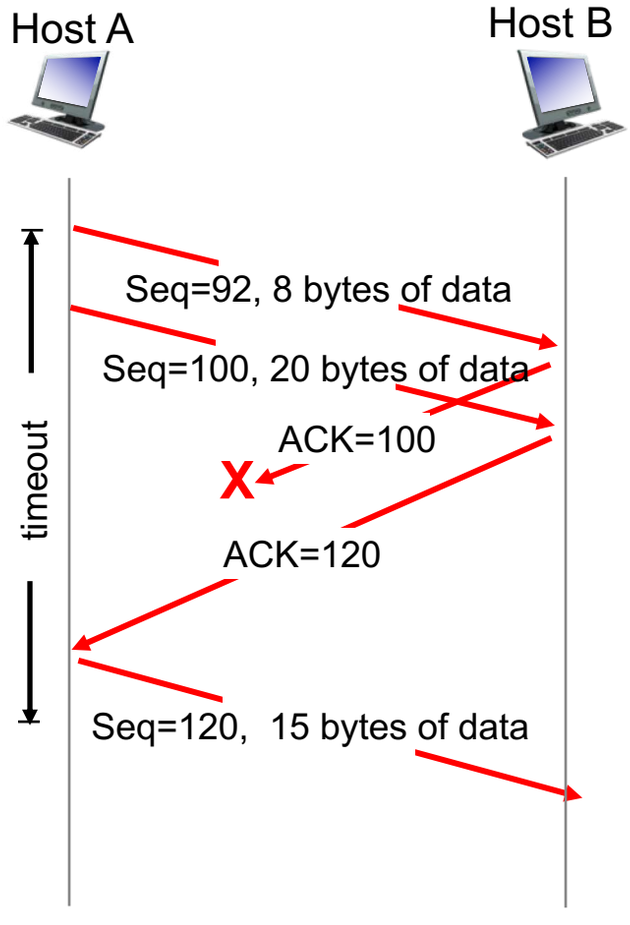


lost ACK scenario



premature timeout

TCP: Retransmission Scenarios



cumulative ACK

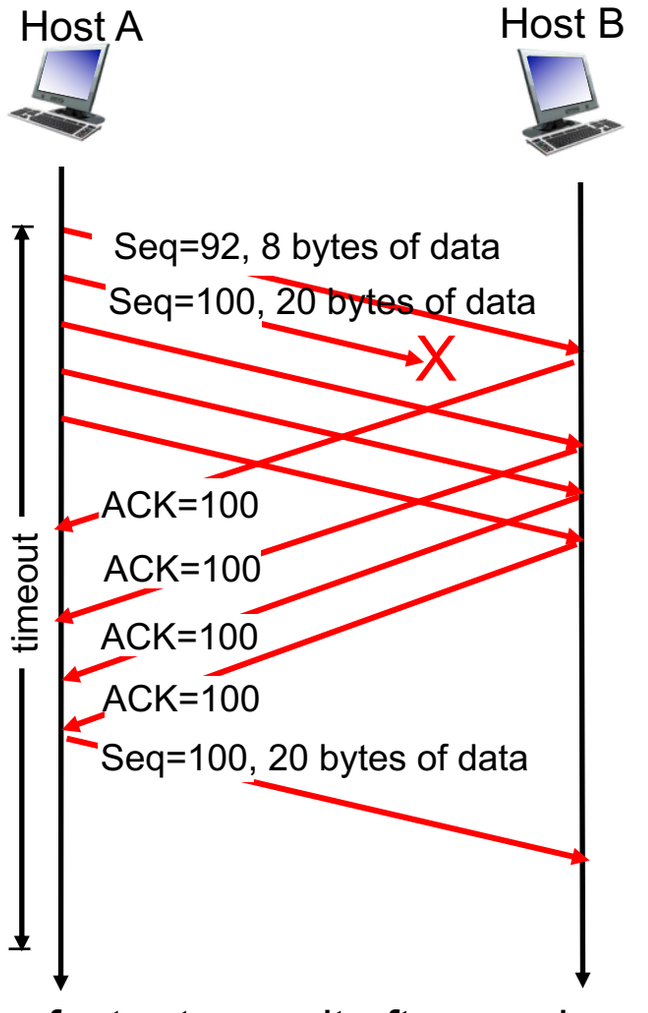
TCP: ACK Generation [RFC 1122, 2581]

- Event: Arrival of in-order segment with expected sequence number; all data up to expected sequence number already ACKed
  - Action: Delayed ACK. Wait up to 500ms for next segment; if no next segment, send ACK
- Event: Arrival of in-order segment with expected sequence number; one other segment has ACK pending
  - Action: Immediately send single cumulative ACK for both in-order segments
- Event: Arrival of out-of-order segment higher-than-expected sequence number; gap detected
  - Action: Immediately send duplicate ACK, indicating sequence number of next expected byte
- Event: Arrival of segment that partially or completely fills gap
  - Action: Immediately 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 ACKs for same data, resend unacked segment with smallest sequence number
    - Likely that unacked segment lost, so no use of waiting for timeout

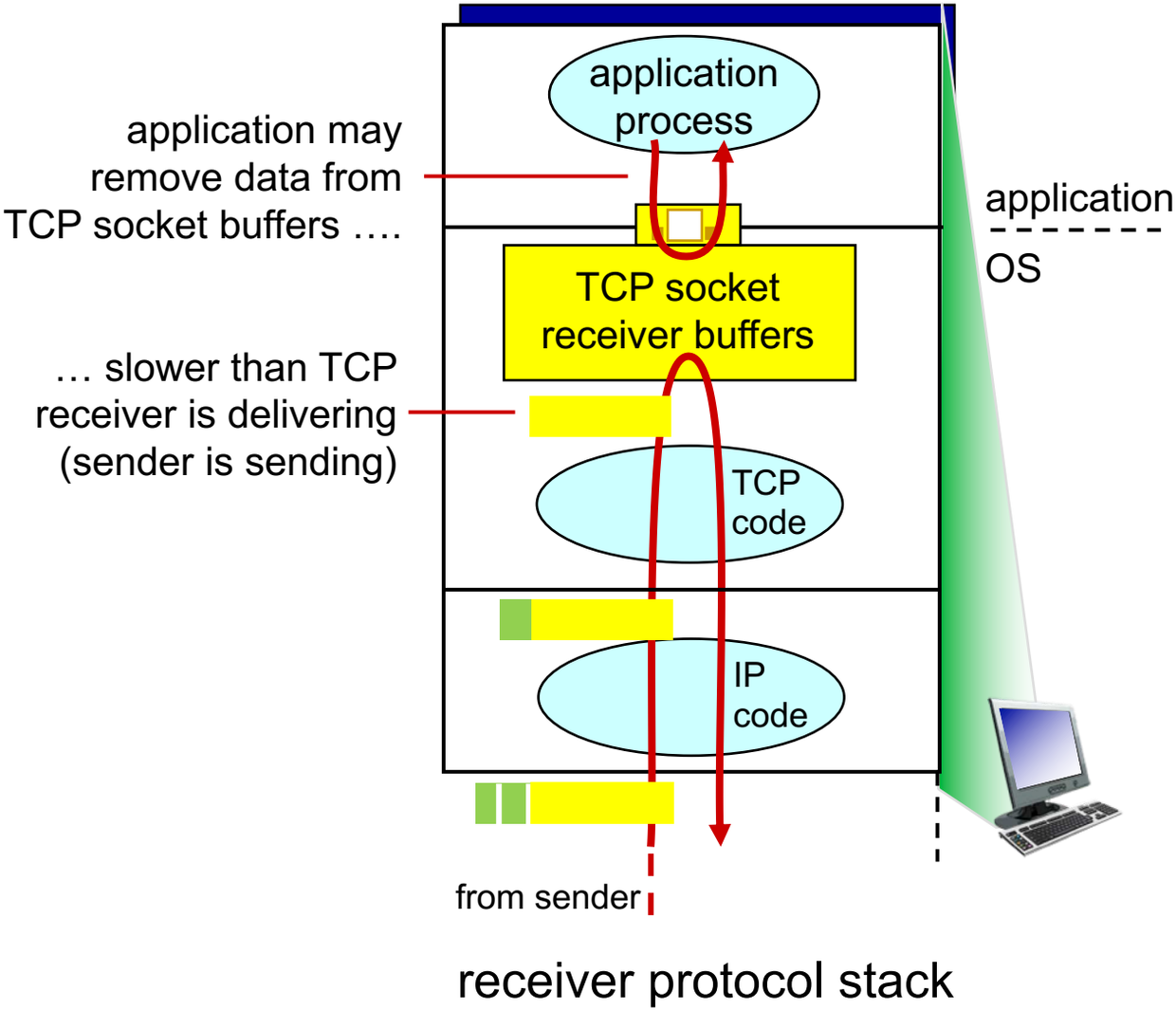
TCP: Fast Retransmit



fast retransmit after sender receipt of triple duplicate ACK

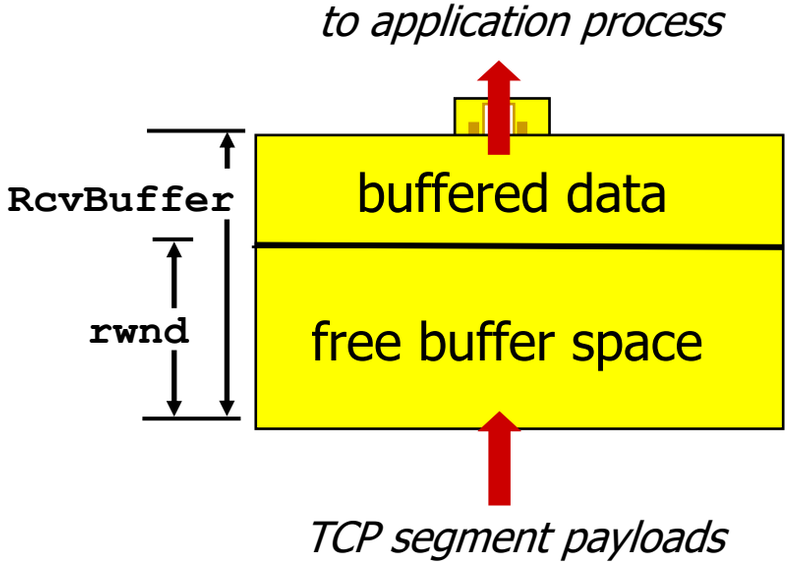
TCP: Flow Control

- Flow control
  - Receiver controls sender, so that sender does not overflow receiver's buffer by transmitting too much, too fast



TCP: Flow Control

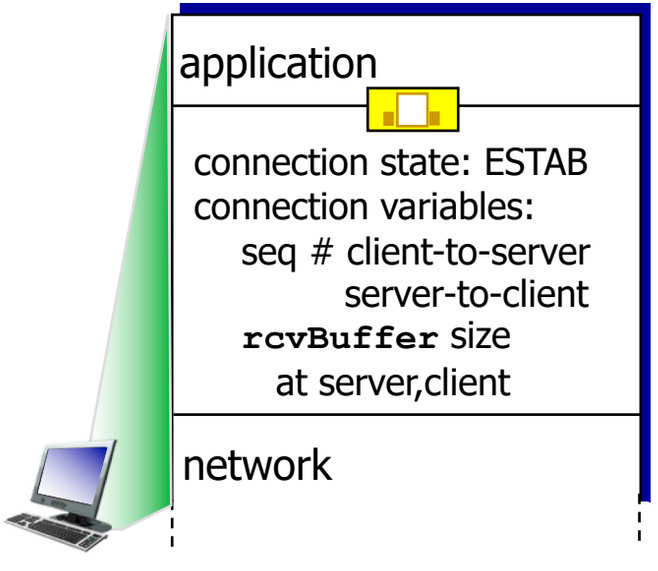
- Receiver "advertises" free buffer space by including `rwnd` value in TCP header of receiver-to-sender segments
  - `RcvBuffer` size set via socket options (e.g., 4096 bytes)
  - Many OS autoadjust `RcvBuffer`
- Sender limits amount of unacked data to receiver's `rwnd` value
- Guarantees receive buffer will not overflow

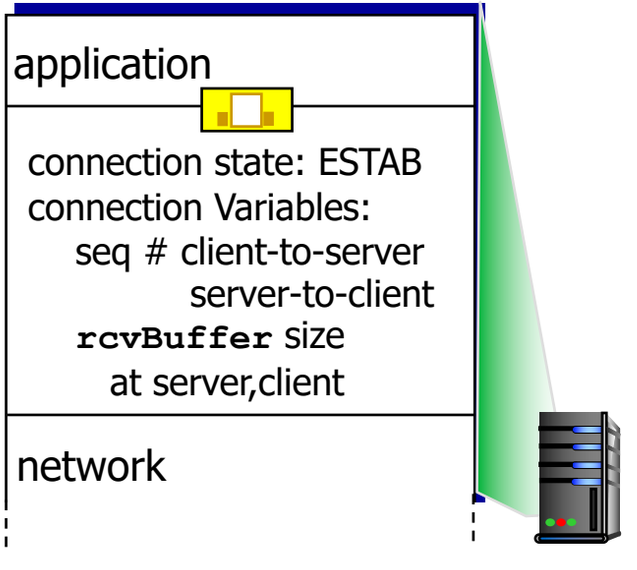


receiver-side buffering

Connection Management

- Handshake before exchanging data
  - Agree to establish a connection
  - Agree on connection parameters



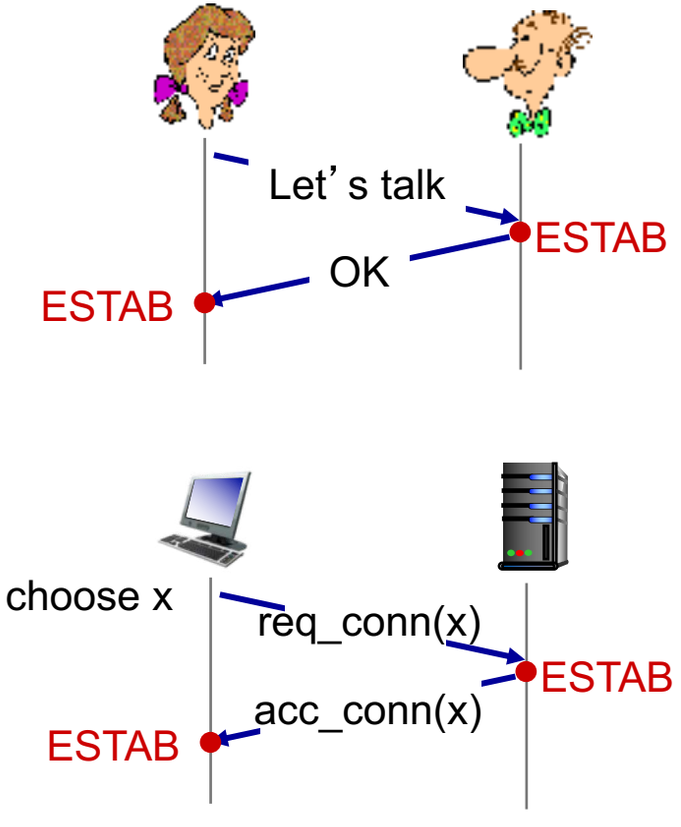



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

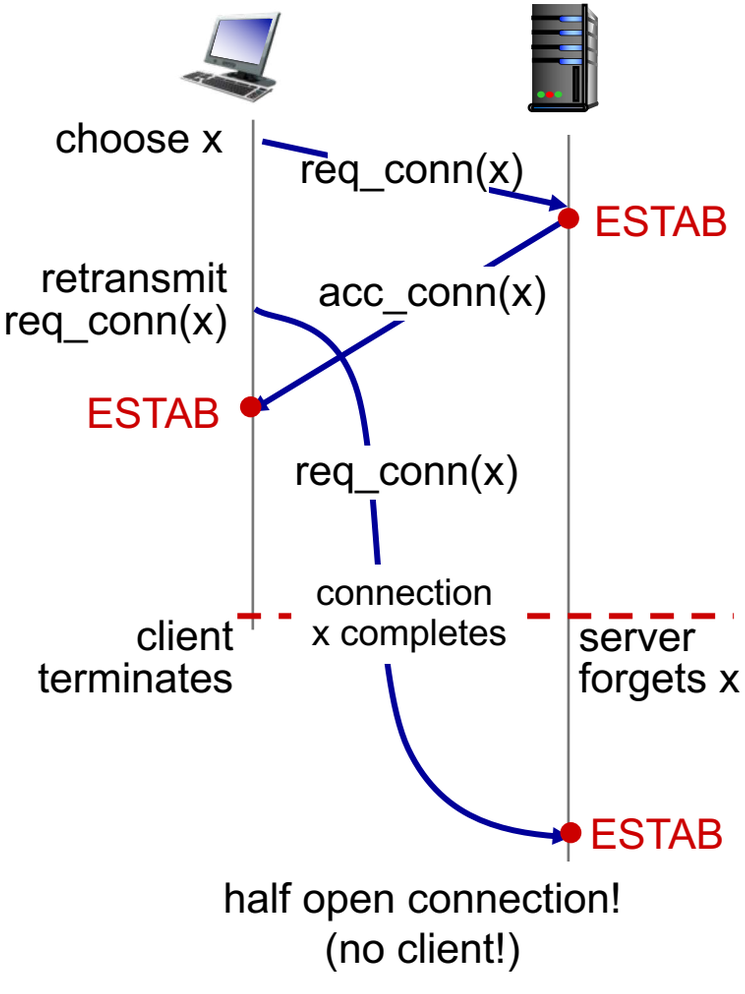
Agreeing to Establish a Connection

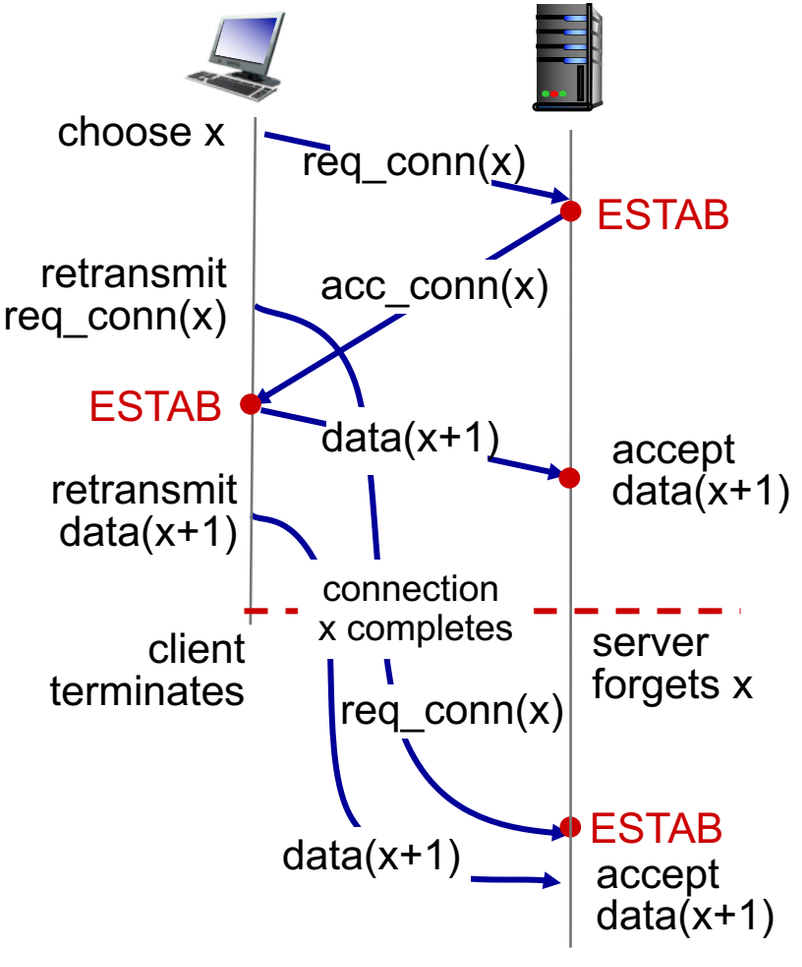
- 2-way handshake



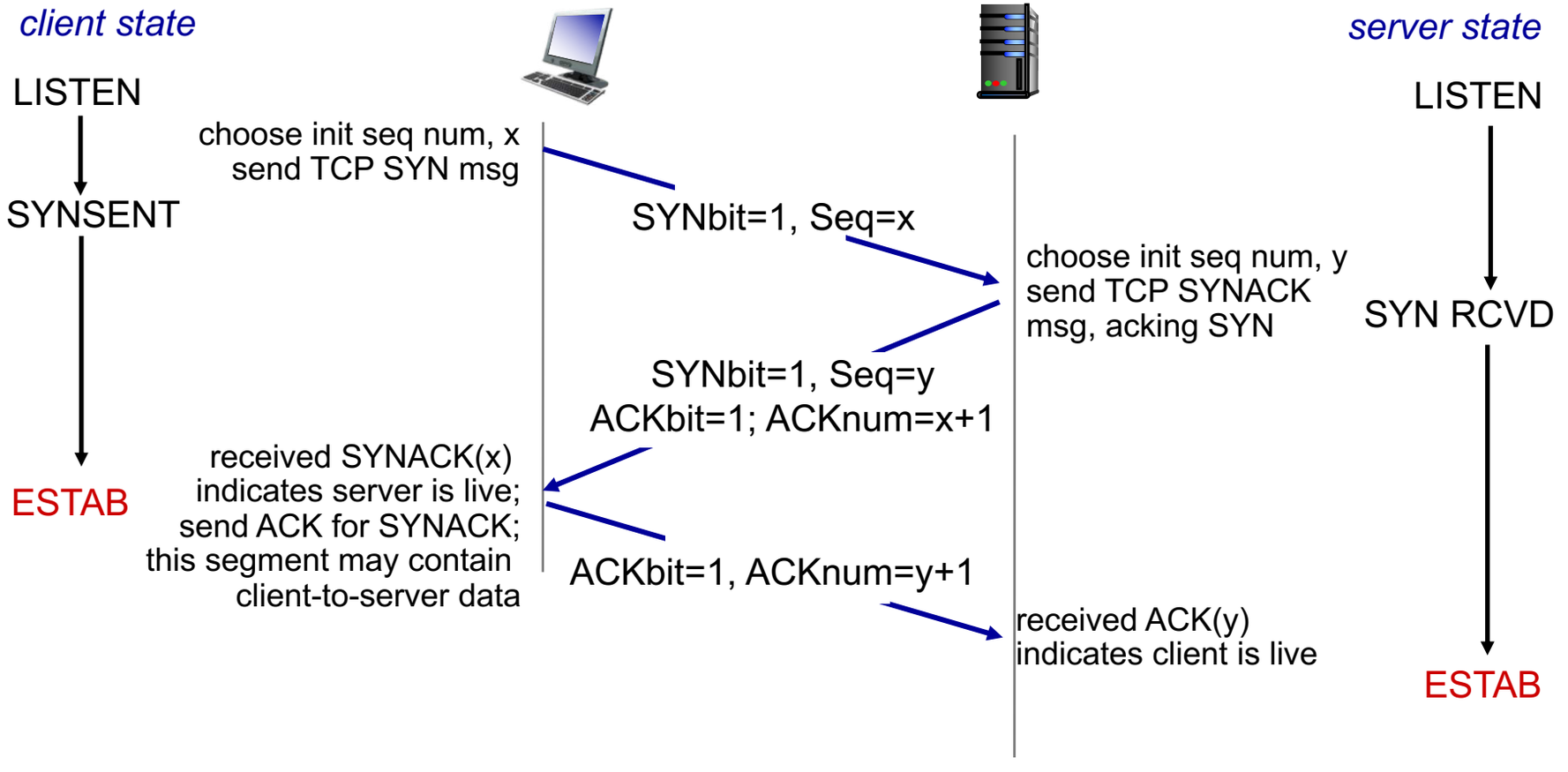
- Does 2-way handshake always work in network?
  - Variable delays
  - Retransmitted messages due to message loss
  - Message reordering
  - Cannot "see" the other side

2-Way Handshake Failure Scenarios

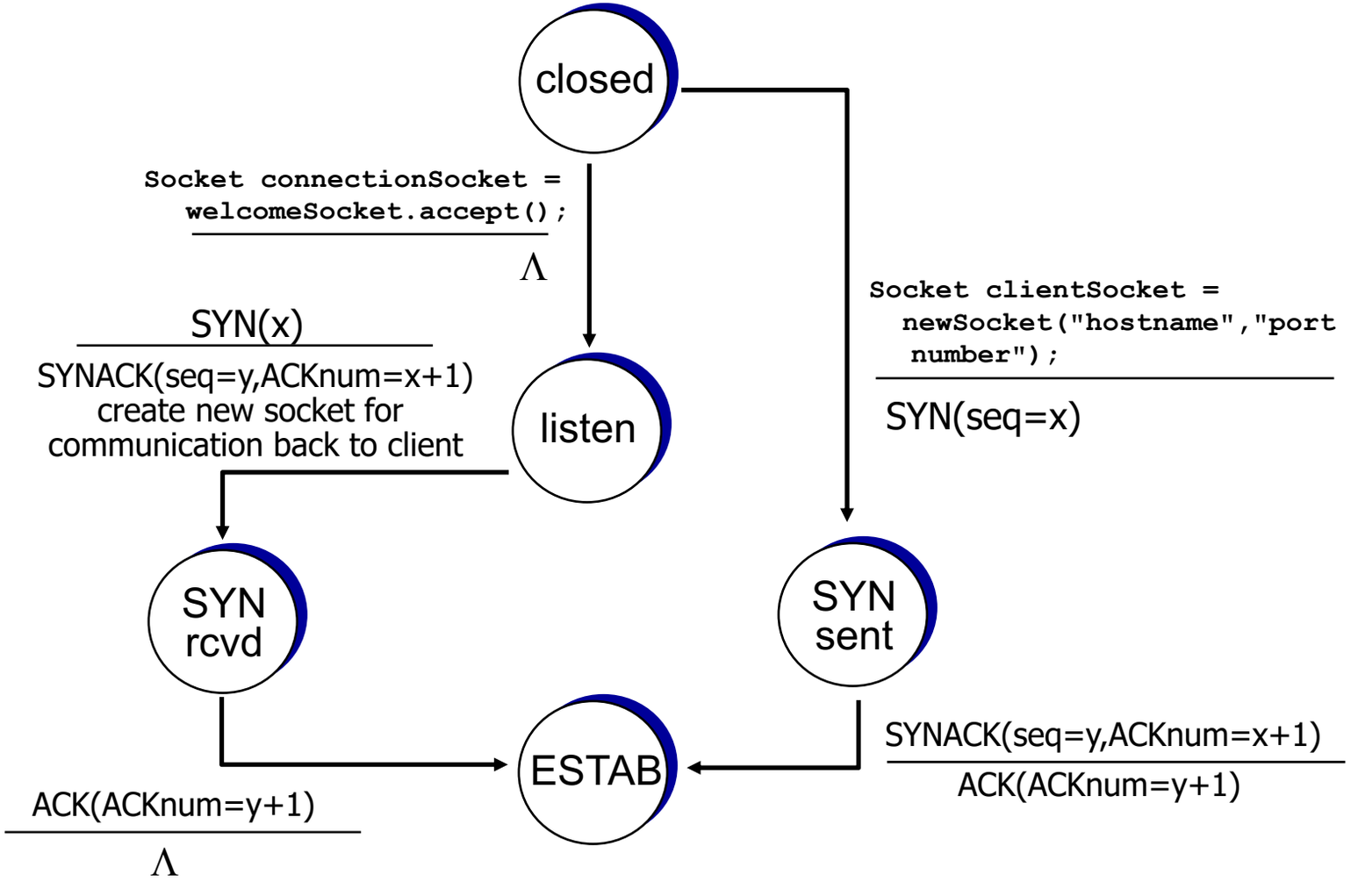




TCP 3-Way Handshake



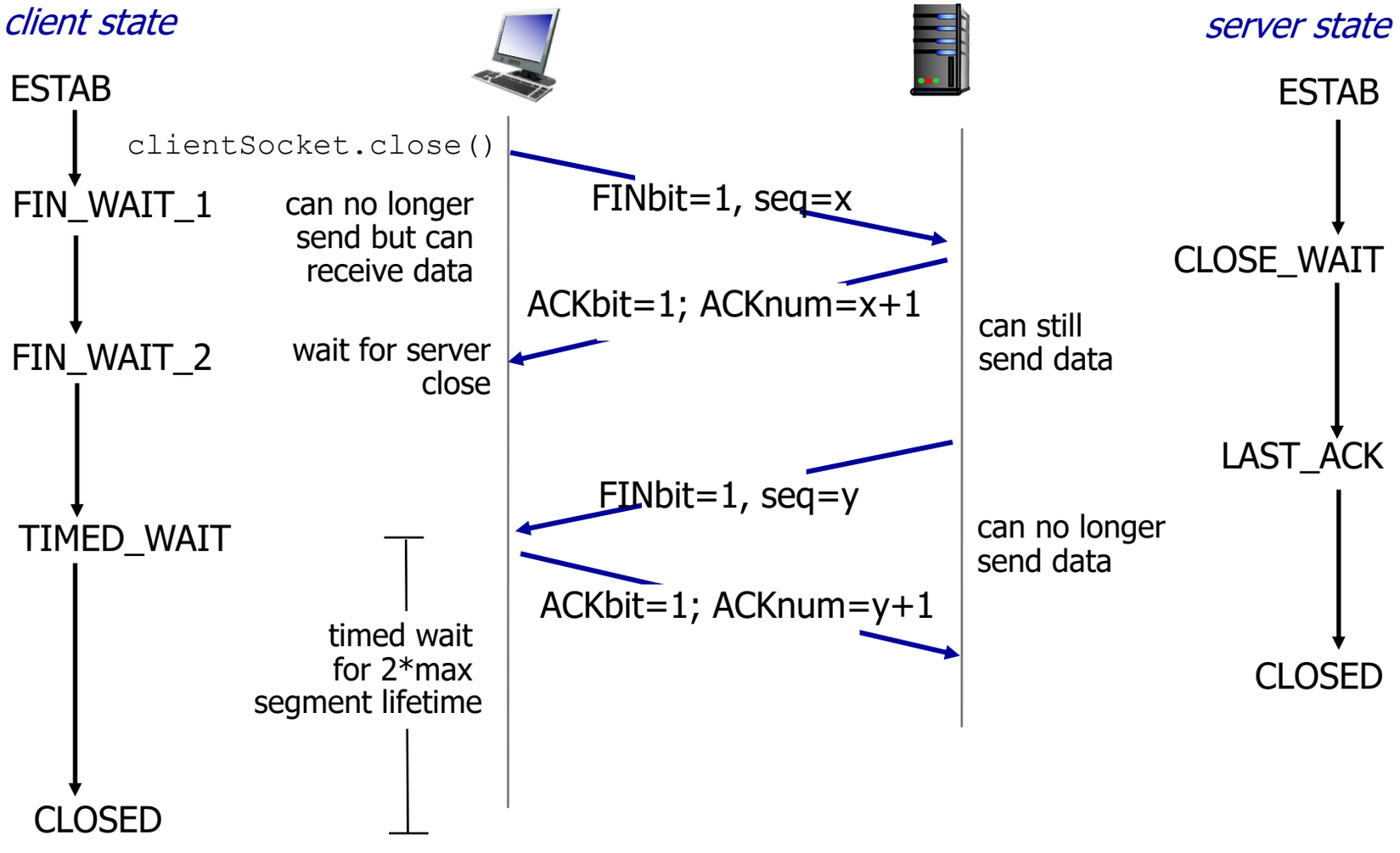
TCP 3-Way Handshake: FSM



TCP: Closing a Connection

- Client and server close each 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

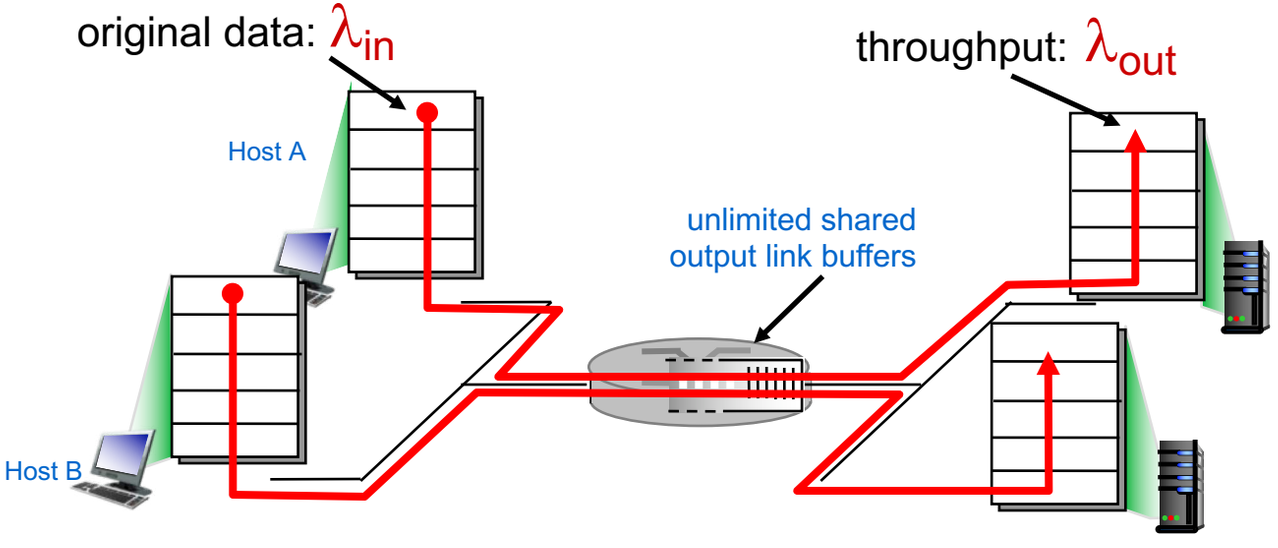


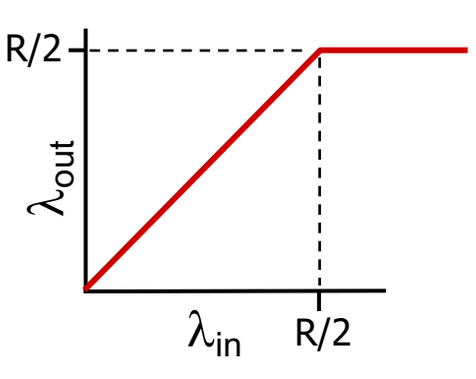
Principles of Congestion Control

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

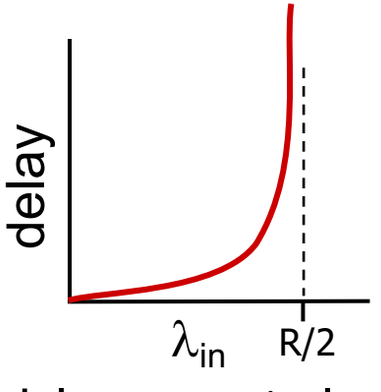
Causes/costs of Congestion: Scenario 1

- Two senders, two receivers
- One router, infinite buffers
- Output link capacity: R
- No retransmission





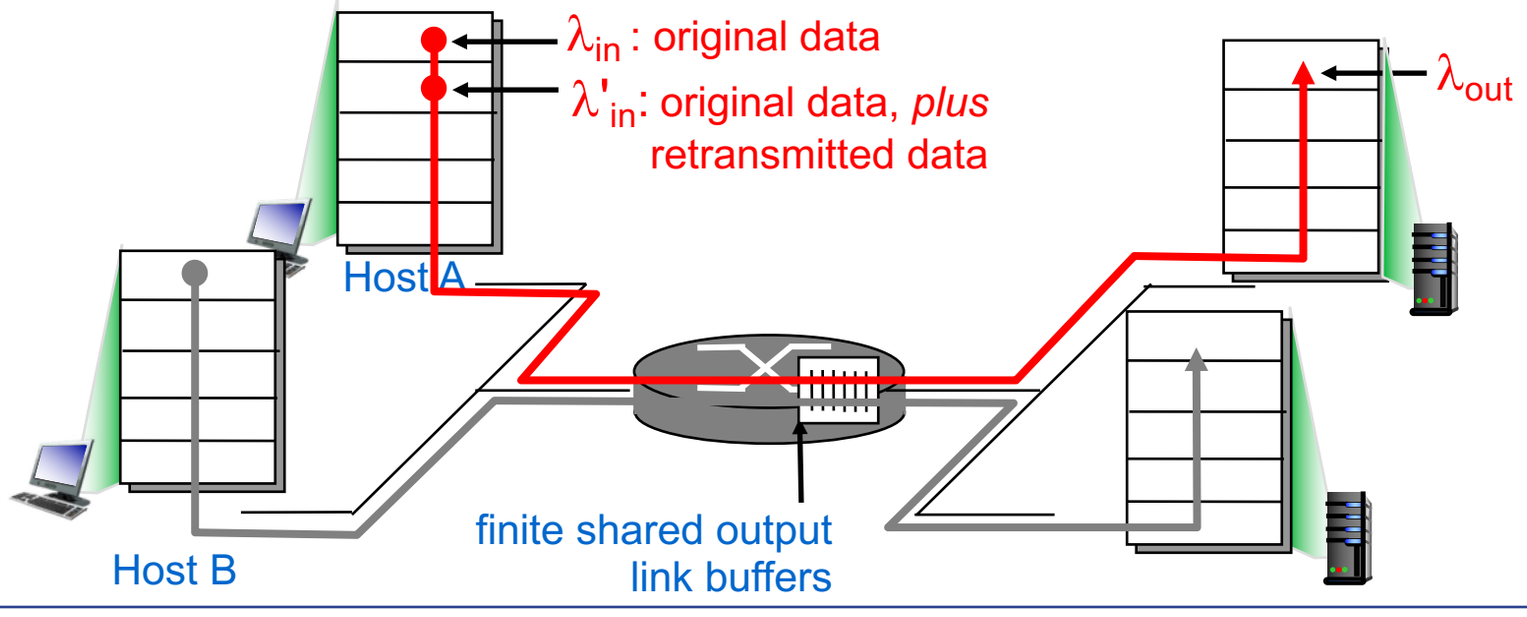
maximum per-connection throughput: R/2



large delays as arrival rate, λ_{in}, approaches capacity

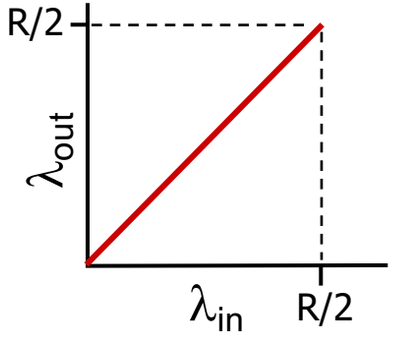
Cuases/Costs of Congestion: Scenario 2

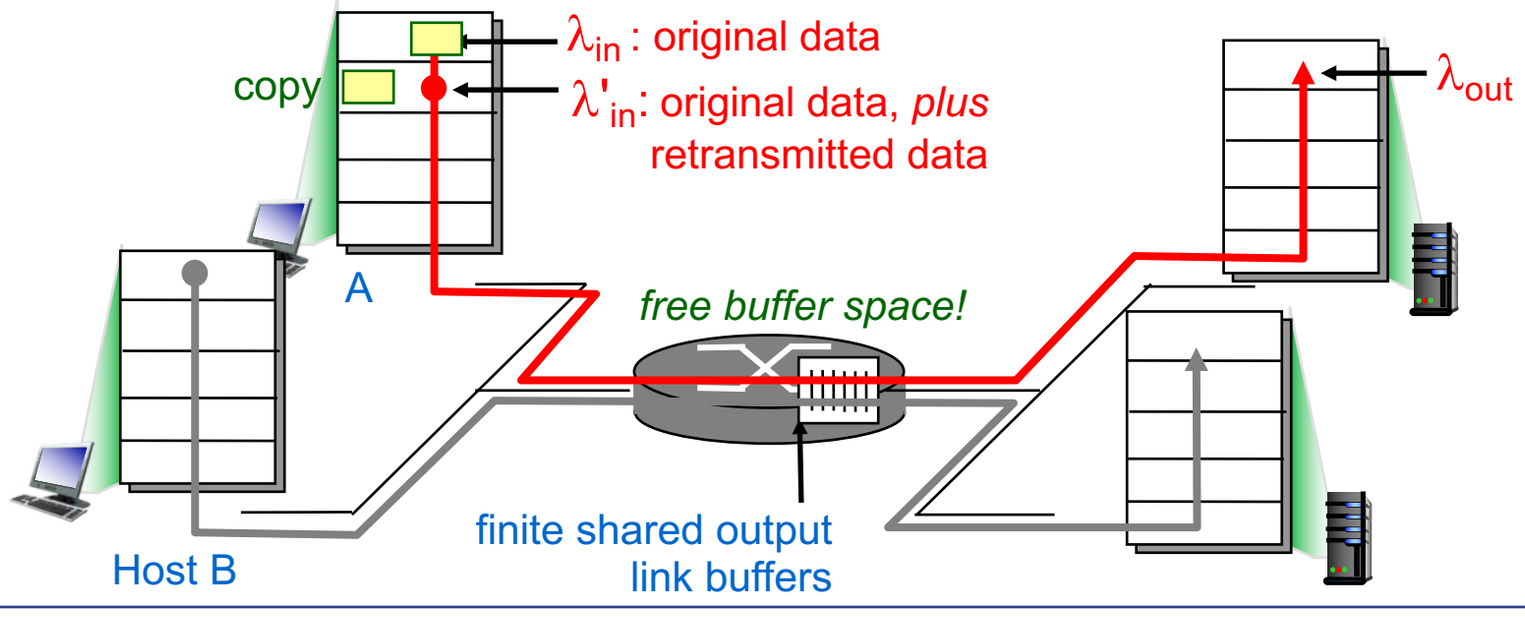
- One router, finite buffers
- Sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes retransmissions: $\lambda'_{in} \geq \lambda_{in}$



Causes/Costs of Congestion: Scenario 2

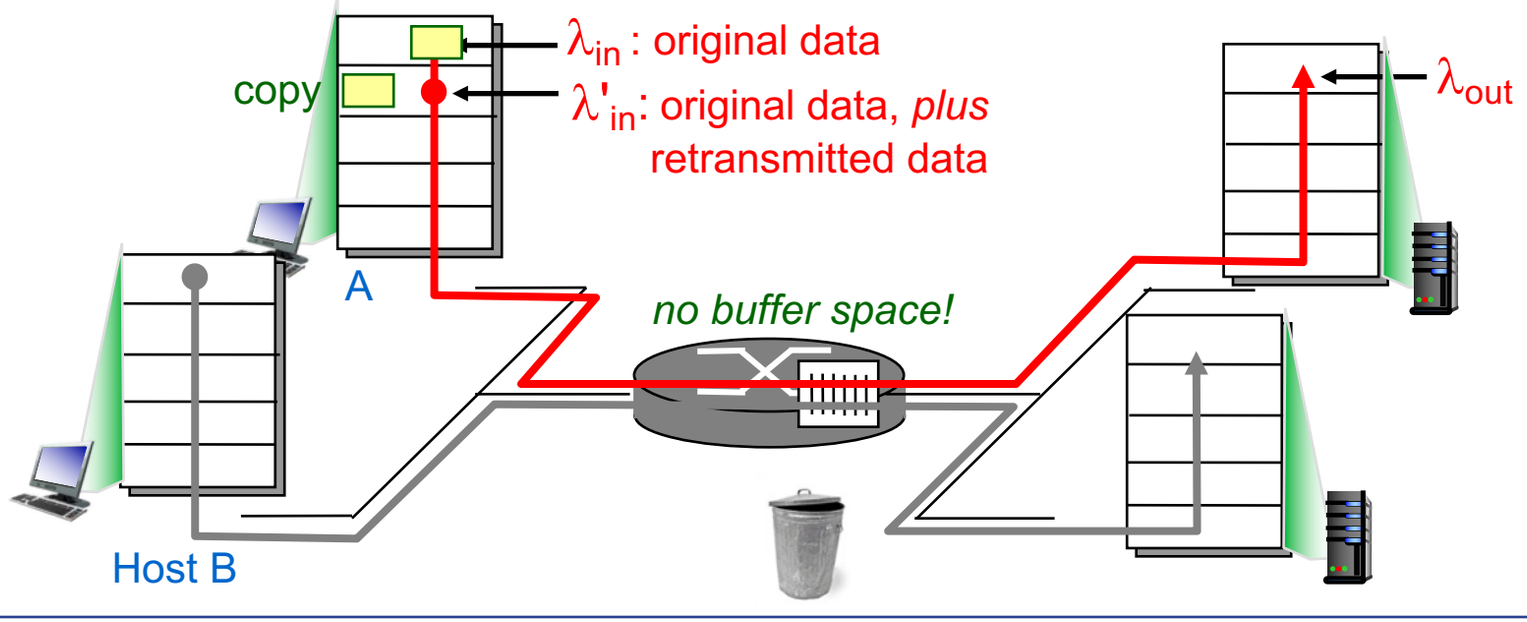
- Idealisation: perfect knowledge
- Sender sends only when router buffers available





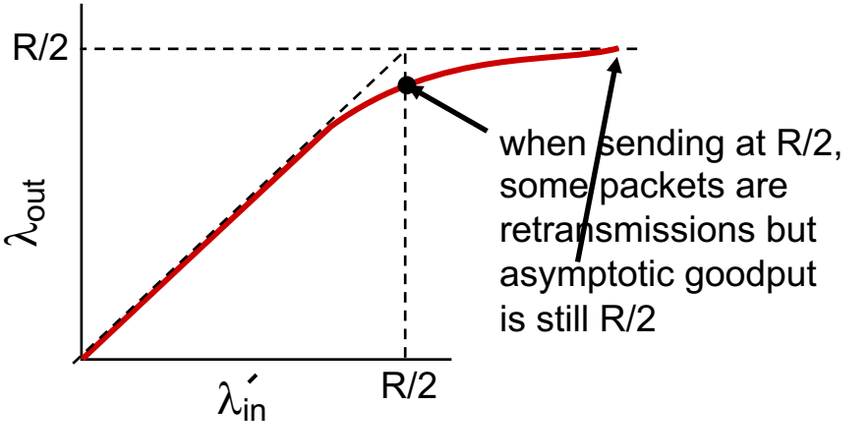
Causes/Costs of Congestion: Scenario 2

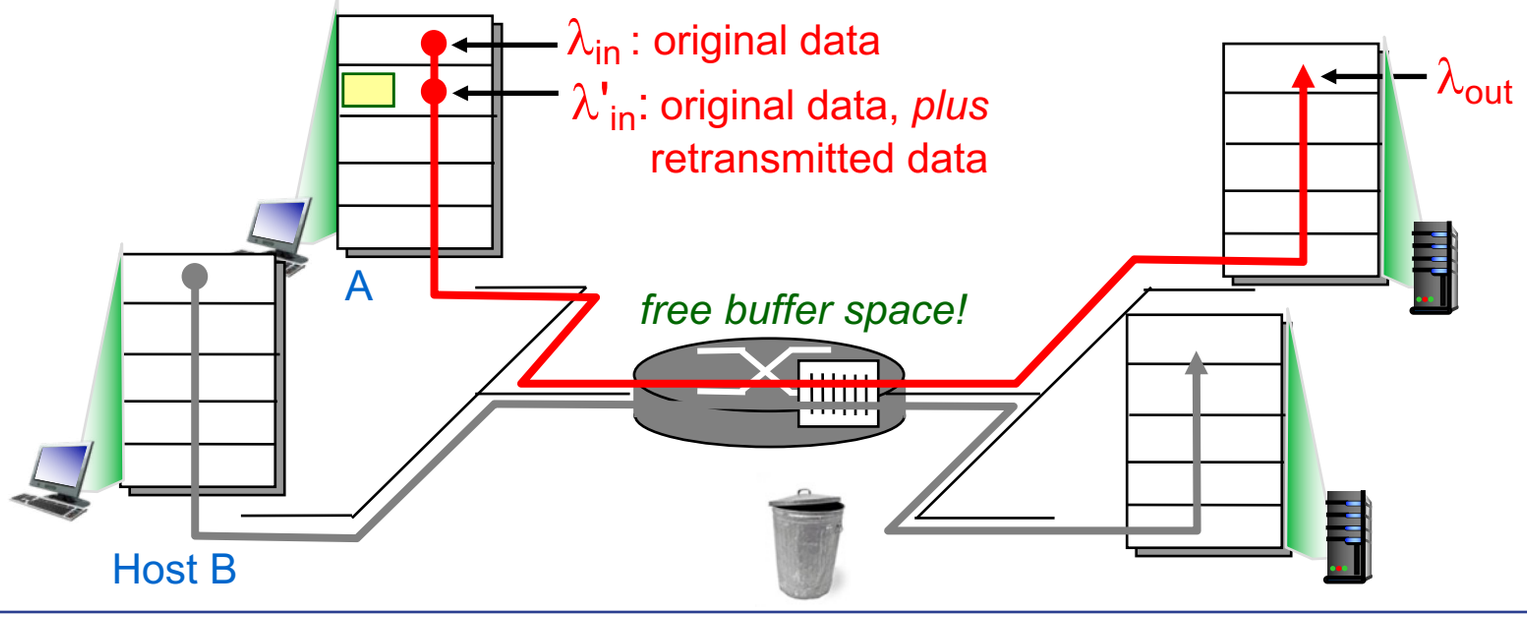
- Idealisation: known loss
  - Packets can be lost, dropped at router due to full buffers
- Sender only resends if packet known to be lost



Causes/Costs of Congestion: Scenario 2

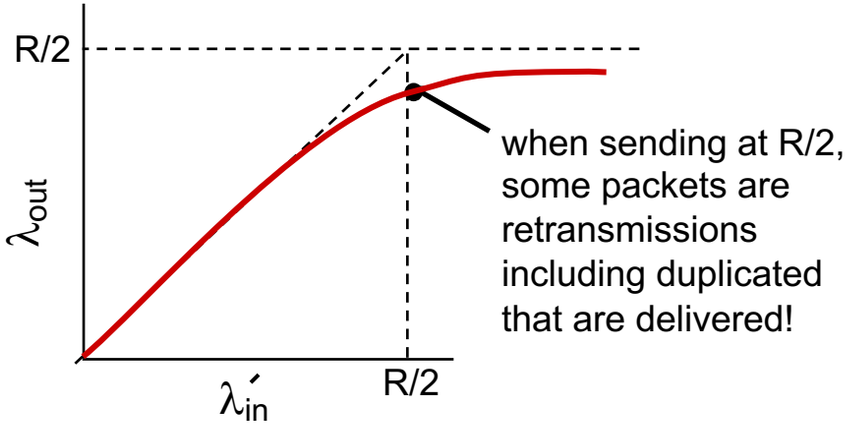
- Idealisation: known loss
  - Packets can be lost, dropped at router due to full buffers
- Sender only resends if packet known to be lost

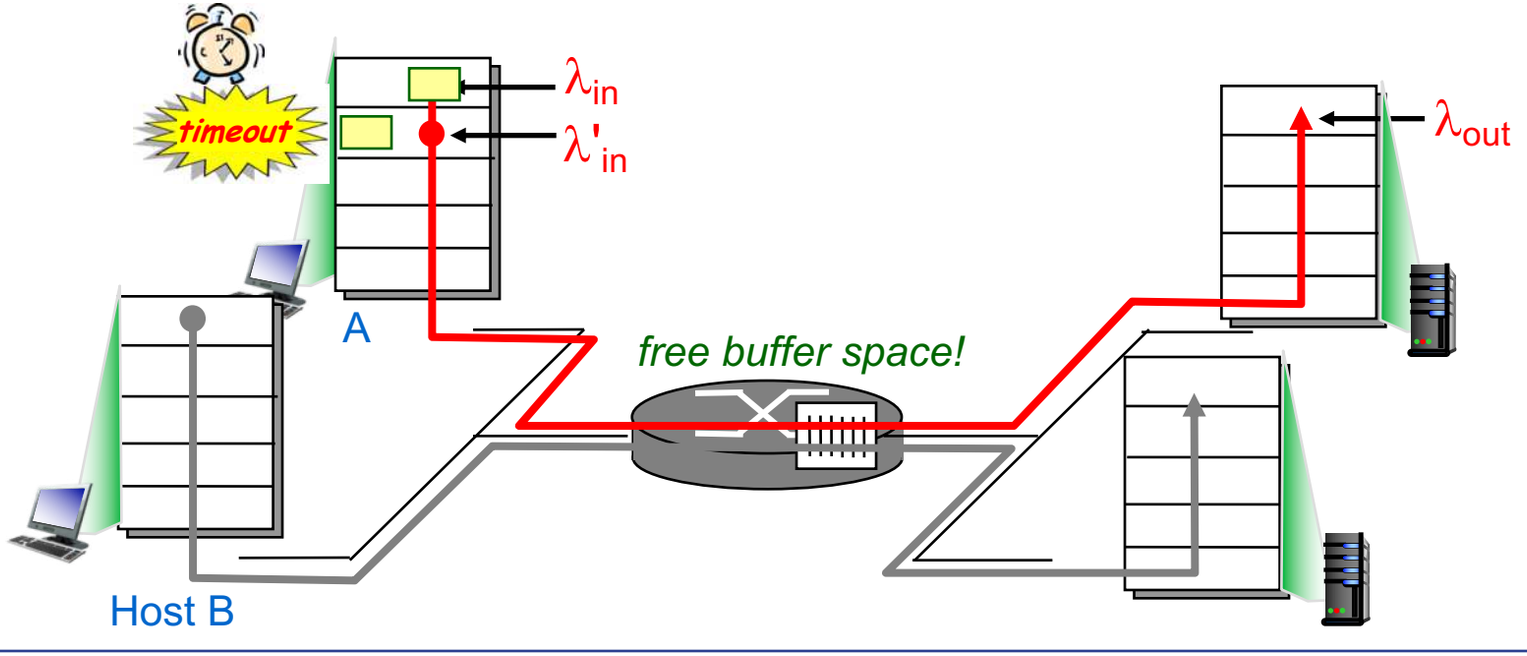




Causes/Costs of Congestion: Scenario 2

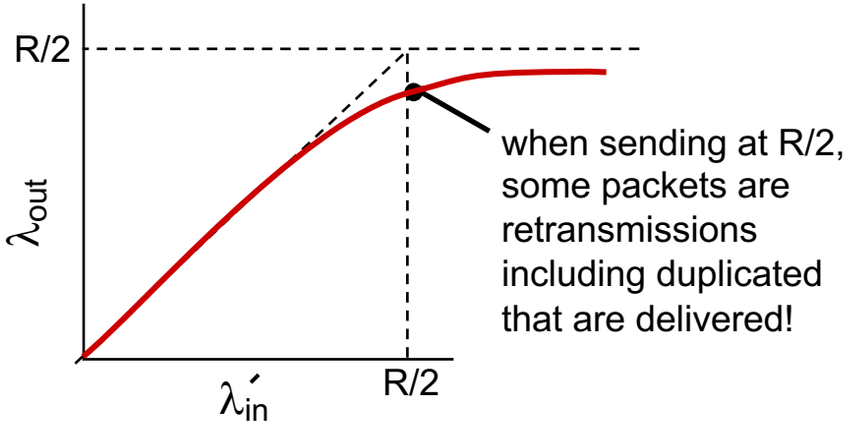
- Realistic: duplicates
  - Packets can be lost, dropped at router due to full buffers
- Sender timers out prematurely, sending two copies both of which are delivered





Causes/Costs of Congestion: Scenario 2

- Realistic: duplicates
  - Packets can be lost, dropped at router due to full buffers
- Sender timers out prematurely, sending two copies both of which are delivered

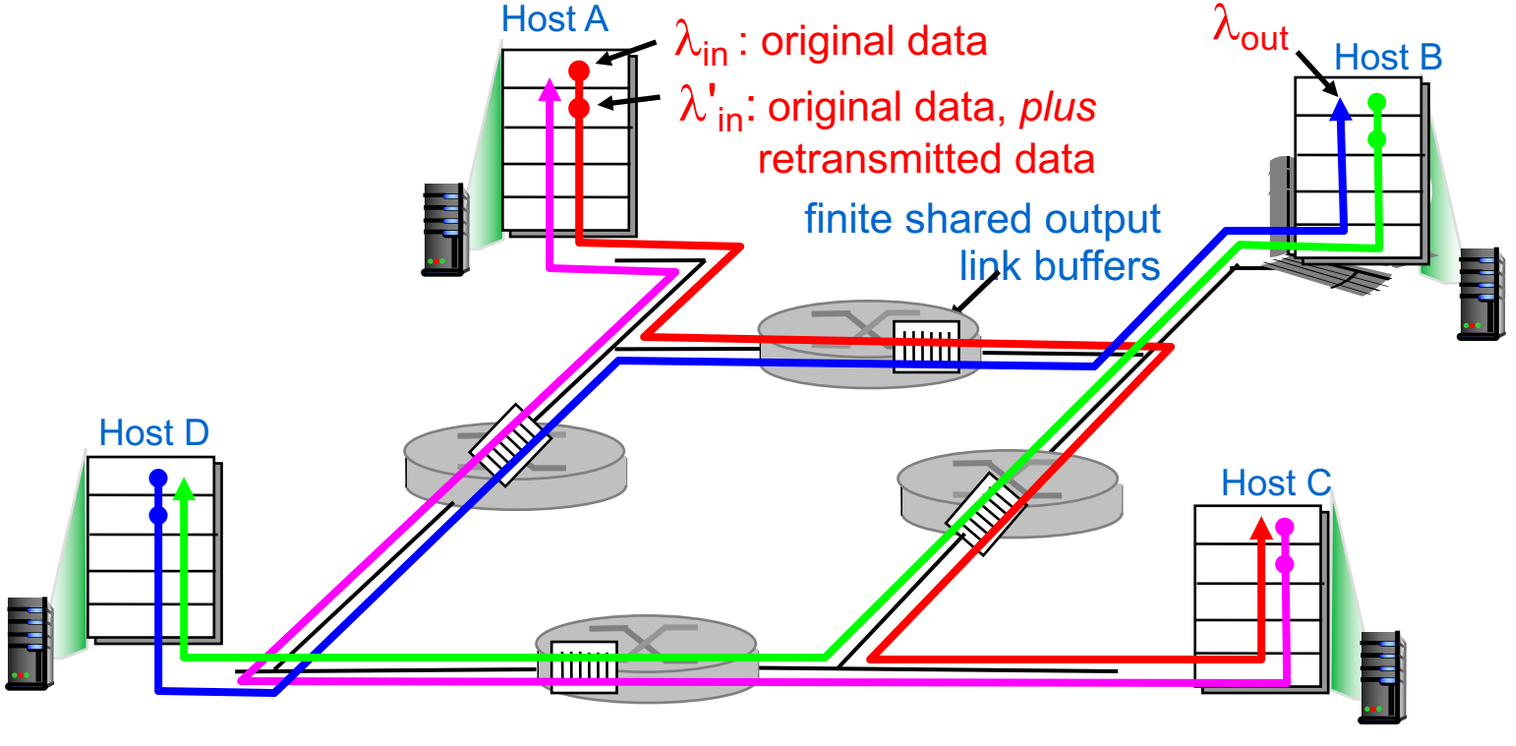


- "Costs" of congestion
  - More work (retransmissions) for given "goodput"
  - Unneeded retransmissions: link carries multiple copies of the same packet

Causes/Costs of Congestion: Scenario 3

- Four senders
- Multihop paths
- Timeout/retransmit

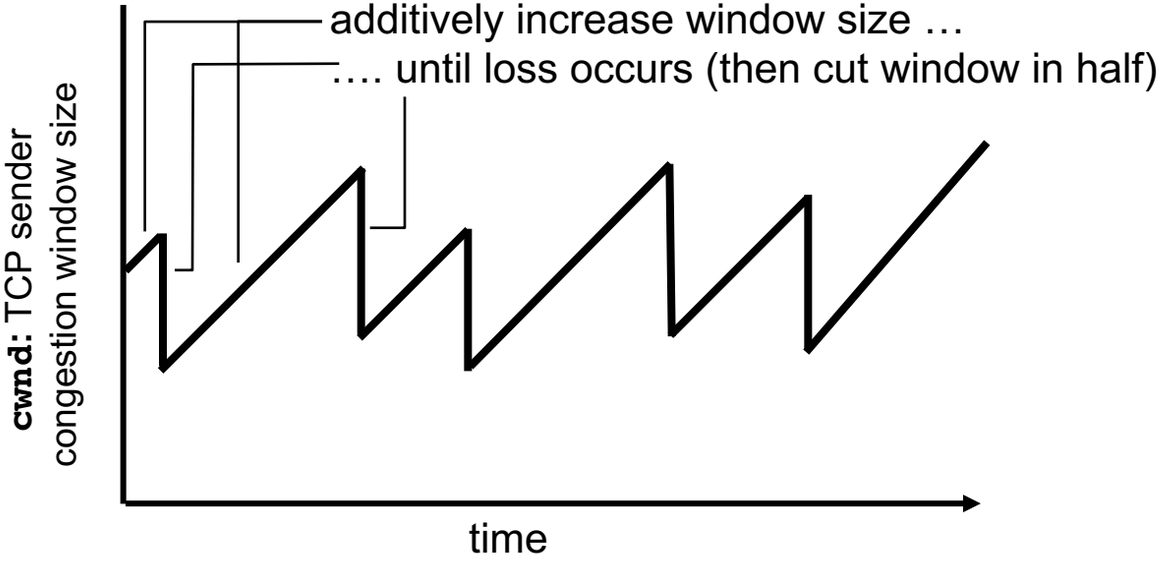
- What happens as λ_{in} and λ'_{in} increase?
- Arriving blue packets at upper queue are dropped, blue throughput → 0



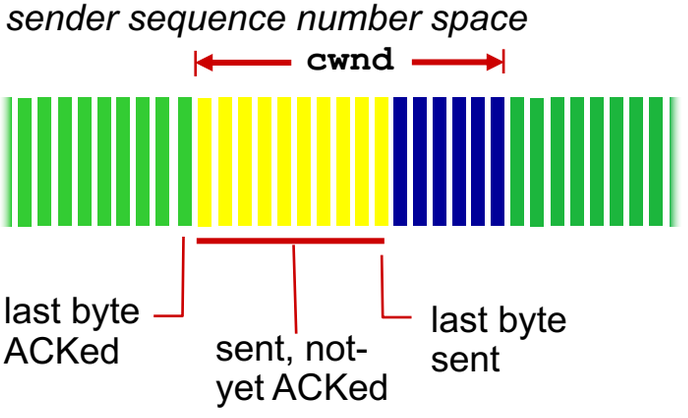
TCP Congestion Control: AIMD

- Approach: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - Additive Increase (AI): increase `cwnd` by 1 message every RTT until loss detected
  - Multiplicative Decrease (MD): cut `cwnd` by half after loss

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control: Details

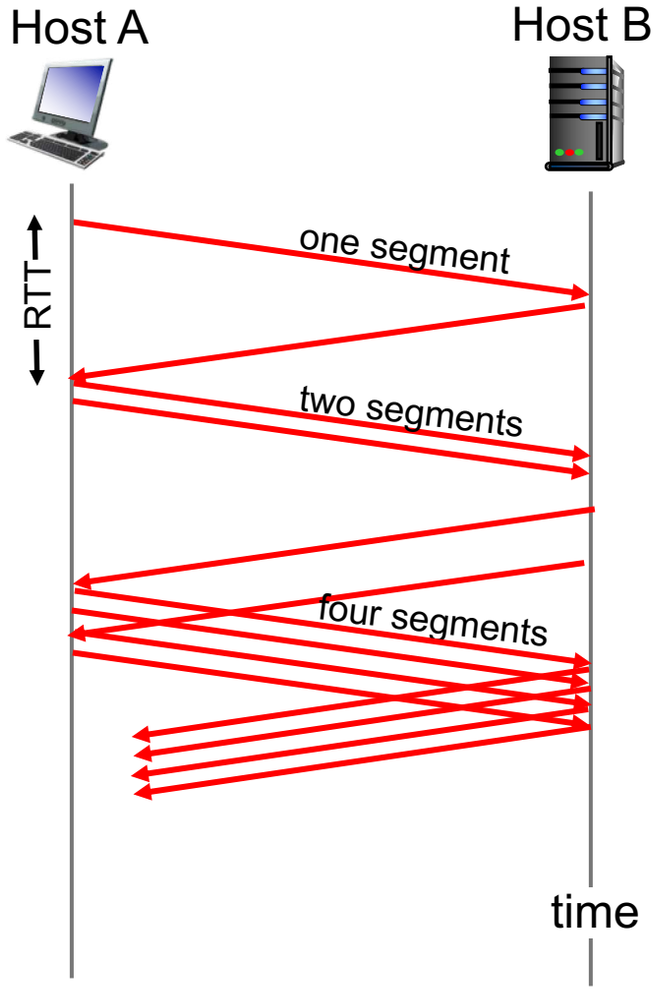


- Sender limits transmission
  - `LastByteSent - LastByteAcked ≤ cwnd`
- `cwnd` is dynamic, function of perceived network congestion
- TCP sending rate
  - Send `cwnd` bytes, wait `RTT` for ACKs, then send more bytes

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

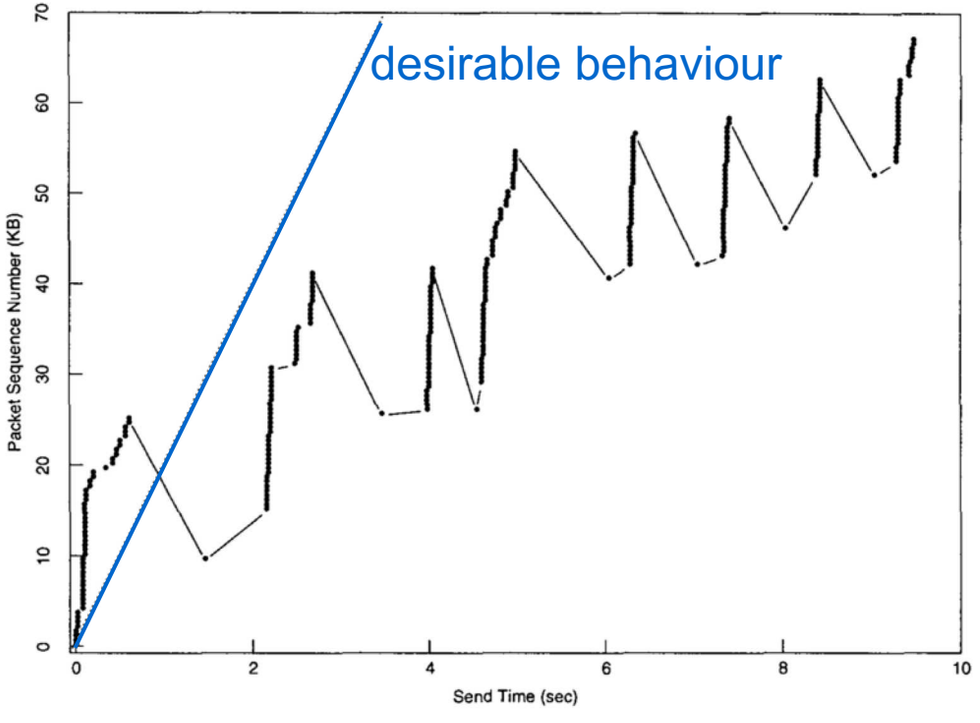
TCP Congestion Control: Slow Start

- When connection begins, increase rate exponentially until first loss event
  - Initially `cwnd` = 1 message
  - Double `cwnd` every RTT
  - Done by incrementing `cwnd` for every ACK received
- Initial rate is slow but ramps up exponentially fast



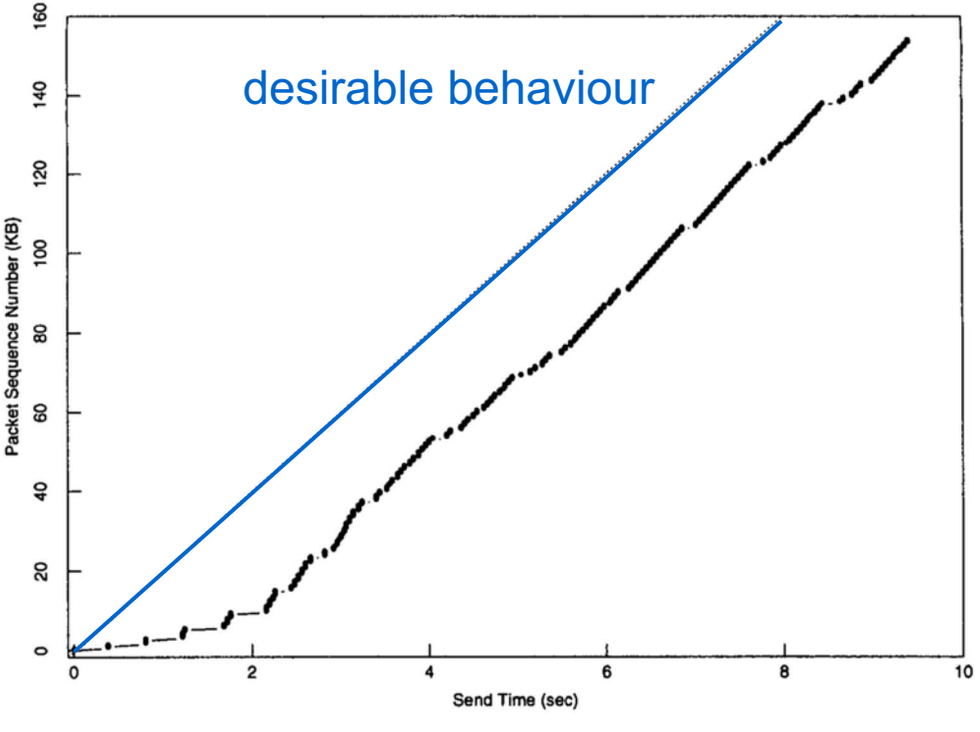
TCP: Effect of Slow Start

Van Jacobson. “Congestion Avoidance and Control”. SIGCOMM ’88.



Without slow start

With slow start

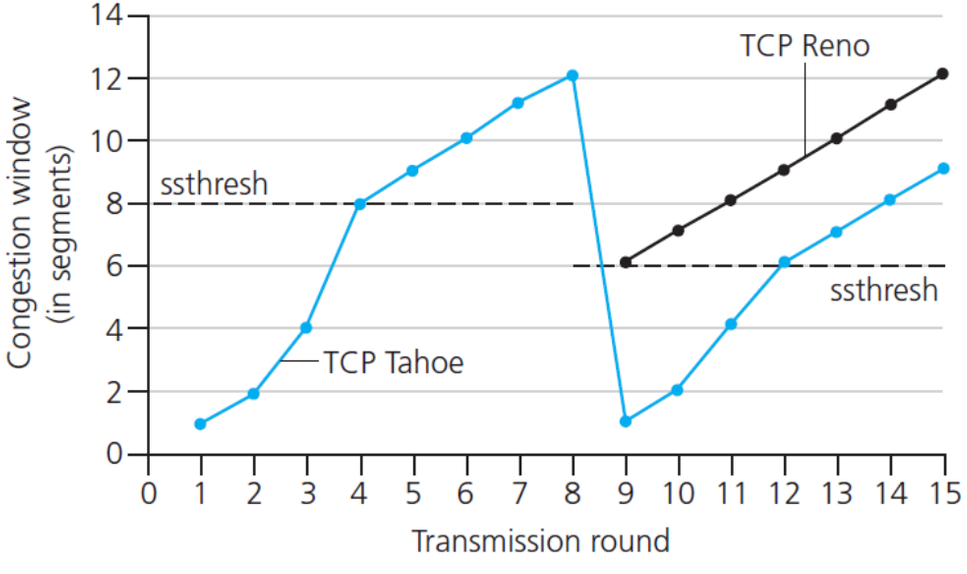


TCP Congestion Control: Reacting to Loss

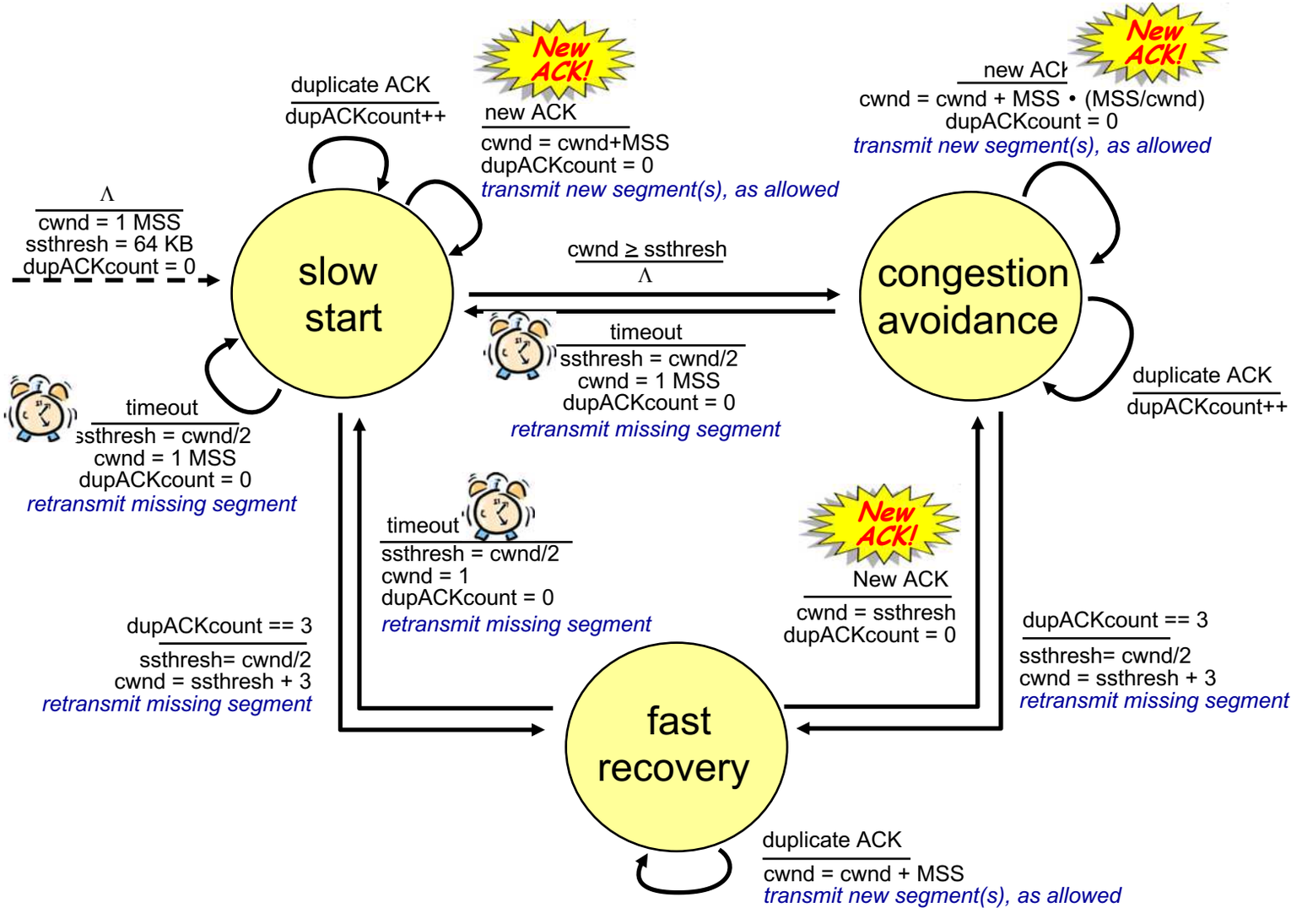
- Loss indicated by timeout
  - `cwnd` set to 1 message
  - Window then grows exponentially (as in slow start) to threshold, then grows linearly
- Loss indicated by 3 duplicate ACKs: TCP RENO
  - Duplicate ACKs indicate network capable of delivering some segments
  - `cwnd` is cut in half window then grows linearly
- TCP Tahoe always sets `cwnd` to 1 (timeout or 3 duplicate acks)

TCP Congestion Control: Switching Strategy

- When should the exponential increase switch to linear?
  - When `cwnd` gets to ½ of its value before timeout
- Implementation
  - Variable `ssthresh`
  - On loss event, `ssthresh` is set to ½ of `cwnd` just before loss event



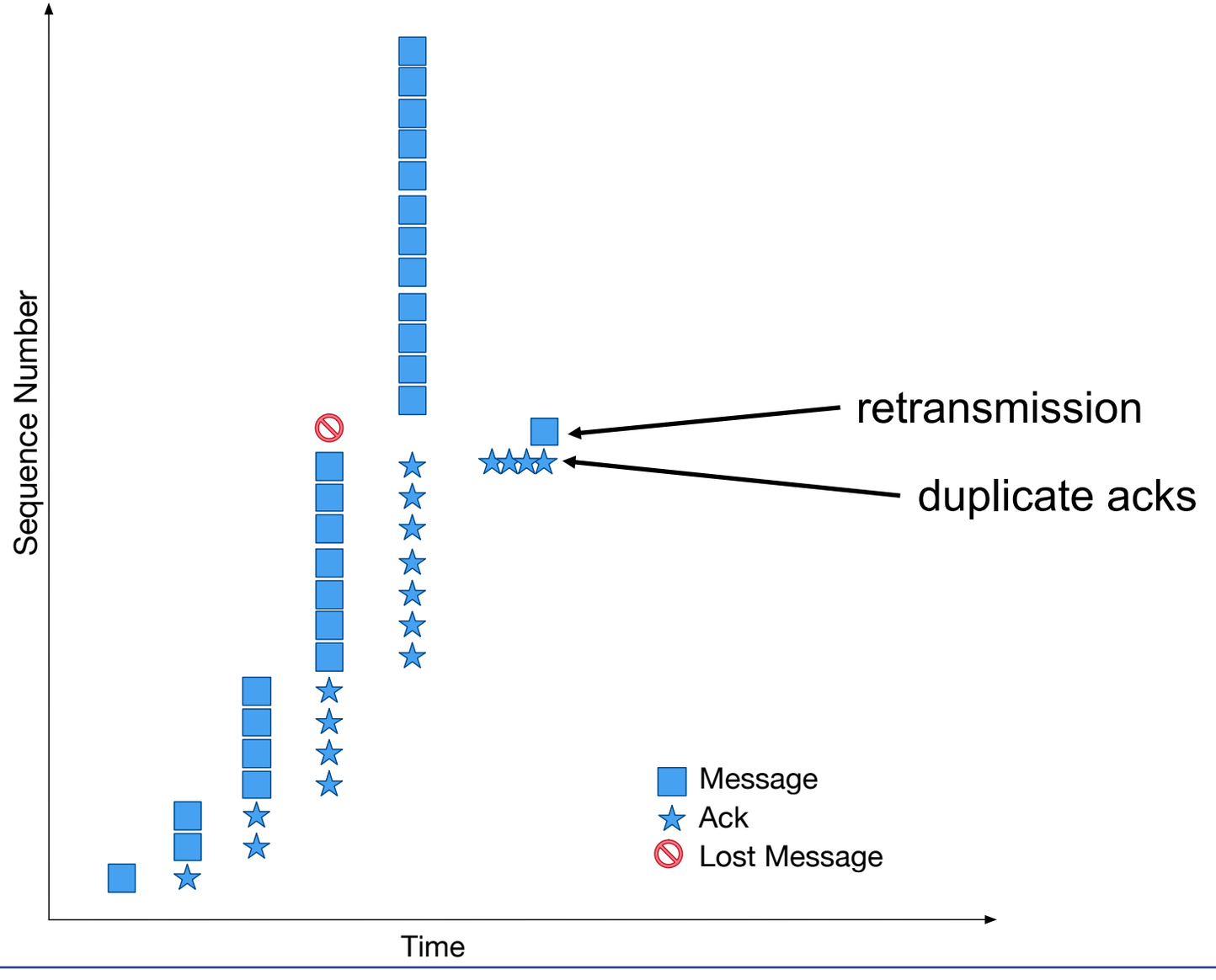
TCP Congestion Control: Summary



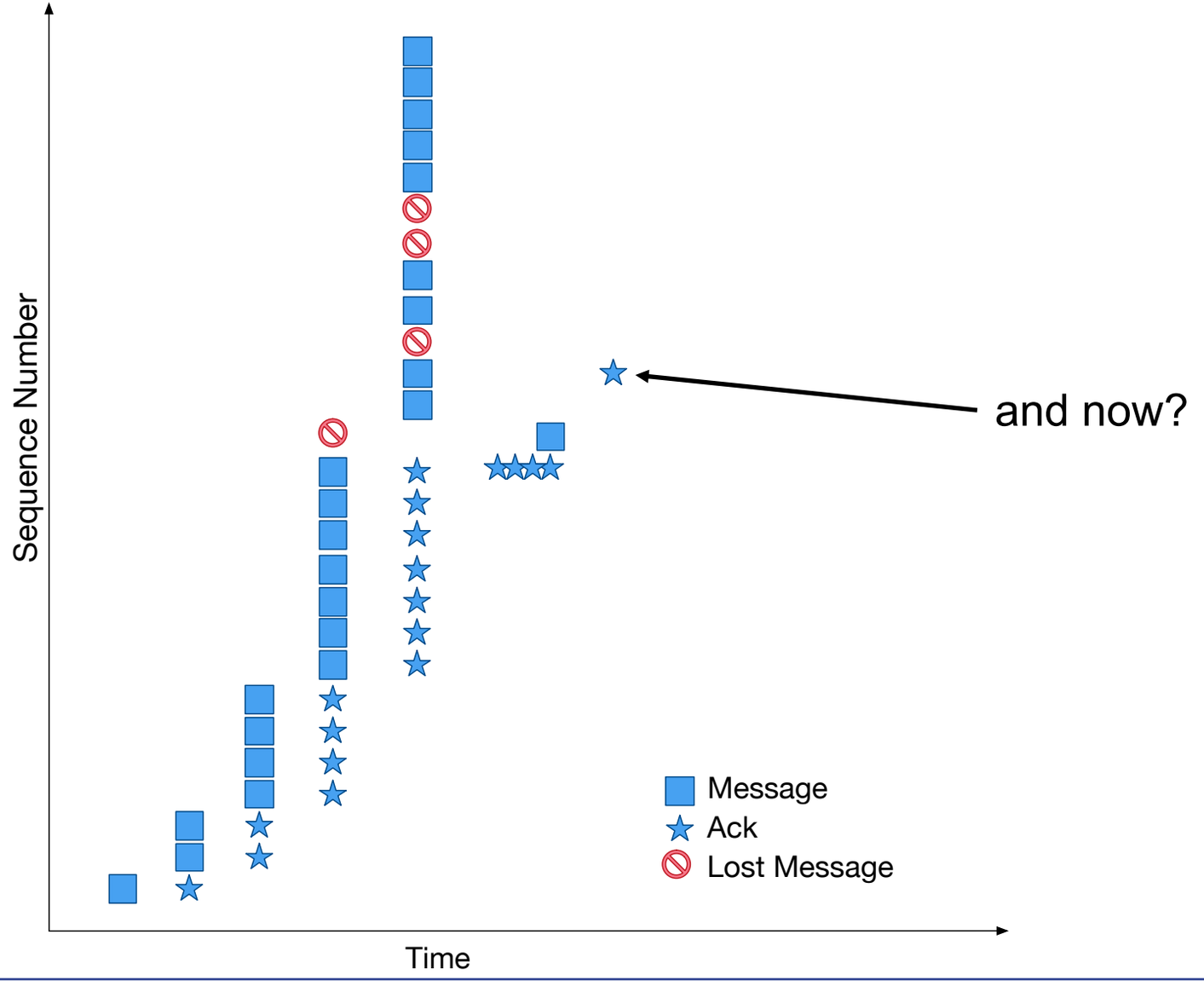
TCP Flavors

- Tahoe, Reno, New Reno, Vegas
- TCP Tahoe
  - The first widely used implementation of TCP that addressed congestion issues
  - Features:
    - Slow-start
    - Congestion avoidance
    - Fast retransmit

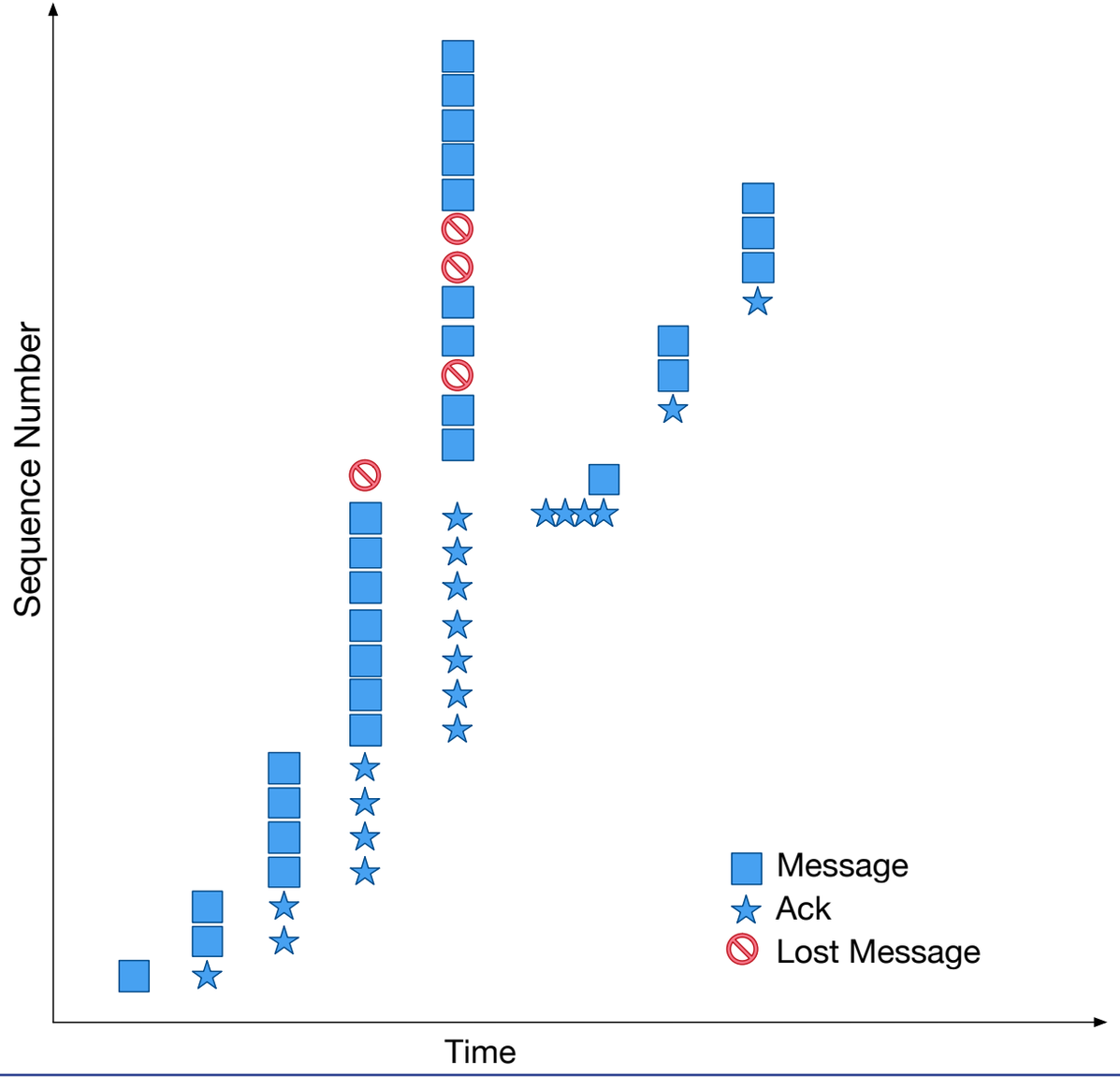
TCP: Tahoe Fast Retransmit



TCP: Multiple Losses



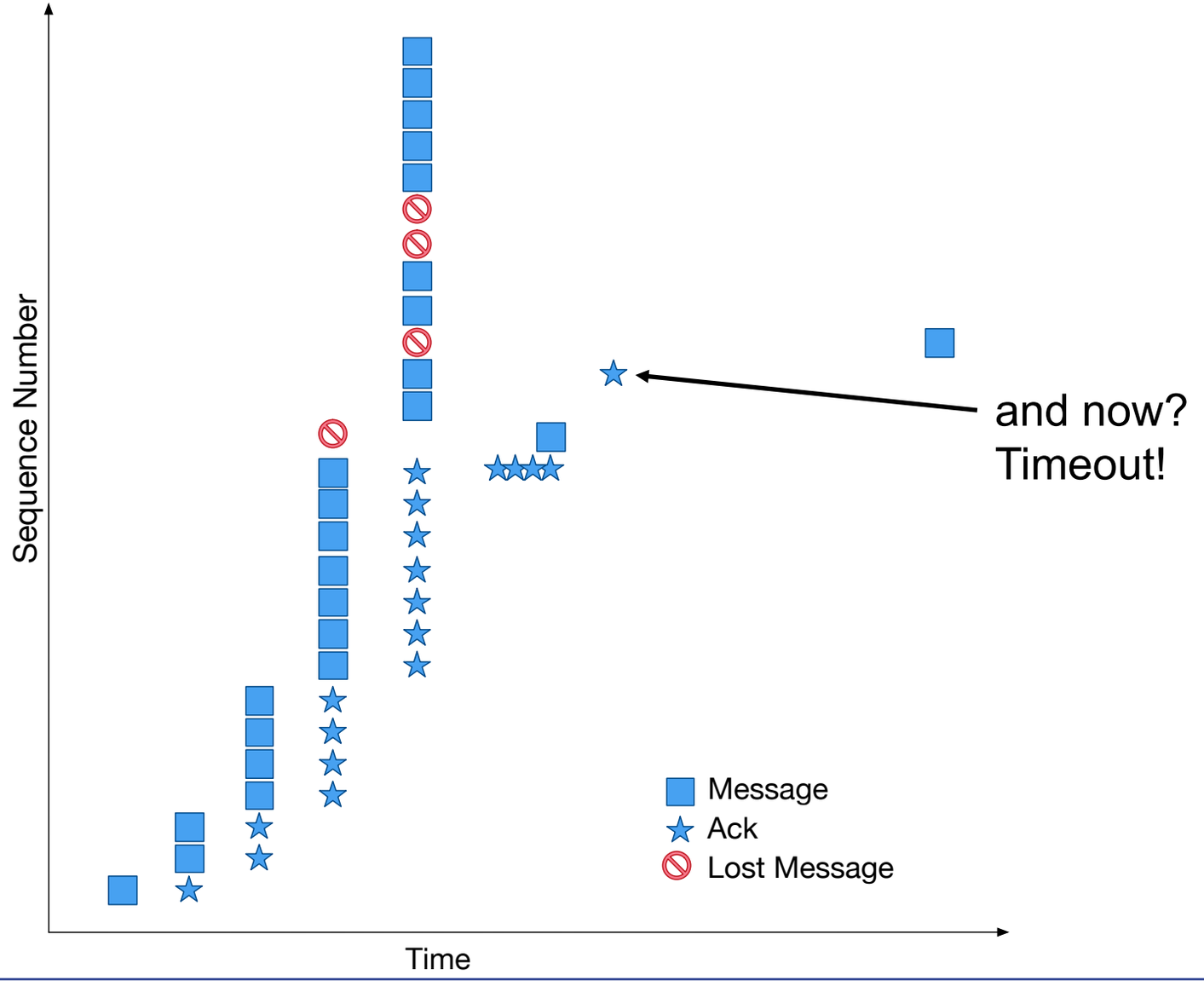
TCP: Tahoe



TCP: Reno

- All mechanisms as in Tahoe
- Fast recovery, if three duplicate ACKs:
  - Halve the congestion window
  - Set the slow start threshold equal to the new congestion window
  - Skip slow start
- If an ACK times out
  - Slow start is used
  - Congestion window set to 1 (as in Tahoe)

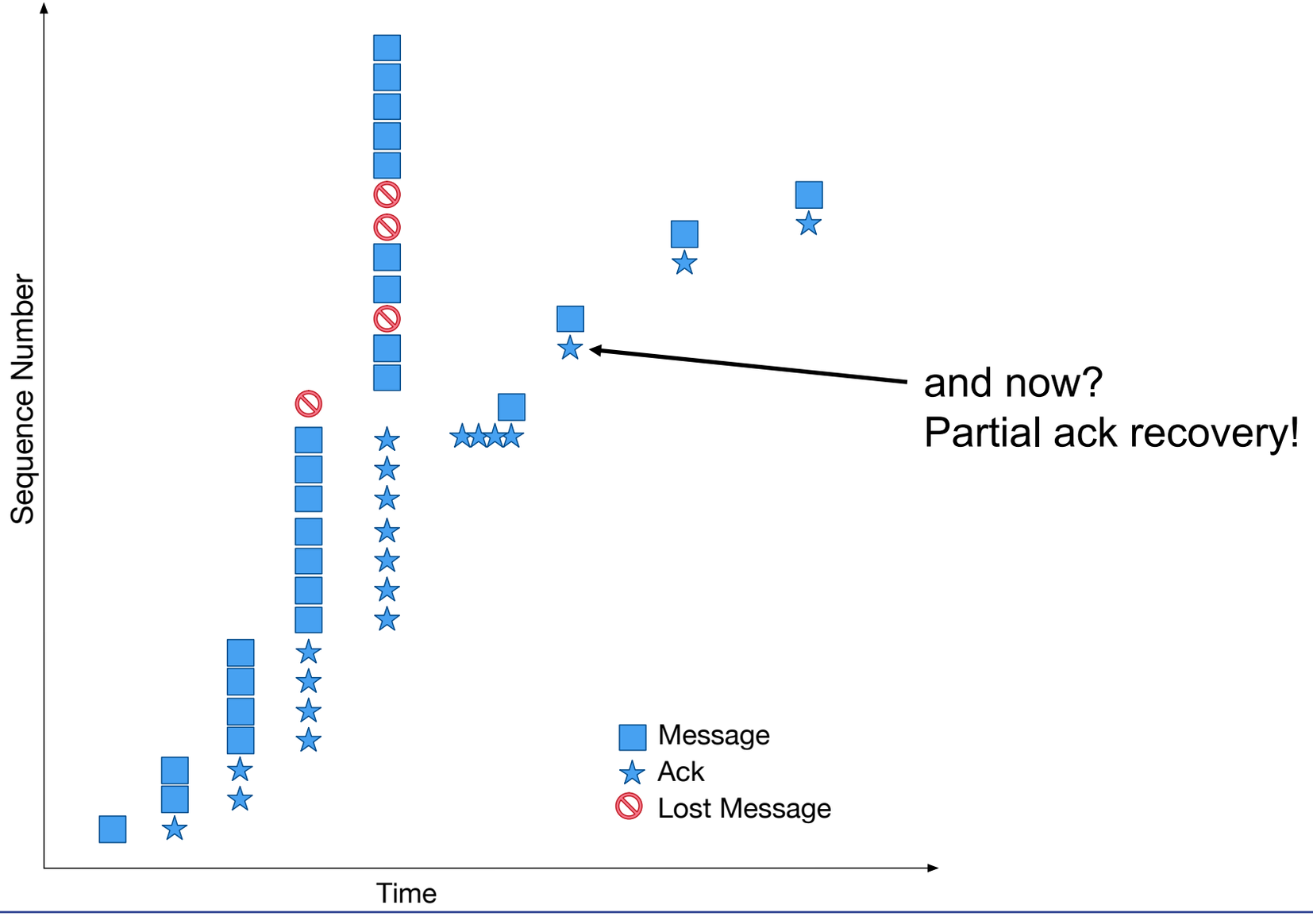
TCP: Reno


Sequence Number
Time
and now?
Timeout!
Message
Ack
Lost Message

TCP: New Reno

- Improved fast recovery
  - For every duplicate ack, an unsent packet from the end of the congestion window is sent
- Reaction to partial acks
  - Only the next packet (with the sequence number after the acked one) is transmitted
- When does it timeout?
  - Fewer than three duplicate acks
  - If a partial ack is lost

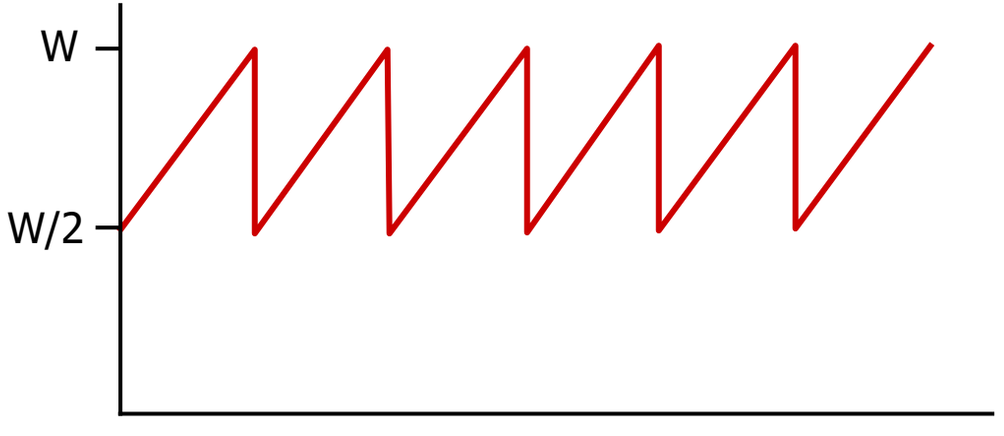
TCP: New Reno


Sequence Number
Time
and now?
Partial ack recovery!
Message
Ack
Lost Message

TCP: Throughput

- Average TCP throughput as function of window size and RTT
  - Ignore slow start, assume always data to send
- W: window size (in bytes) where loss occurs
  - Average window size (number of bytes “in-flight”) is ¾ W

$$\text{avg. TCP throughput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$

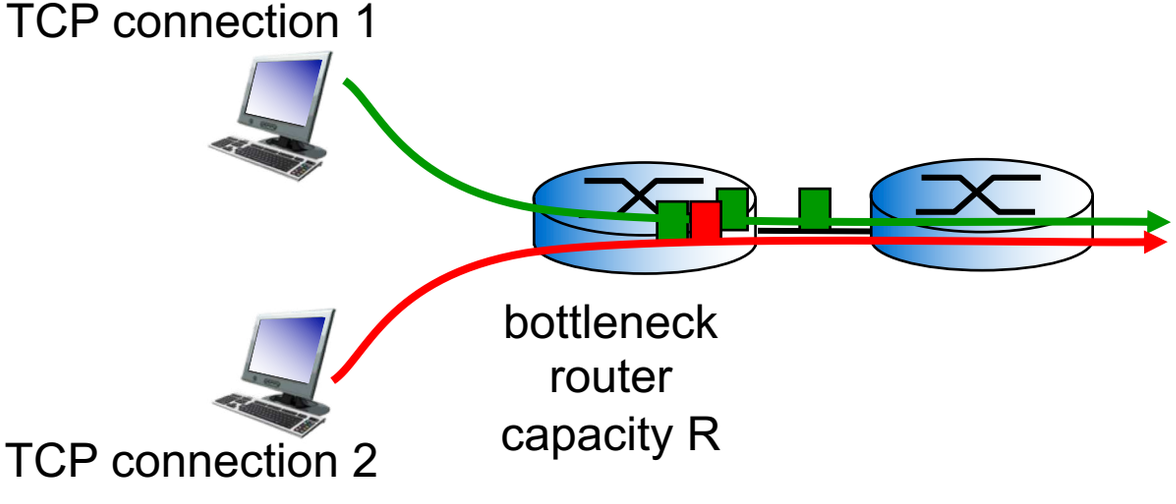


TCP Futures: TCP over "long, fat pipes"

- Internet has evolved since the SMTP, FTP, Telnet times…
- Example: 1500 byte segments, 100 ms RTT, desired 10 Gbps throughput
  - Requires W = 83,333 in-flight segments
  - Throughput in terms of segment loss probability, L, and maximum segment size, MSS [Mathis 1997]
    - TCP throughput = $\frac{1.22 * MSS}{RTT * \sqrt{L}}$
    - To achieve 10 Gbps throughput with aforementioned scheme, need a loss rate of $L = 2 * 10^{-10}$ (very, very small!)
- New versions of TCP for high-speed have been investigated

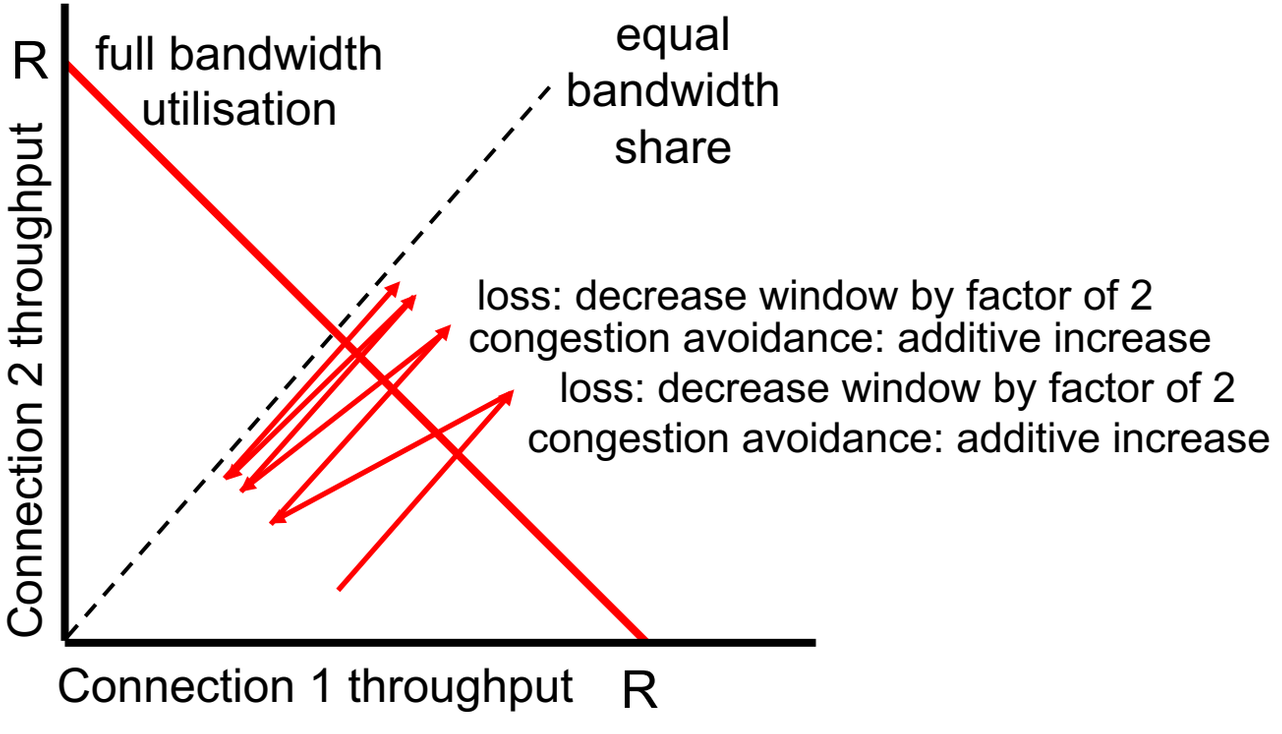
TCP: Fairness

- Fairness goal
  - If K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



TCP: Why Is It Fair?

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



TCP: More on Fairness

- Fairness and UDP
  - Multimedia apps often do not use TCP
    - Do not want rate throttled by congestion control
  - Instead with UDP
    - Send audio/video at constant rate, tolerate packet loss
- Fairness and parallel TCP connections
  - Application can open multiple parallel connections between two hosts
  - Web browsers do this
  - E.g., if on link of rate R with 9 existing connections
    - A new app asks for 1 TCP and gets rate R/10
    - The same app asks for 11 TCP and gets more than R/2

TCP: Explicit Congestion Notification (ECN)

- Network-assisted congestion control
  - Two bits in IP header (ToS field) marked by network router to indicate congestion
  - Congestion indication carried to receiving host
  - Receiver (seeing congestion indication in IP datagram) sets ECE bit on receiver-to-sender ACK segment to notify sender of congestion