# CSCE 560
# Introduction to
# Computer Networking

Dr. Barry Mullins
AFIT/ENG
Bldg 642, Room 209
255-3636 x7979

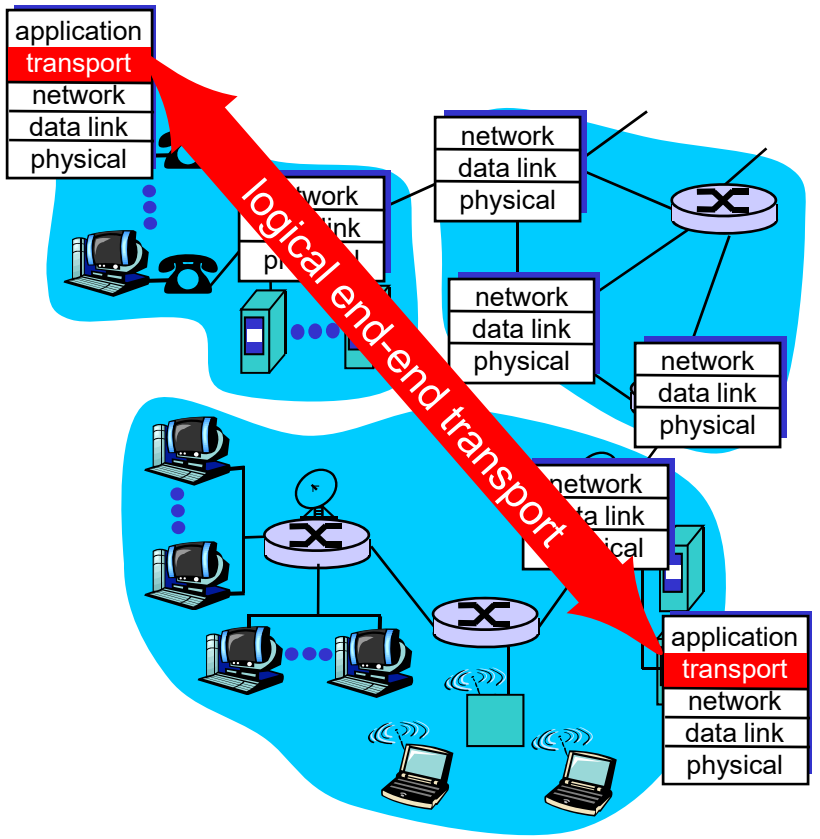
President George W. Bush stands with Presidential Medal of Freedom recipients, **Vinton G. Cerf** and **Robert E. Kahn**, Nov. 9, 2005, during ceremonies at the White House. Cerf and Kahn were co-designers of the TCP/IP Internet network protocol.

# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

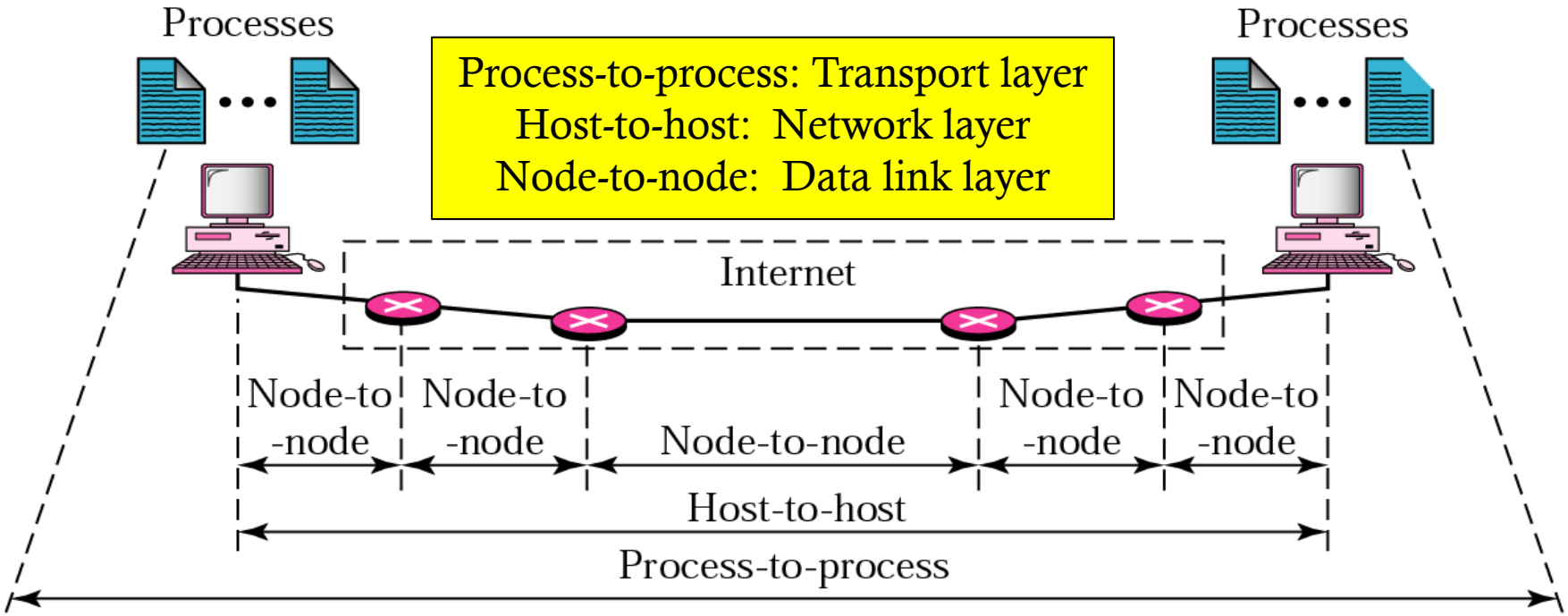
# Transport Services and Protocols

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



# Transport vs. Network Layer

- Transport: logical end-to-end communication between processes
  - Relies on, enhances, network layer services
- Network: logical communication between hosts



# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

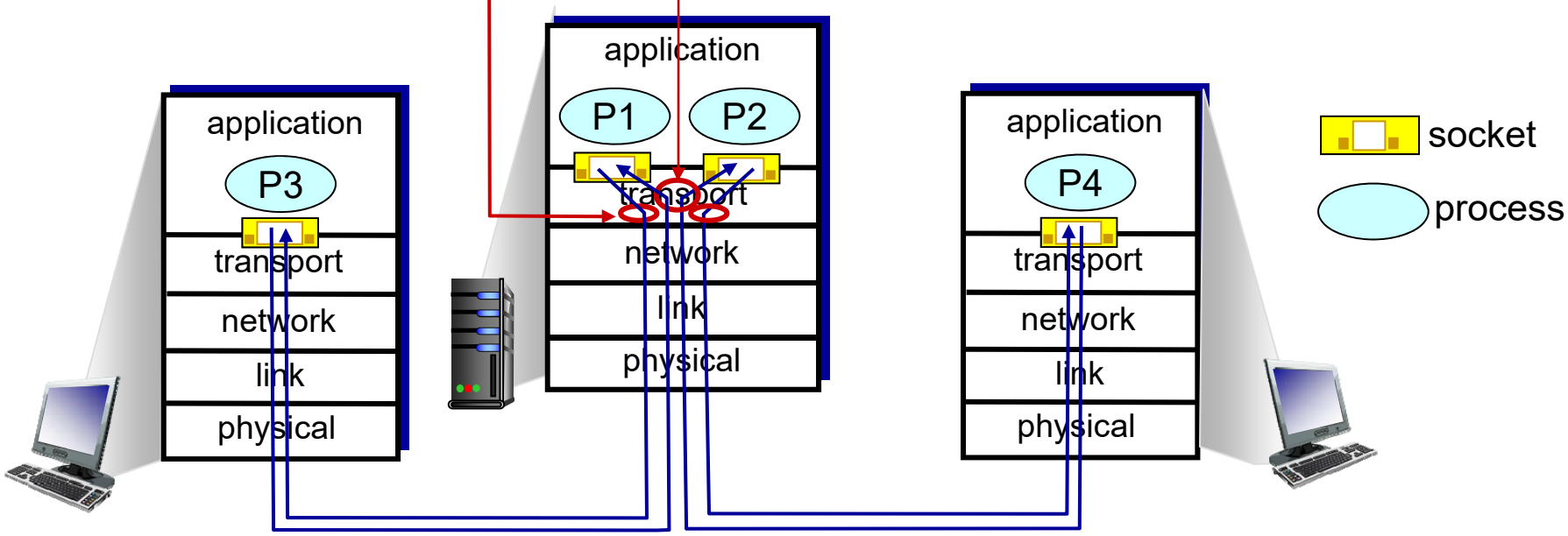
# Multiplexing / Demultiplexing

*Multiplexing at sender:*

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

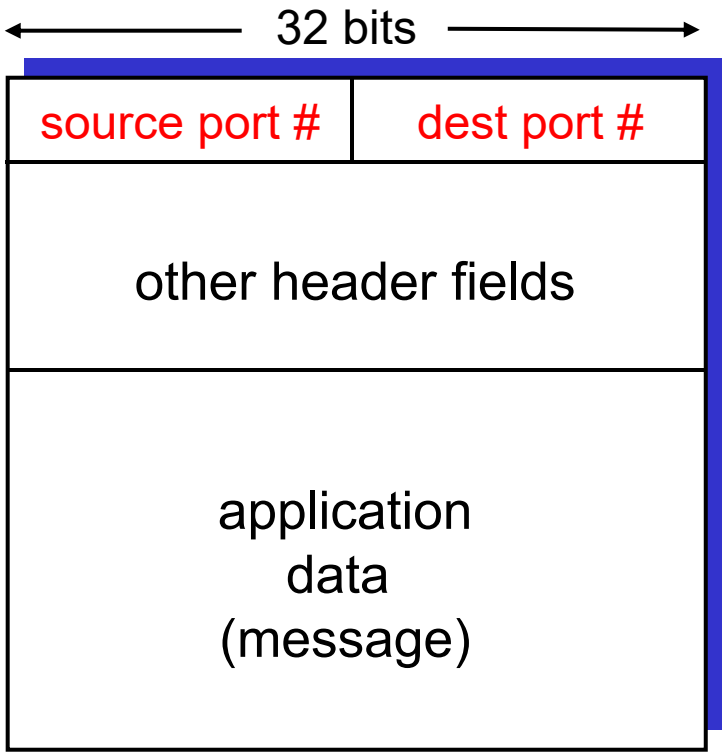
*Demultiplexing at receiver:*

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 1 transport-layer segment
- Each TCP/UDP segment has
  - Source port number
  - Destination port number
- Destination uses IP addresses & port numbers to direct segment to appropriate socket




TCP / UDP segment format

# Connectionless (UDP) Demultiplexing

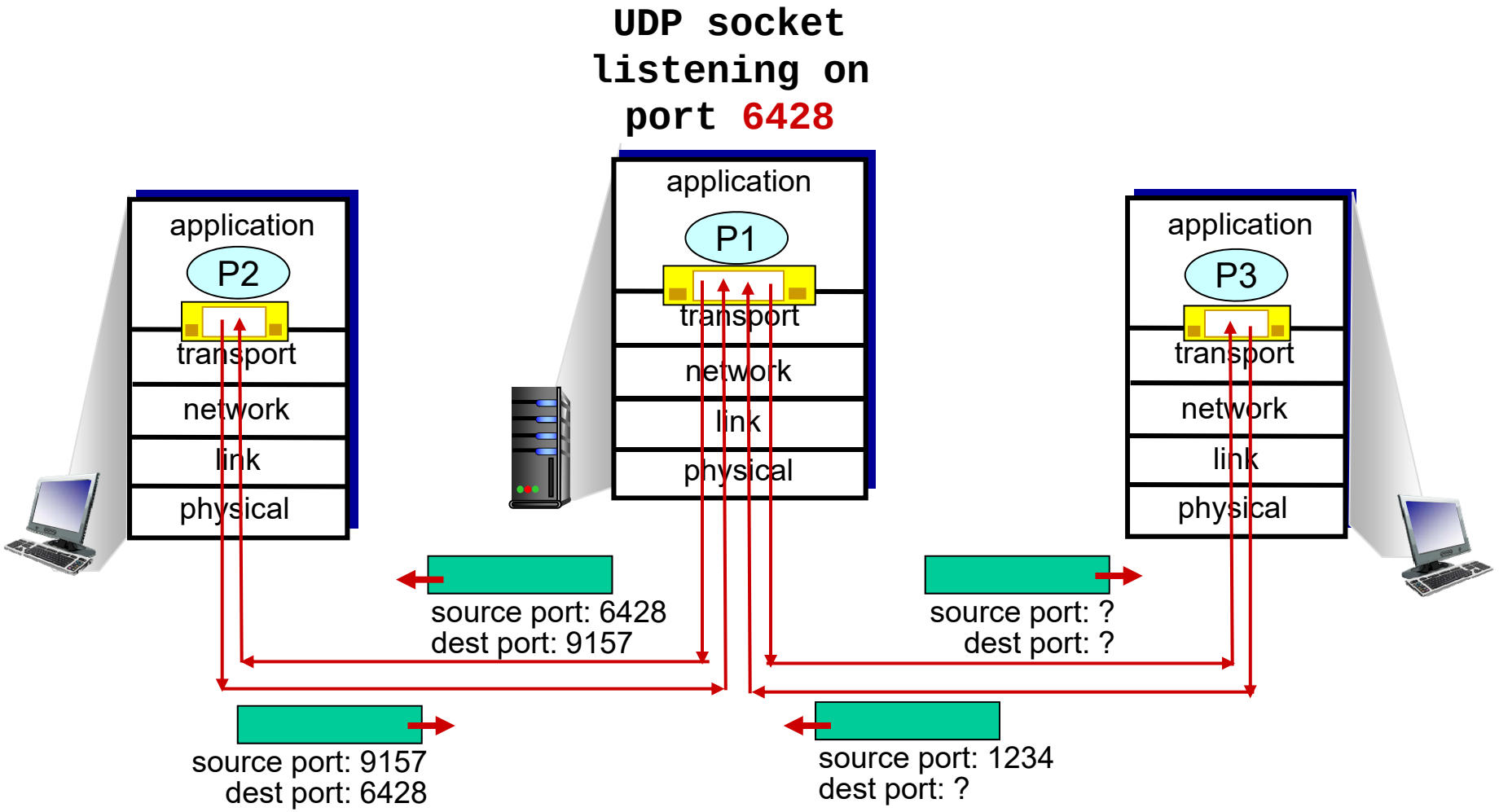
- Create sockets with port numbers:

```
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', 6428))
```

- UDP socket identified by two-tuple:
  - dest IP address
  - dest port number

- When host receives UDP segment:
  - Checks destination port number in segment
  - Directs UDP segment to socket with that port number …
    - even if IP datagrams came from different sources
      - different source IP addresses and/or source port numbers

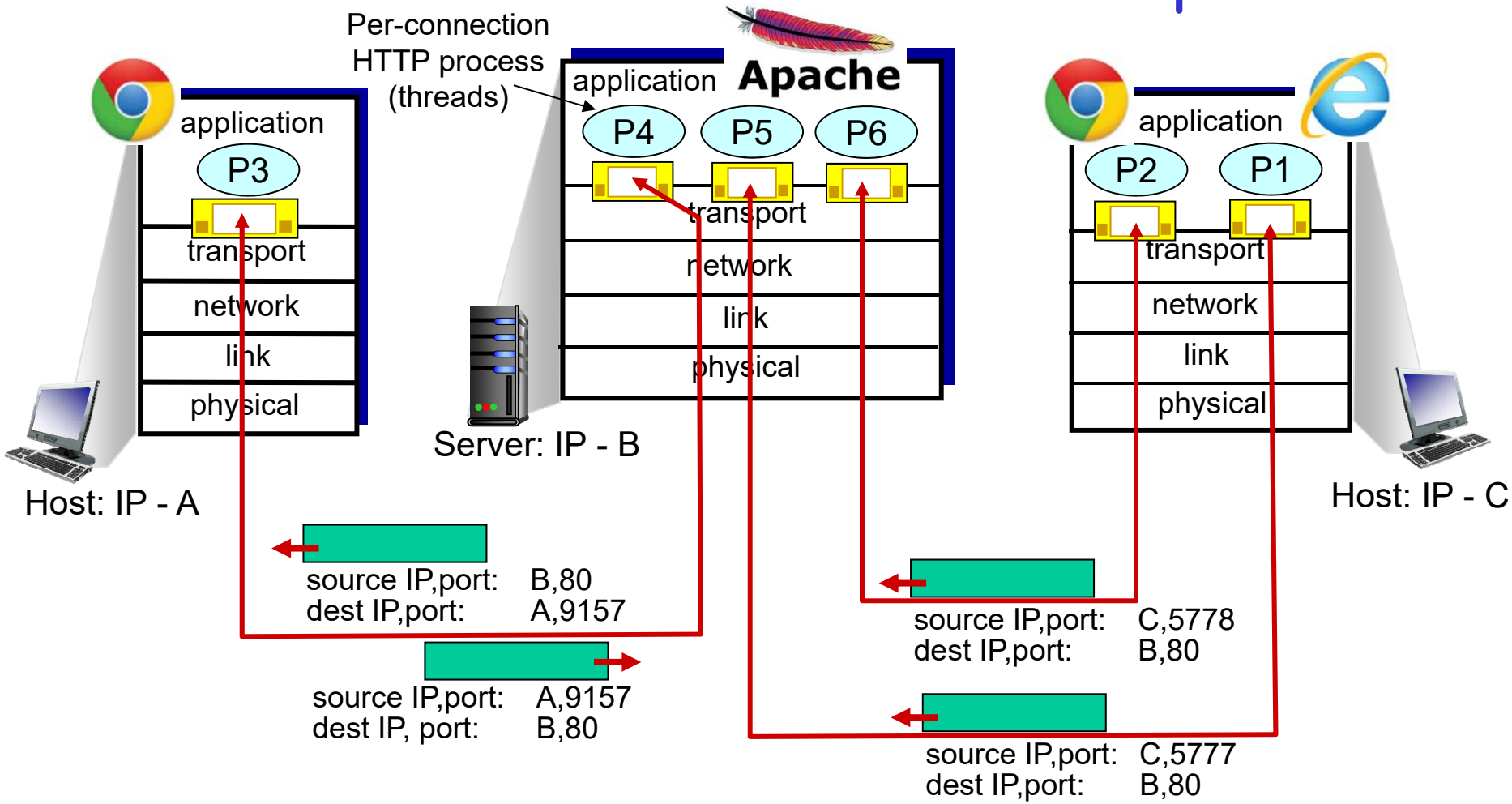
# Connectionless Demux: Example



# Connection-oriented (TCP) Demux

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Receiver uses all four values to direct segment to appropriate socket
- Servers support many simultaneous TCP sockets:
  - Each socket identified by its own 4-tuple
- Web servers have different sockets for each TCP connection from a client
  - Non-persistent HTTP will have a different socket for each object request!

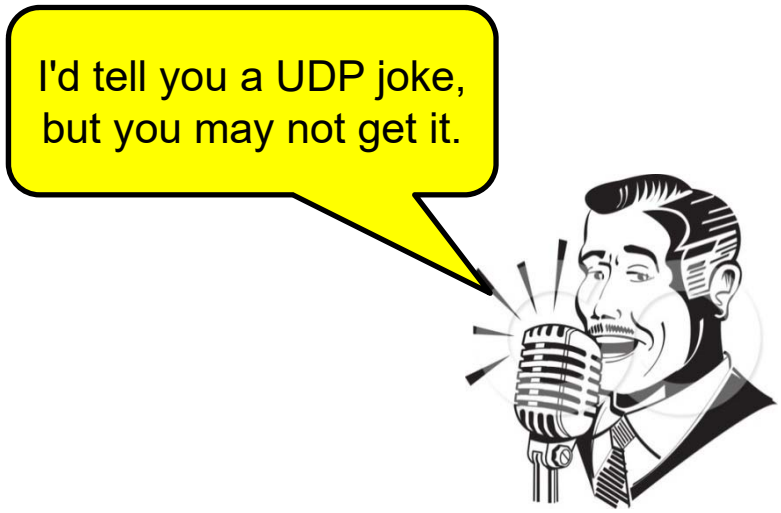
# Connection-oriented Demux: Example




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

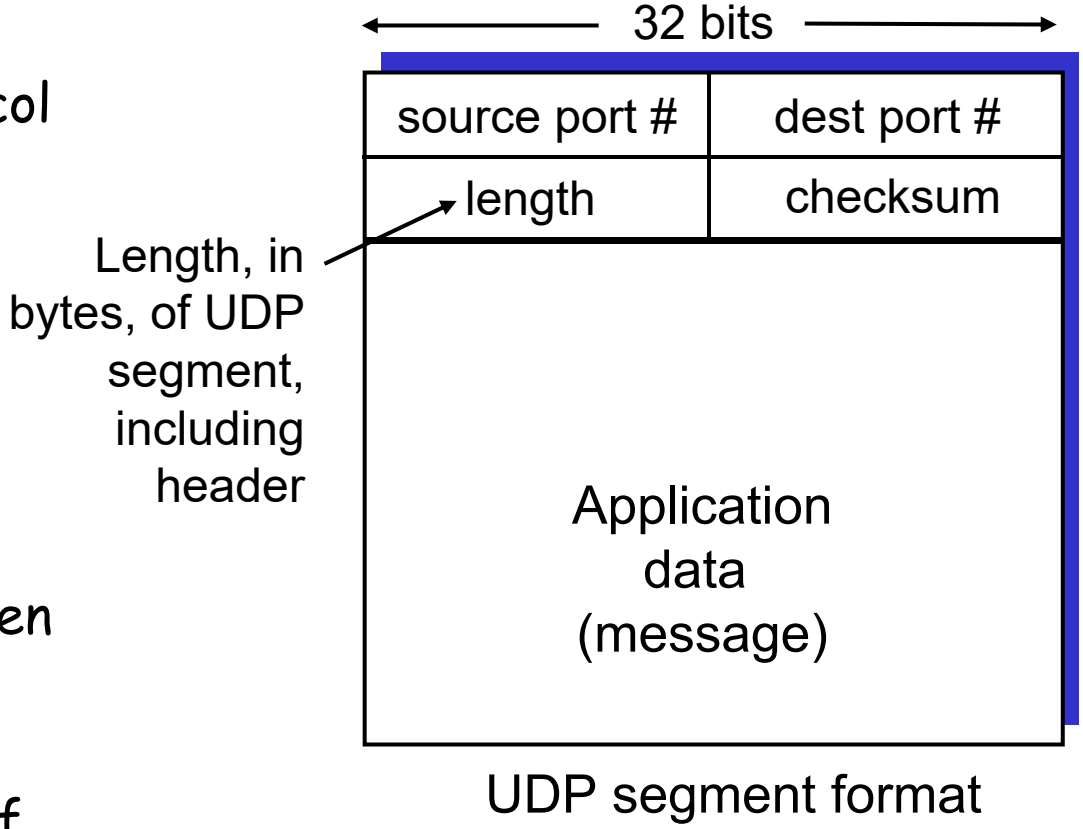
# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control




# 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 independent of others




UDP segment format

# So Why is There a UDP?

- Application can control exactly what data is sent and when
  - UDP passes data to network layer immediately
  - No congestion control: UDP can blast away as fast as desired
- No connection set up (which can add delay)
- Simple: no connection state at sender, receiver
  - No buffers and fewer variables to maintain
  - Server can support more UDP clients than TDP
- Small segment header
  - 8 bytes instead of 20 for TCP
- You want reliable transfer over UDP?
  - Add reliability at the application layer
  - Application-specific error recovery!

# UDP Checksum

Goal: detect "errors" (e.g., flipped bits) in received segment

**Sender:**

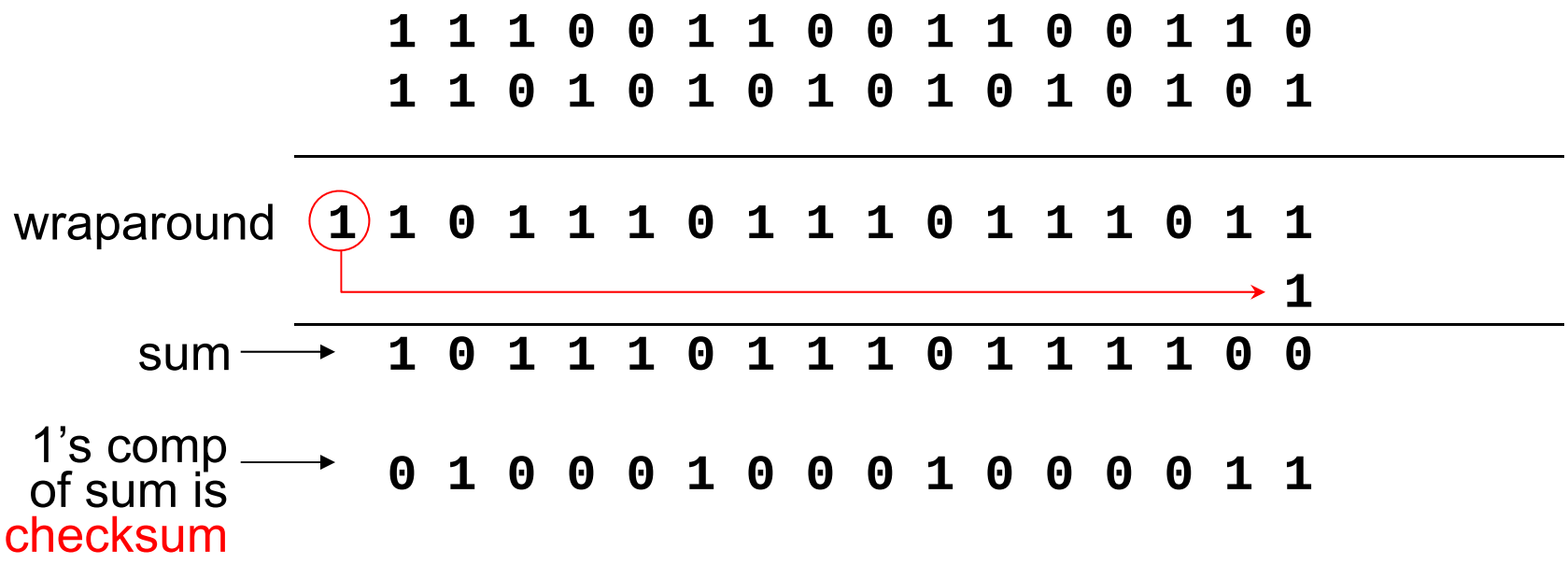
- Treat entire segment contents as sequence of 16-bit integers
- Checksum = 1's complement of sum of segment contents
- Sender puts checksum value into UDP checksum field

**Receiver:**

- Add all 16-bit integers plus the checksum of received segment
- If result is FFFF:
  - No error detected
    - *But there may be errors nonetheless? More later*
- If result contains a 0: error!
  - Discard segment (UDP) or
  - Pass segment to app with warning
    - UDP Lite - RFC 3828

# Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

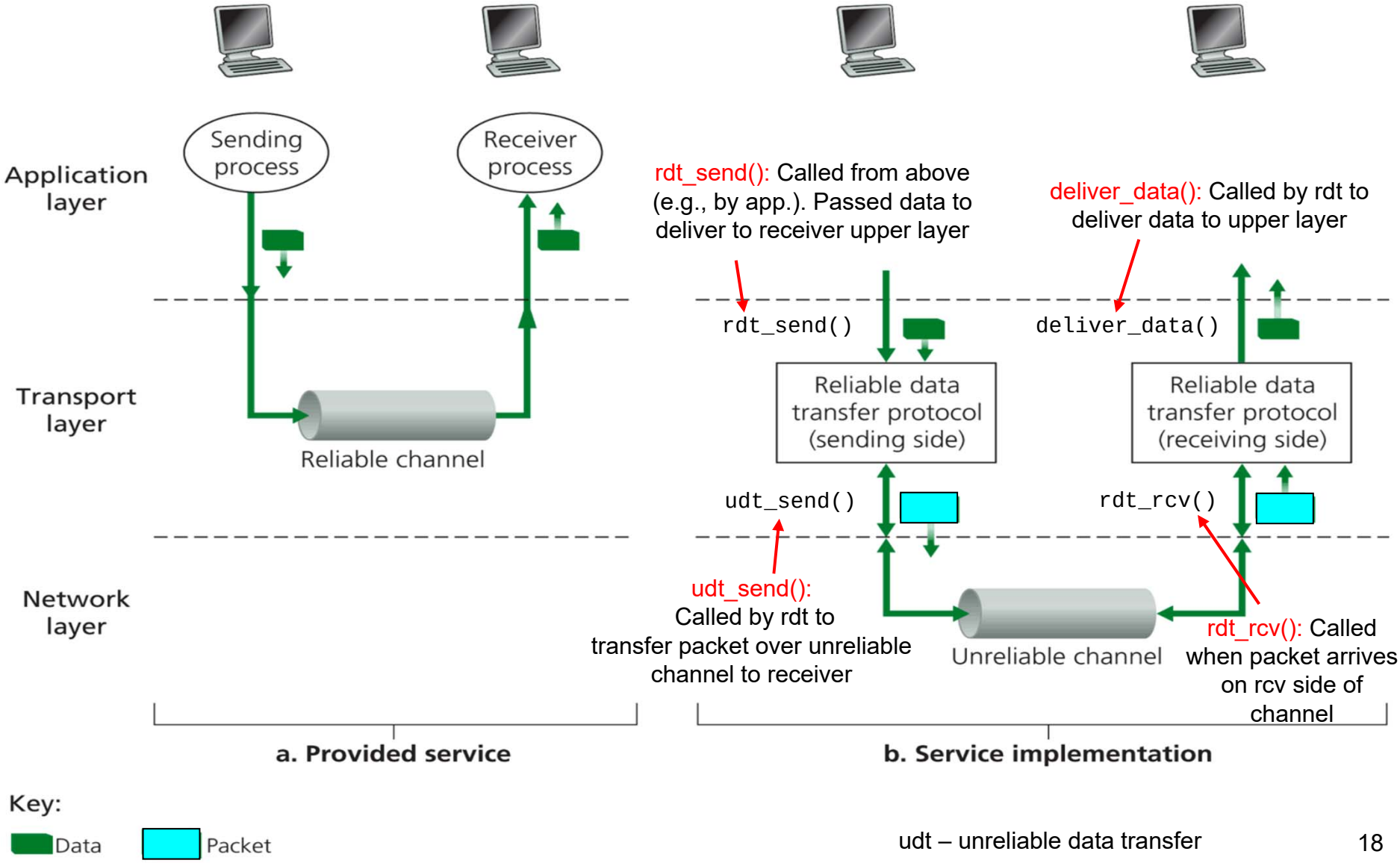


# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# Principles of Reliable Data Transfer (rdt)

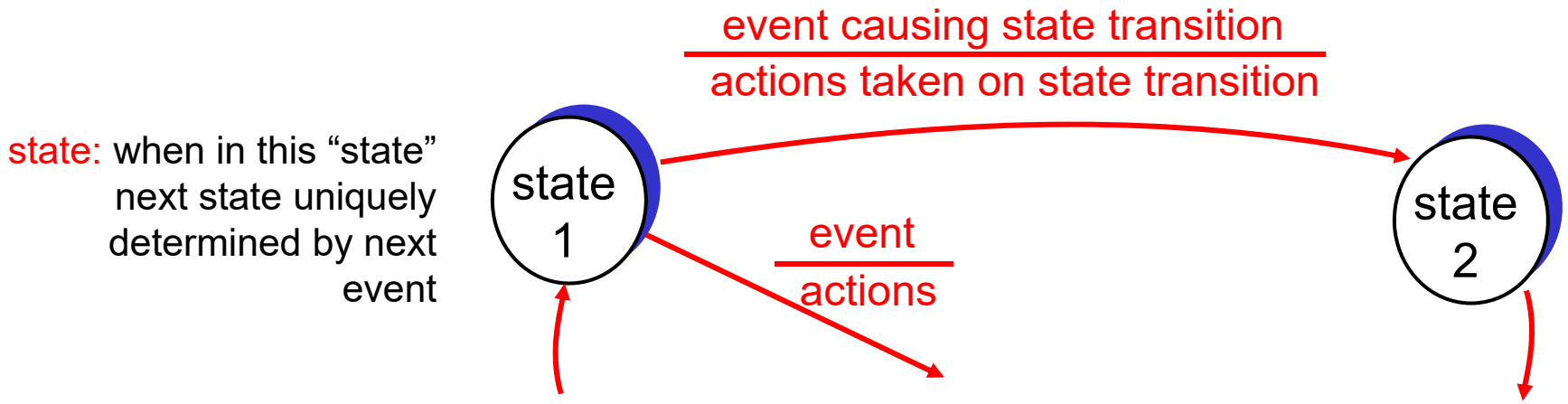
- Important in application, transport, and link layers
- Top-10 list of important networking topics!



# Reliable Data Transfer: Getting Started

We'll:

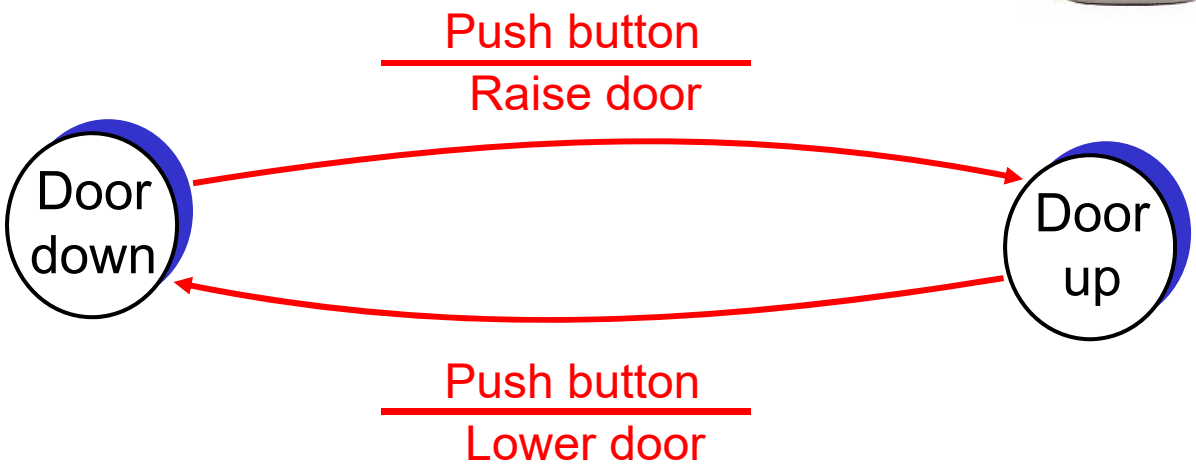
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - but control info will flow in both directions!
- Use finite state machines (FSM) to specify sender, receiver



# Reliable Data Transfer: Example

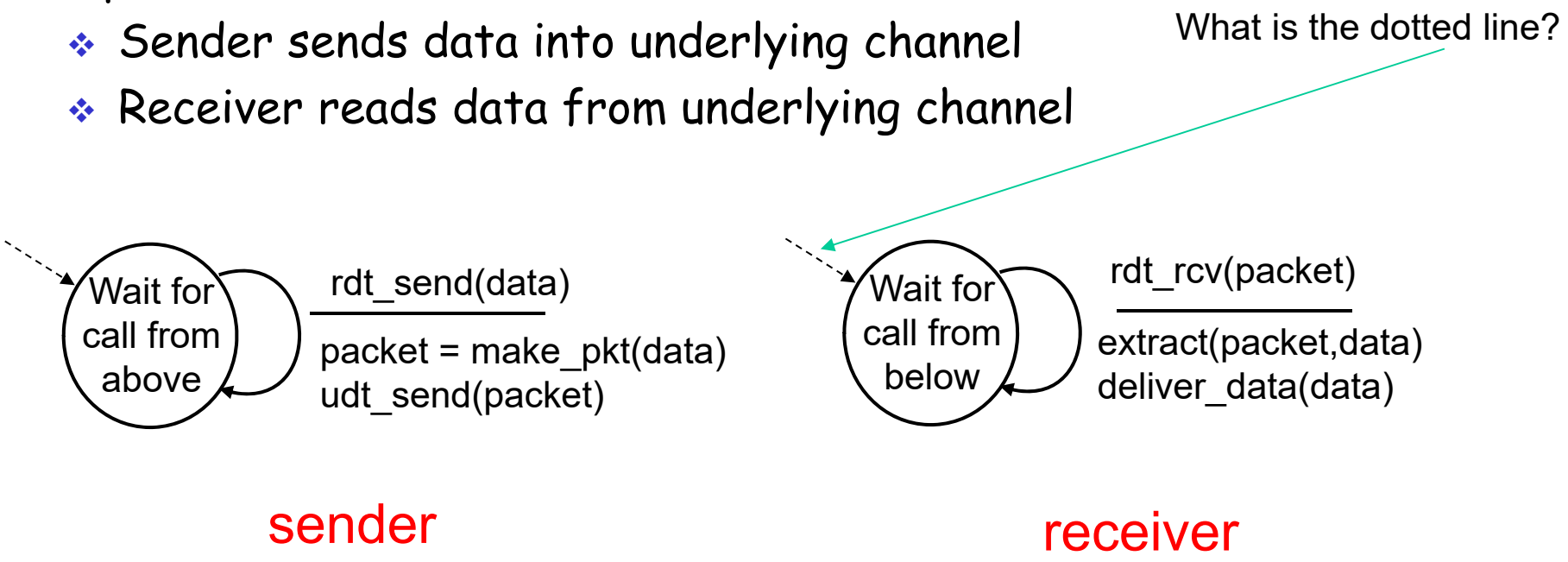
What happens when you push the garage door remote button?


state: Is the garage door up or down



# rdt1.0: Reliable Transfer Over a Reliable Channel

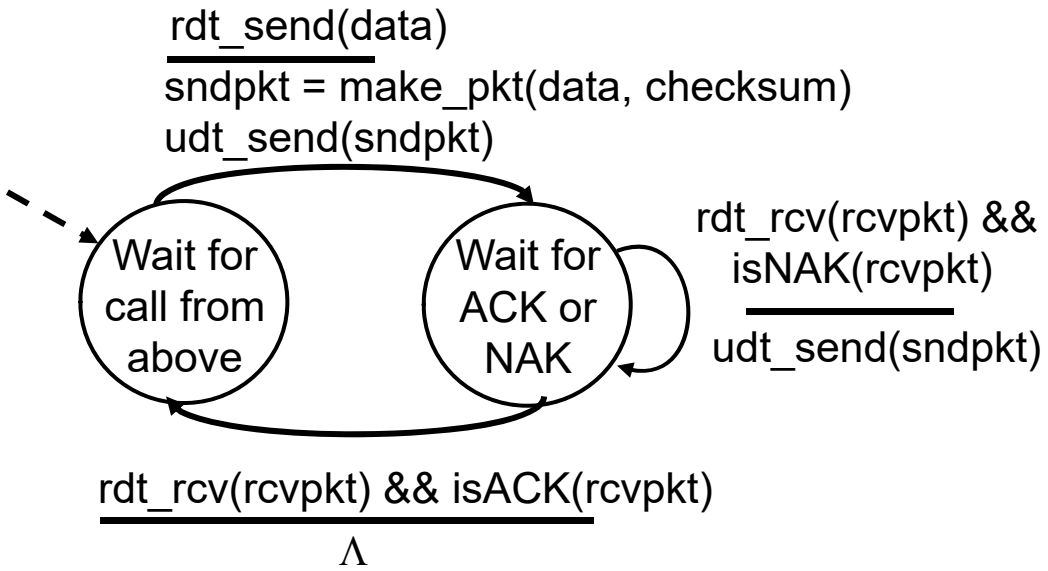
- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
  - No flow control required
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



# rdt2.0: Channel With Bit Errors

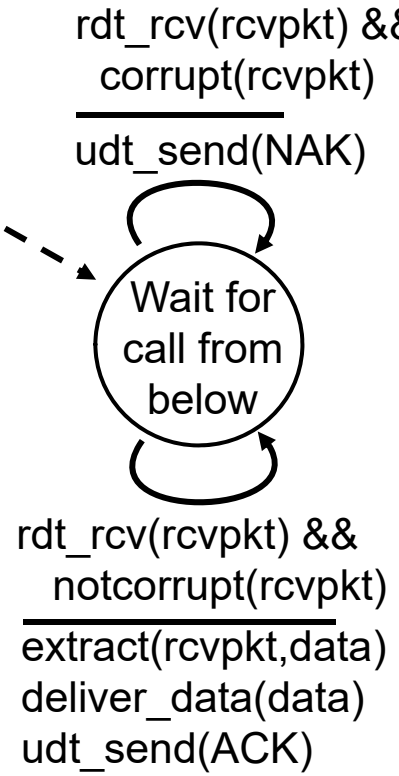
- Underlying channel may flip bits in packet
  - Use checksum to detect bit errors
- The question: how to recover from errors:
  - Acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - Negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors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

# rdt2.0: FSM Specification

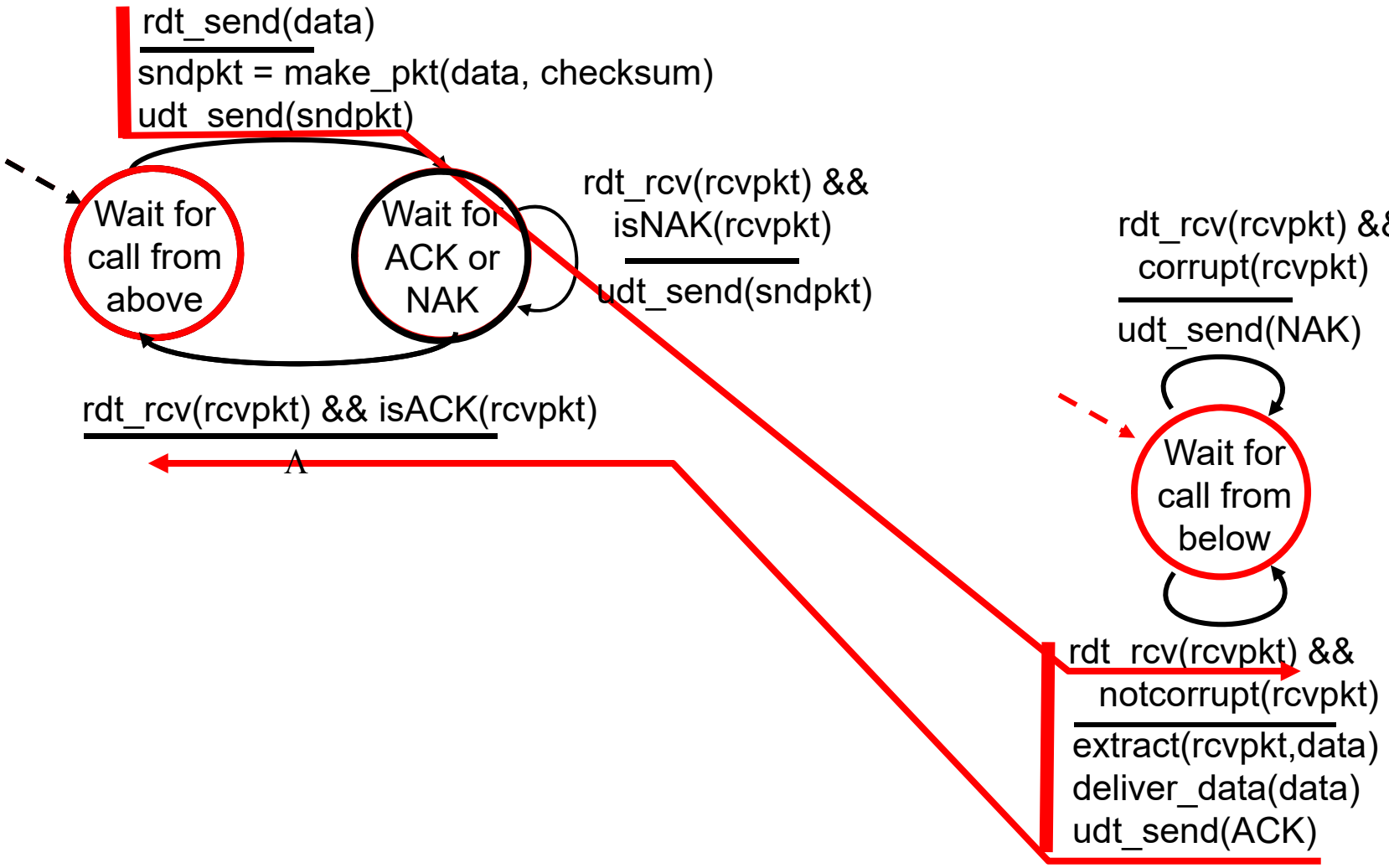



sender

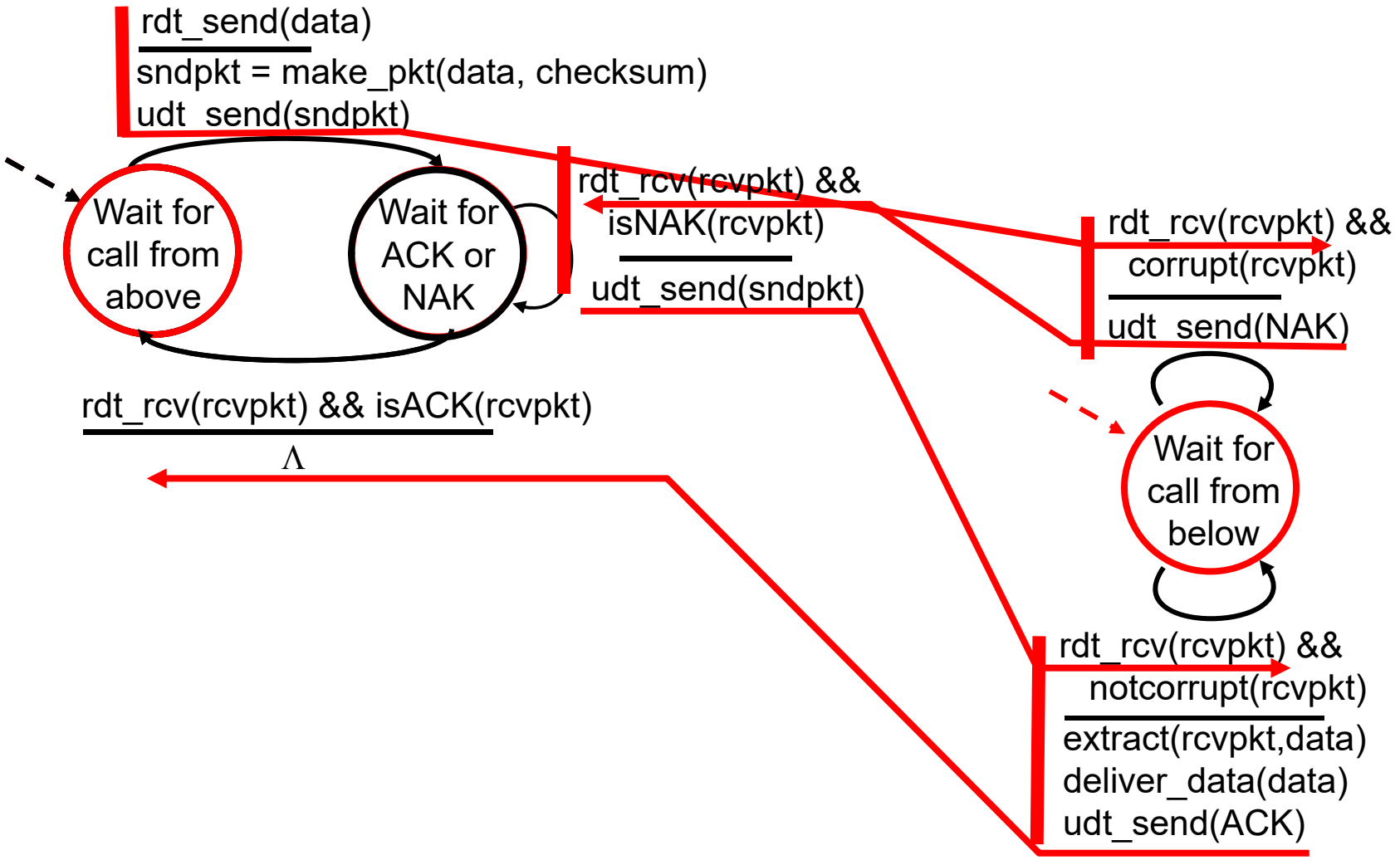
receiver



# rdt2.0: Operation With No Errors



# rdt2.0: Error Scenario



# rdt2.0 Has a Fatal Flaw!

What if ACK/NAK corrupted?

- Sender doesn't know what happened at receiver!
- Can't just retransmit: possible duplicate

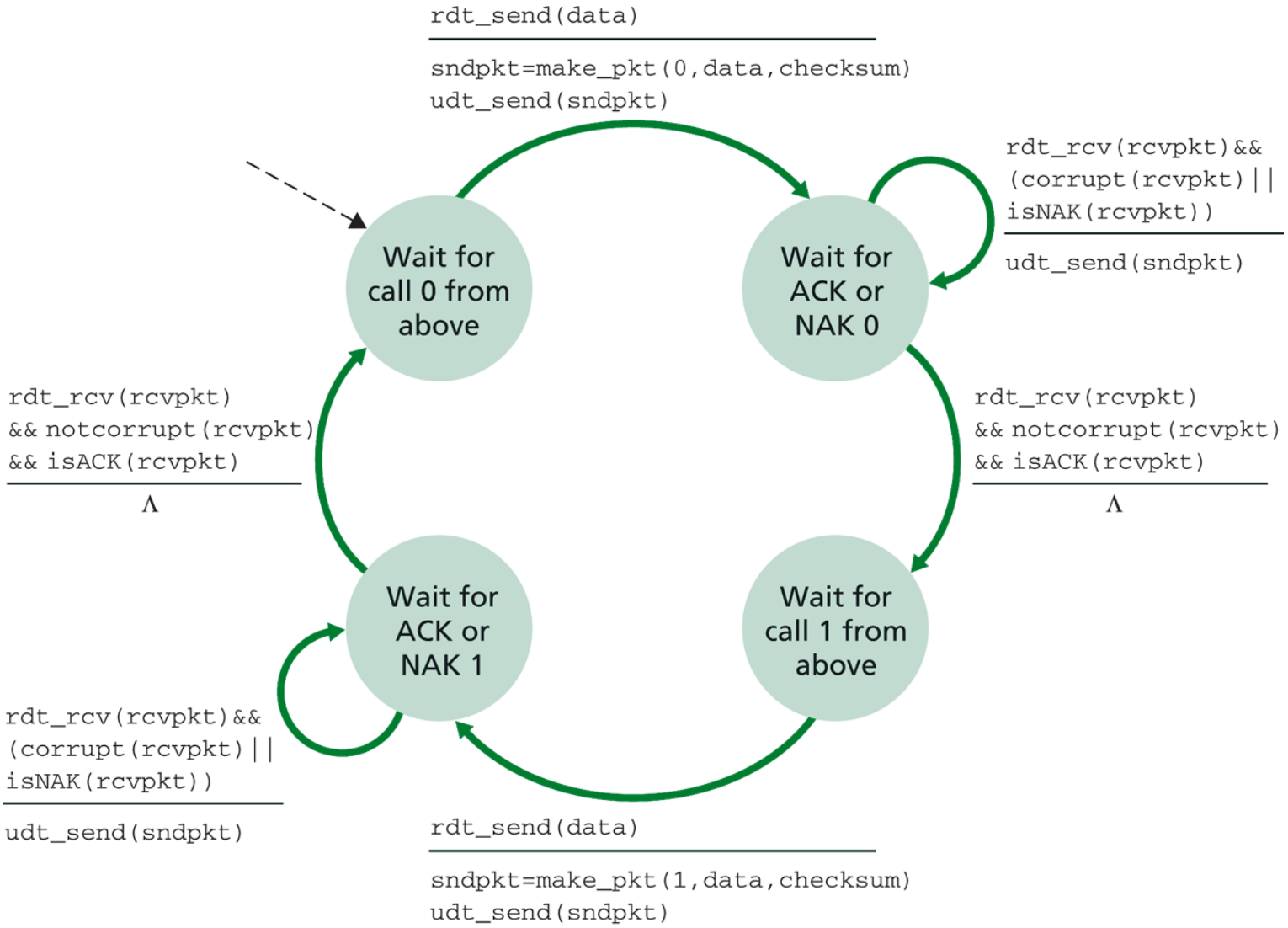
Handling duplicates:

- Sender adds sequence number to each pkt
- Sender retransmits current pkt if ACK/NAK garbled
- Receiver discards (doesn't deliver up) duplicate pkt

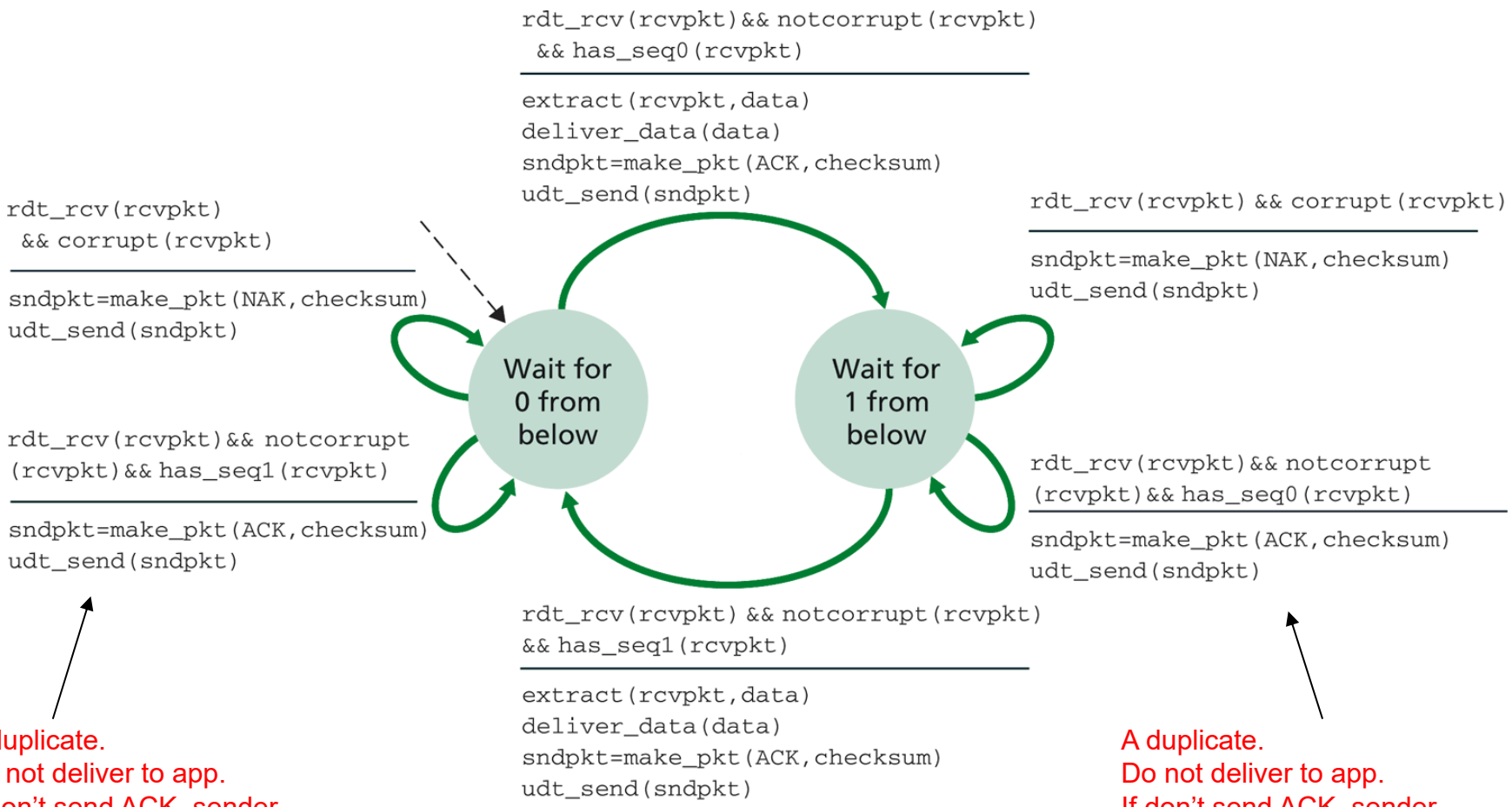
stop and wait

Sender sends one packet,
then waits for receiver response
before sending next packet

# rdt2.1: Sender, Handles Garbled ACK/NAKs



# rdt2.1: Receiver, Handles Garbled ACK/NAKs




A duplicate.
Do not deliver to app.
If don't send ACK, sender stuck in "Wait for ACK or NAK 1"

A duplicate.
Do not deliver to app.
If don't send ACK, sender stuck in "Wait for ACK or NAK 0"

# rdt2.1: 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 "current" pkt has 0 or 1 seq. #

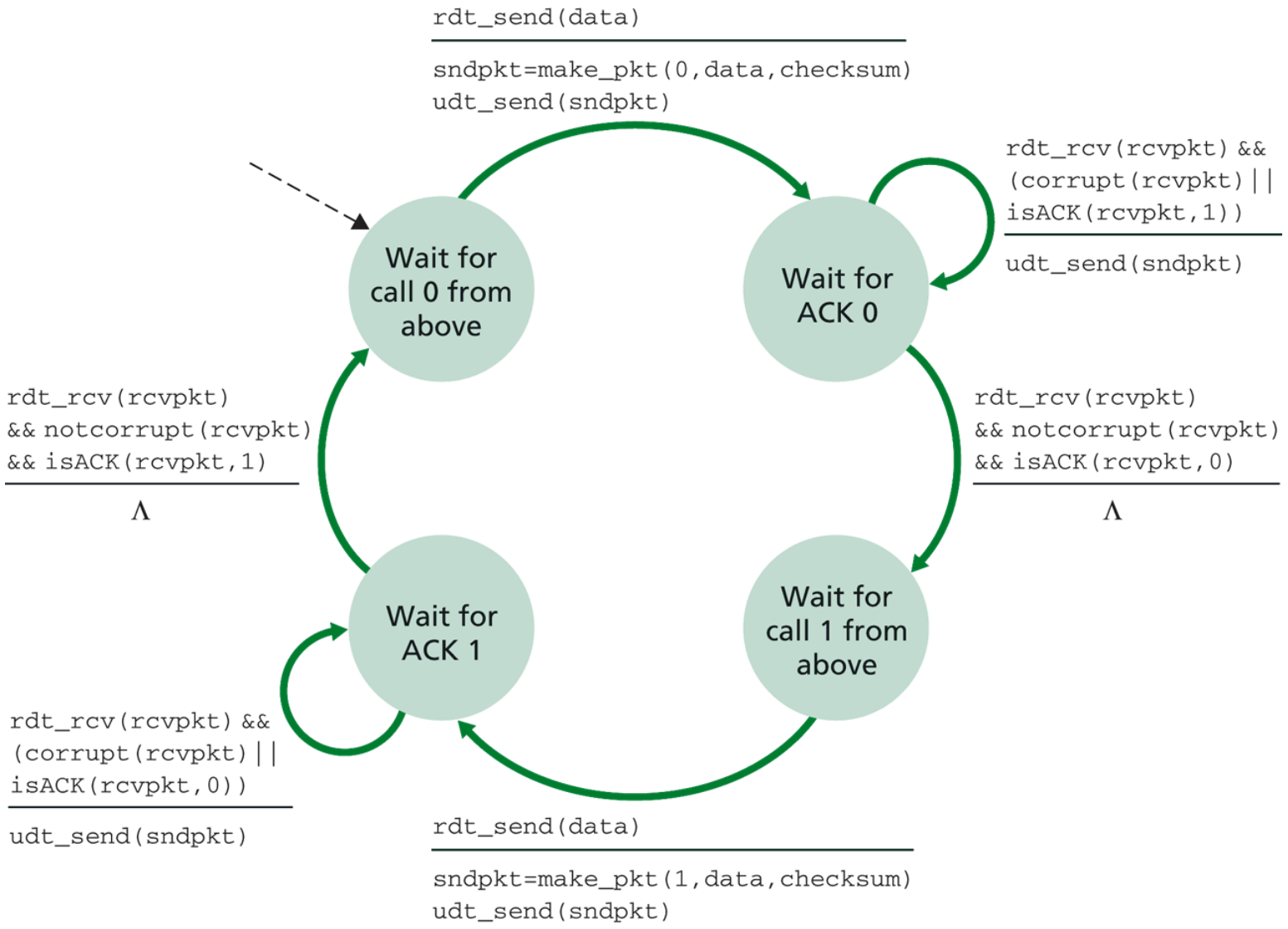
Receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected pkt seq #
- Note: receiver cannot know if its last ACK/NAK received OK at sender

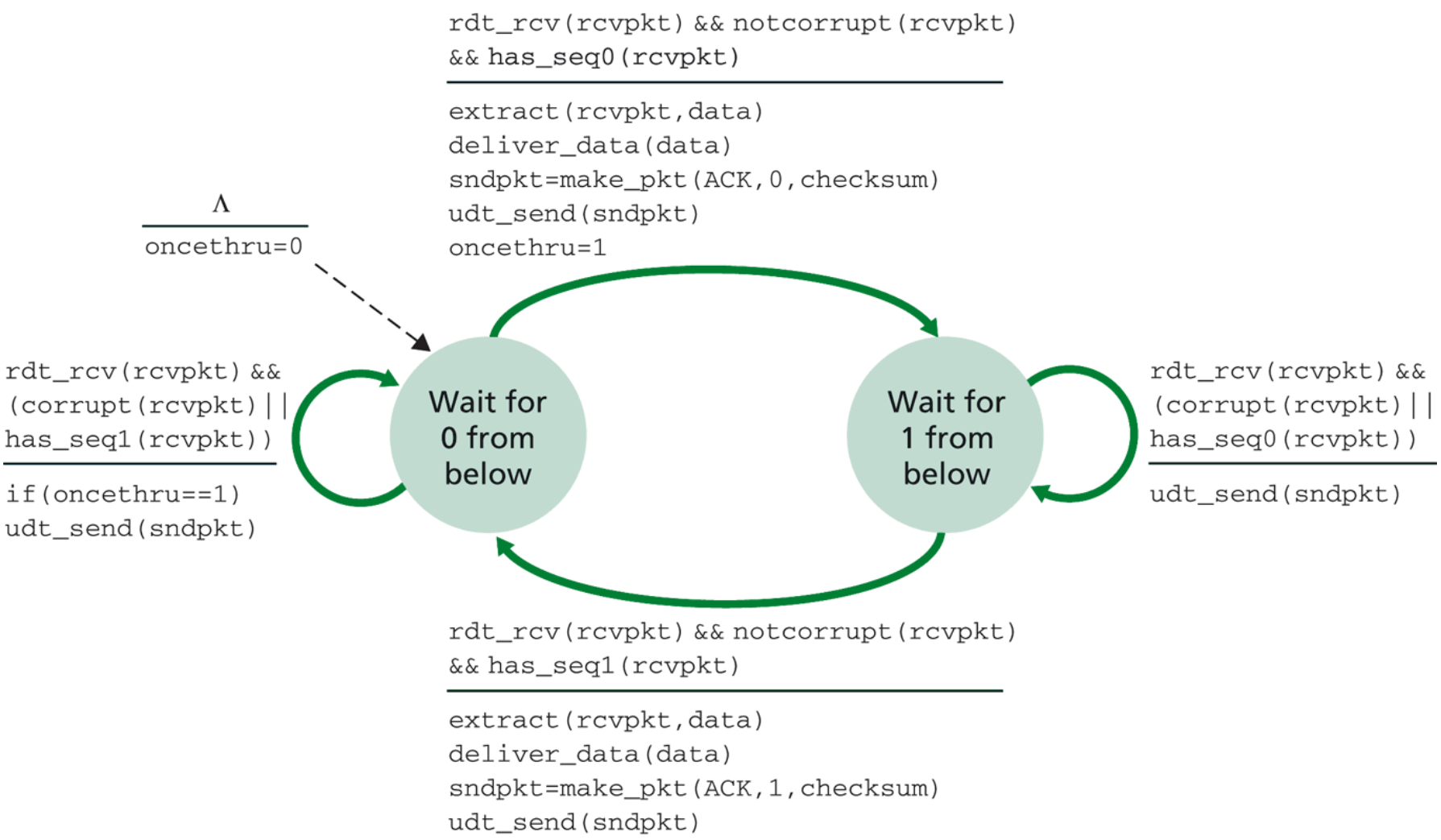
# rdt2.2: A NAK-free Protocol

- Same functionality as rdt2.1, but 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

# rdt2.2: Sender



# rdt2.2: Receiver



# rdt3.0: Channels With Errors and Loss

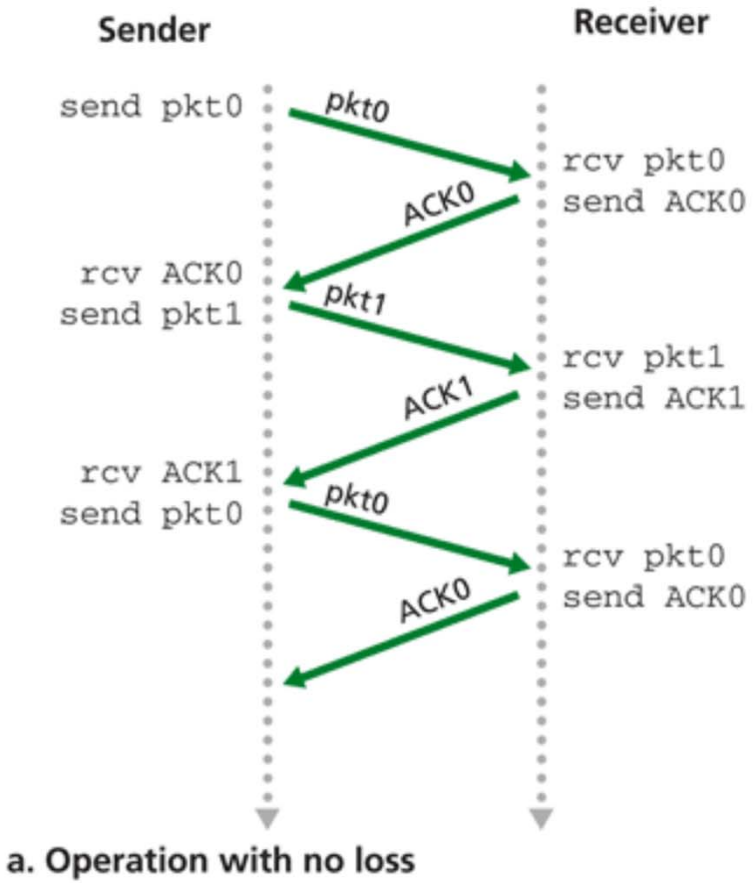
New assumption:

- Underlying channel can also lose packets (data or ACKs)
  - checksum, seq. #, ACKs, retransmissions will be of help, but not enough

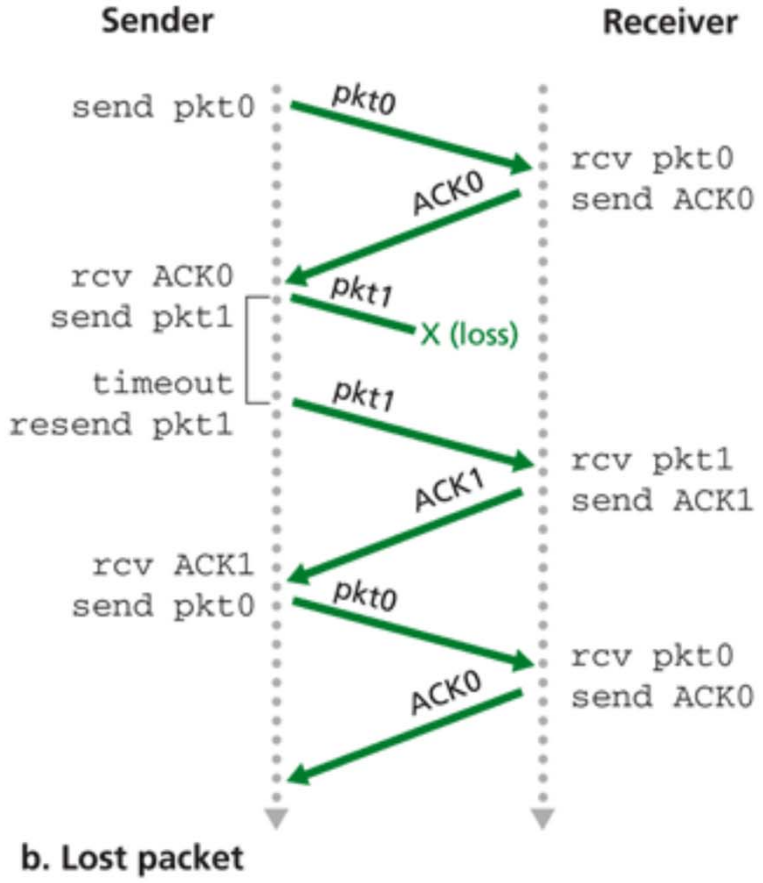
Approach:

- Sender waits "reasonable" amount of time for ACK
  - Requires countdown timer
- 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

# rdt3.0 In Action

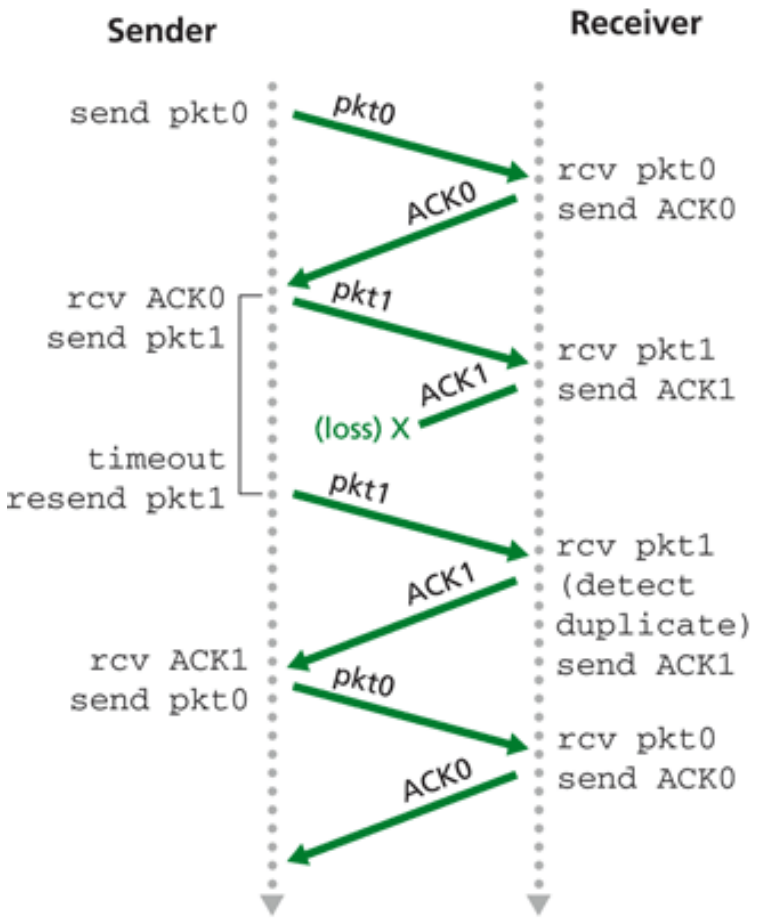



a. Operation with no loss

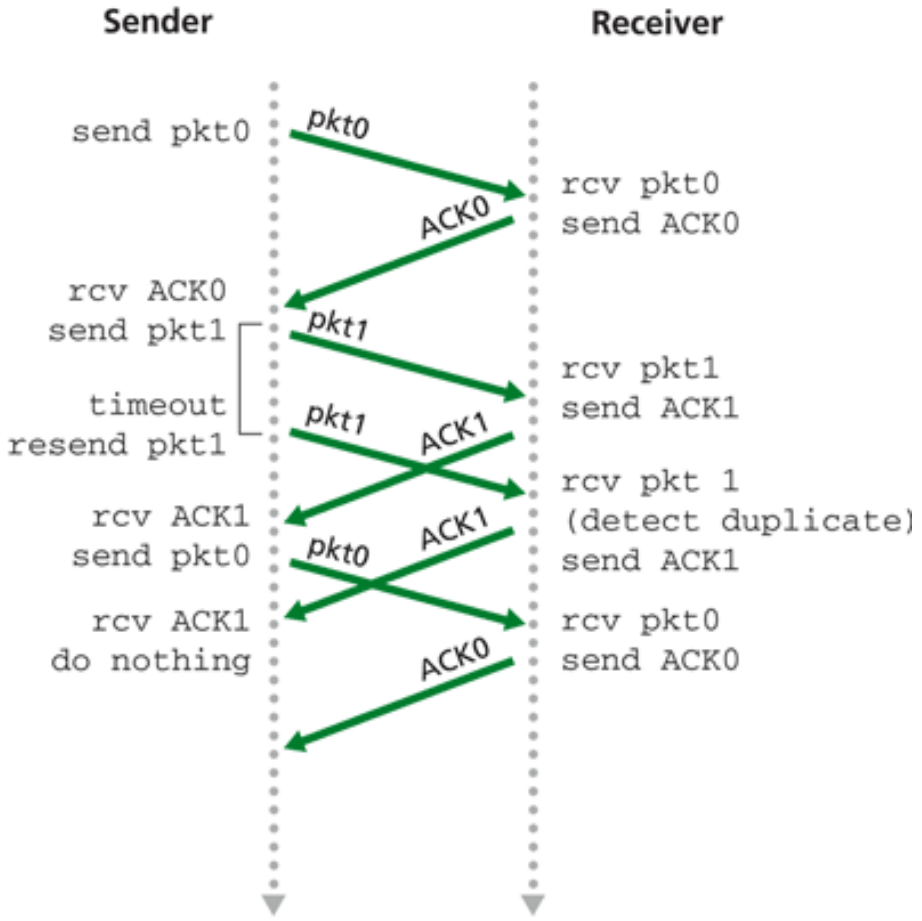



b. Lost packet

# rdt3.0 In Action

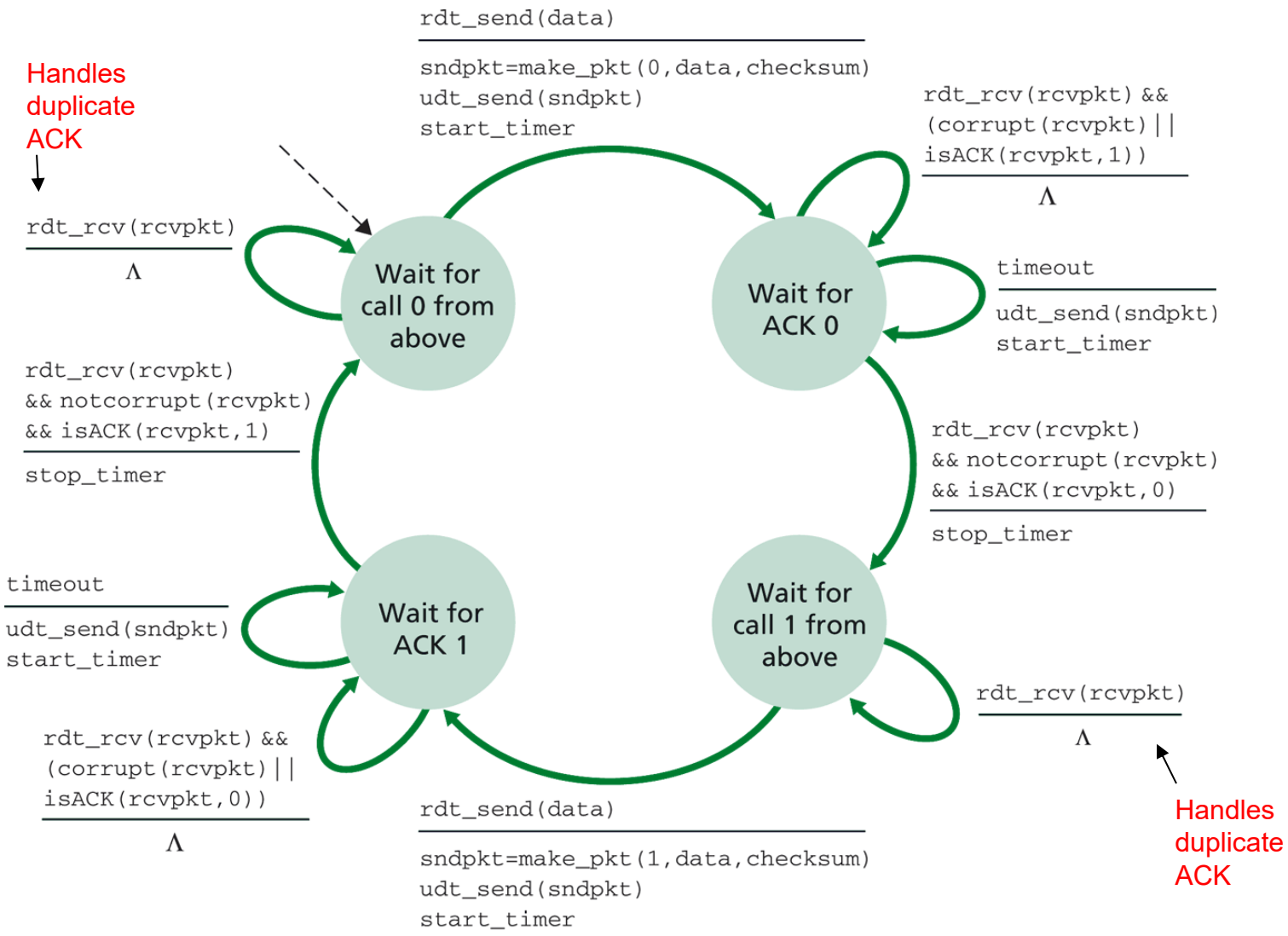



c. Lost ACK

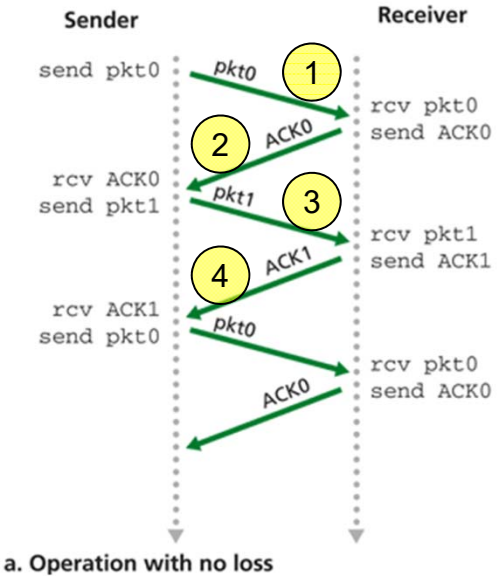



d. Premature timeout

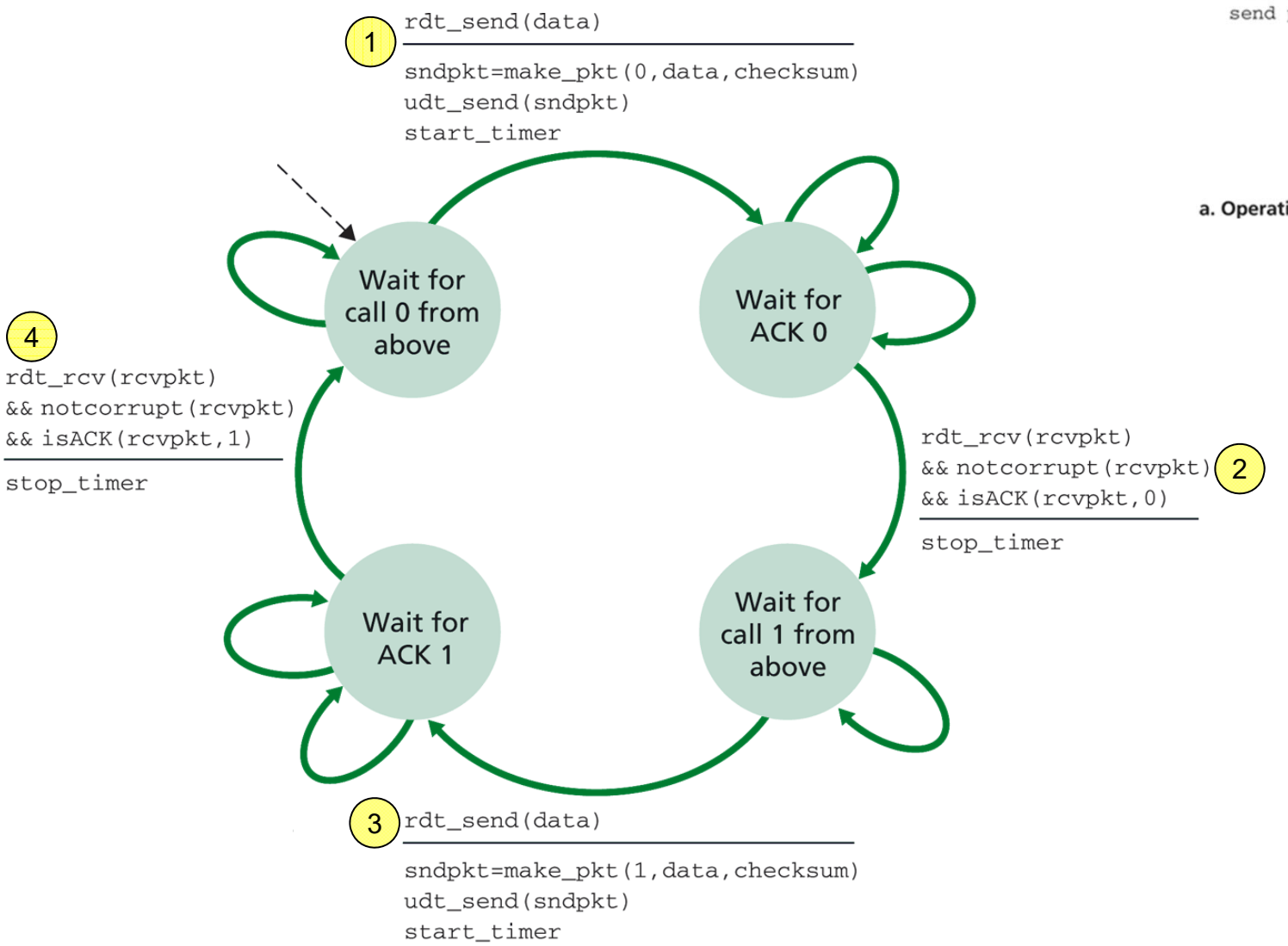
# rdt3.0 Sender



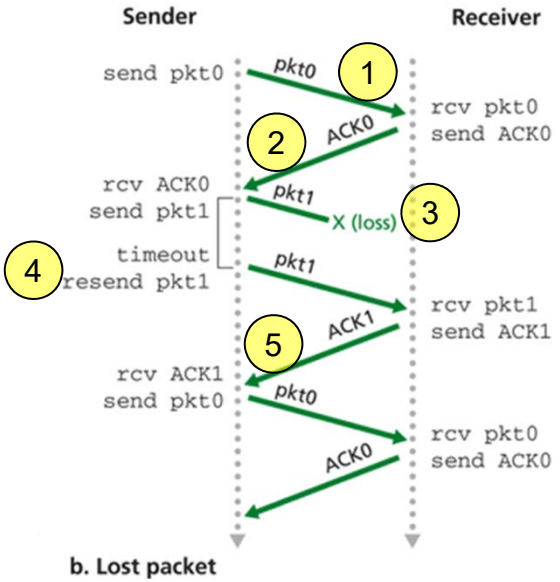
# rdt3.0 Sender

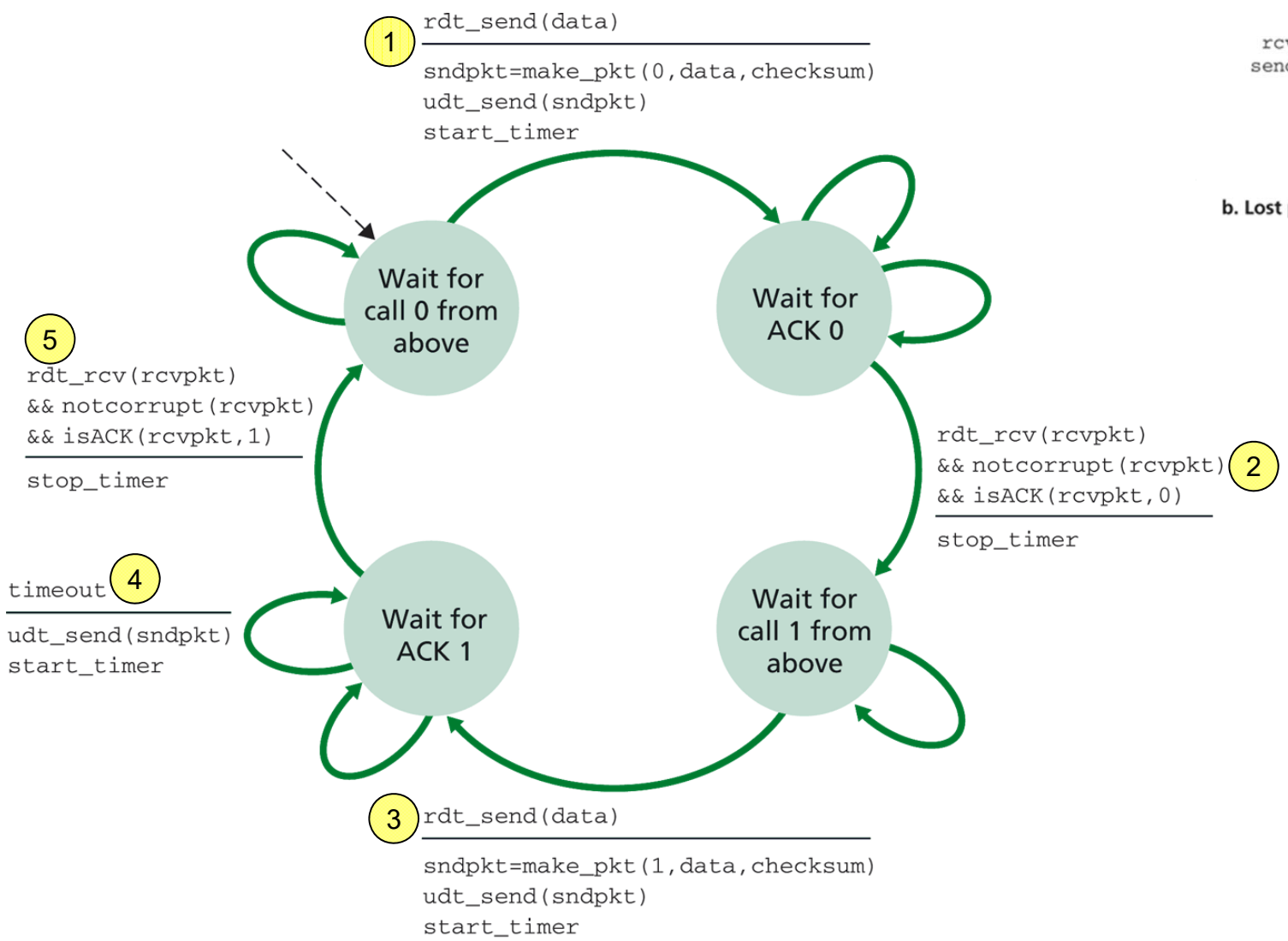



a. Operation with no loss

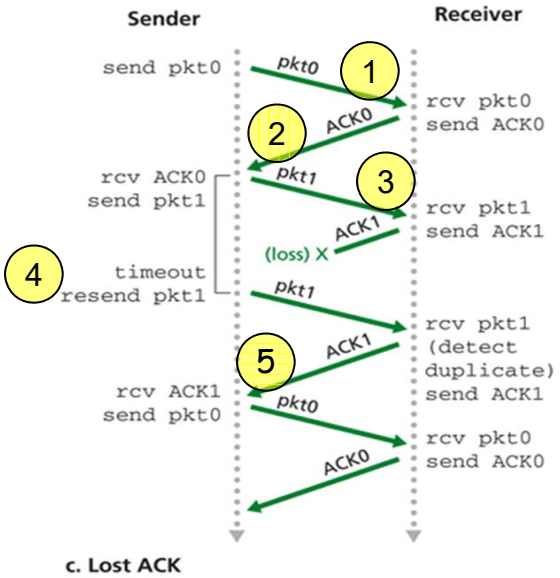


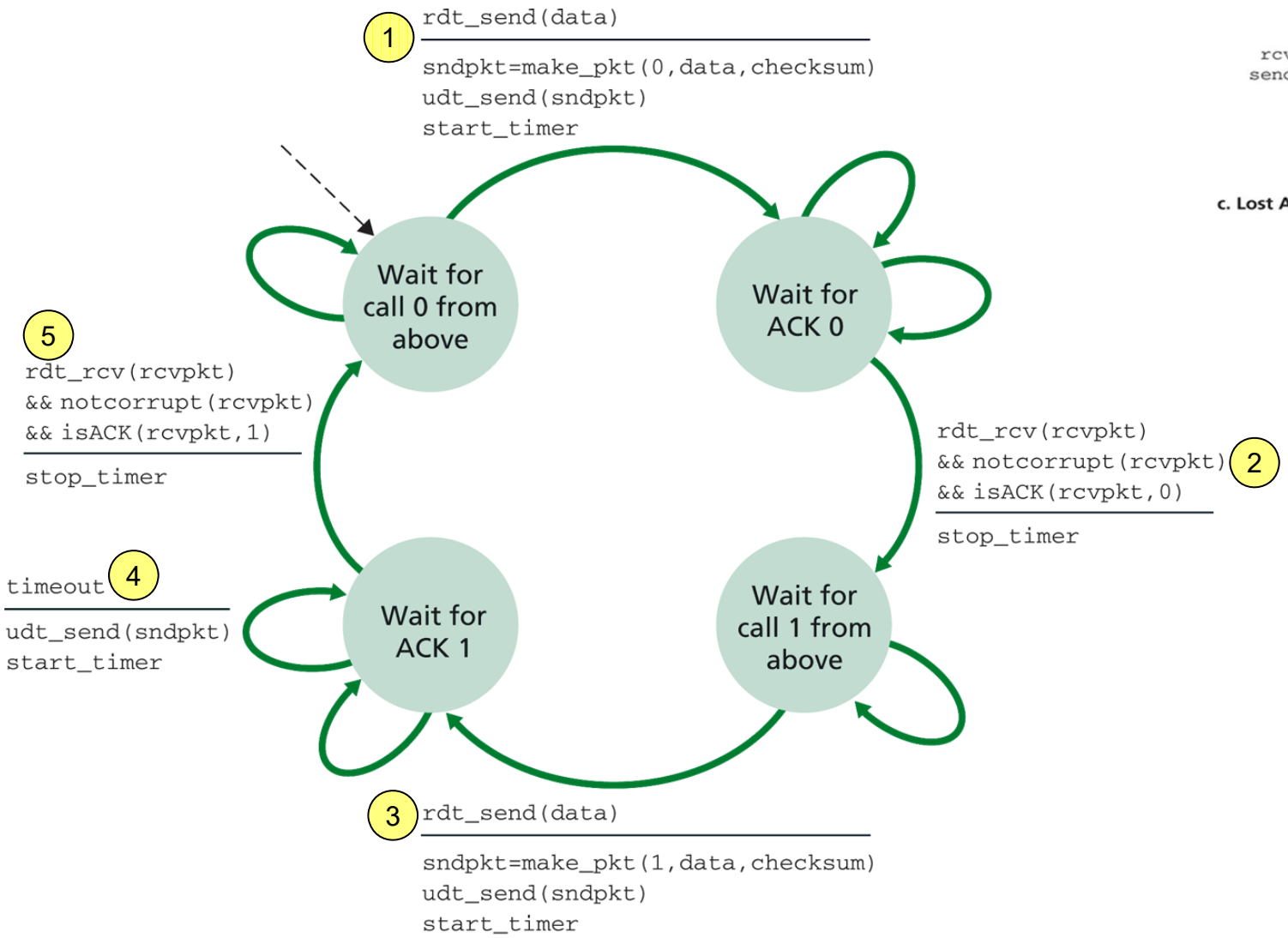
# rdt3.0 Sender



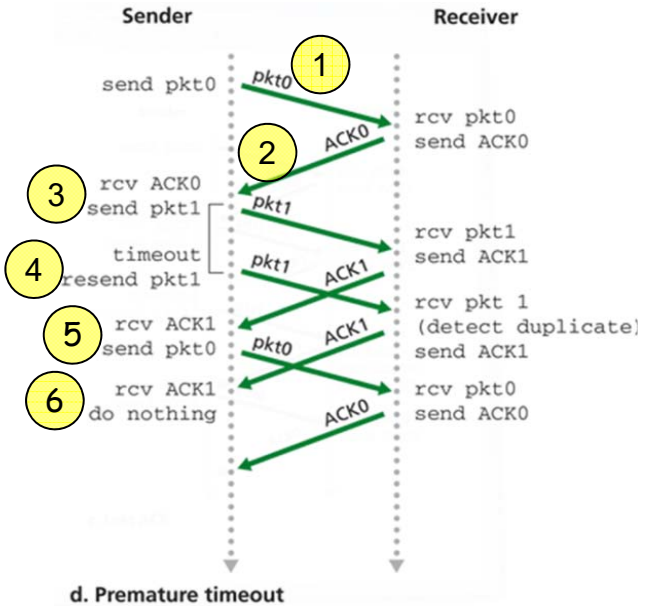



# rdt3.0 Sender



# rdt3.0 Sender




Handles duplicate ACK

6 `rdt_rcv(rcvpkt)` / Λ

1 `rdt_send(data)` / `sndpkt=make_pkt(0,data,checksum)` `udt_send(sndpkt)` `start_timer`

Wait for call 0 from above

Wait for ACK 0

2 `rdt_rcv(rcvpkt)` `&& notcorrupt(rcvpkt)` `&& isACK(rcvpkt,0)` / `stop_timer`

Wait for call 1 from above

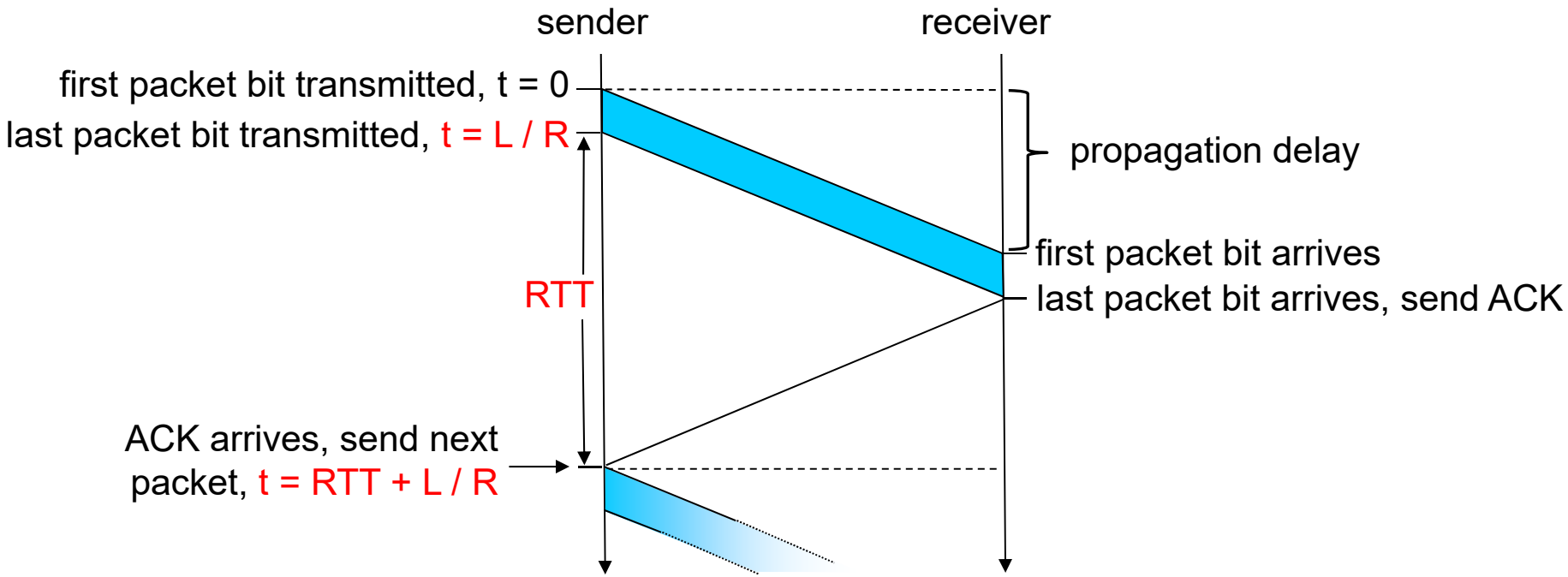
3 `rdt_send(data)` / `sndpkt=make_pkt(1,data,checksum)` `udt_send(sndpkt)` `start_timer`

Wait for ACK 1

4 `timeout` / `udt_send(sndpkt)` `start_timer`

5 `rdt_rcv(rcvpkt)` `&& notcorrupt(rcvpkt)` `&& isACK(rcvpkt,1)` / `stop_timer`

# rdt3.0: Stop-and-Wait Operation / Performance



# Performance of rdt3.0

- rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms end-to-end prop. delay, 1000 byte packet:

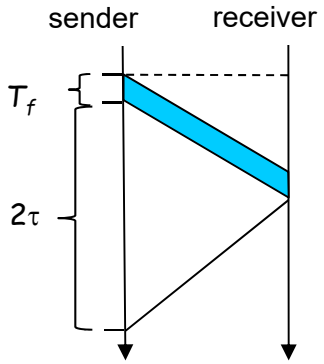
$$T_{transmit} = \frac{L\ (packet\ length\ in\ bits)}{R\ (transmission\ rate, bps)} = \frac{8000\,b/\,pkt}{10^9\,b/\sec} = 8\mu s/\,pkt$$

$$U = \frac{\frac{L}{R}}{\frac{L}{R} + RTT} = \frac{8x10^{-6}}{30.008x10^{-3}} = 0.00027$$

- $U_{sender}$: utilization → fraction of time sender busy sending
  - We want $U_{sender}$ to be closer to 1, not 0.00027!
- 1000 B pkt every 30 msec -> 267 kbps thruput over 1 Gbps link!
- Network protocol limits use of physical resources

# Efficiency of Stop-and-Wait

- What length of time should the timeout (T) timer be set to?
  - Must compensate for:
    - Propagation time one way ($\tau$)
    - Processing time at each station ($T_p$)
    - Time to transmit an acknowledgment ($T_{ack}$)
  - Normally timeout (T) value is > $2\tau + T_{ack} + 2T_p$

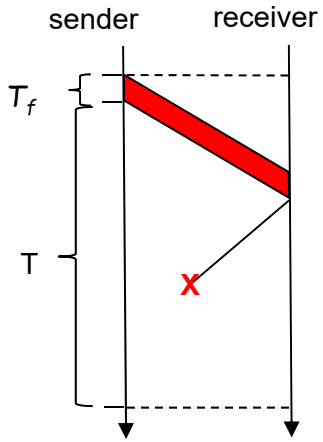



RTT

- Let's assume
  - processing time ($T_p$) is relatively negligible
  - acknowledgment frame time ($T_{ack}$) is small compared to data frame
  - Time to transmit one frame is $T_f$
- What is the efficiency of Stop-and-Wait in an errorless environment?

$$U = \frac{\frac{L}{R}}{\frac{L}{R} + RTT} = \frac{T_f}{T_f + 2\tau}$$

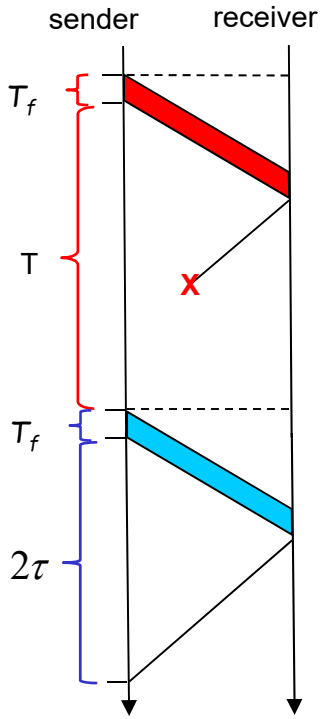
# Problems with Stop-and-Wait (cont.)




- What happens in a noisy environment (i.e., errors)?
  - Let *P* be the probability of an unsuccessful transmission of a data or acknowledgment frame
  - Assume that errors on different frames are independent
  - Every unsuccessful transmission wastes $T_f$ + *T* seconds, where *T* is the timeout period

- Q: What is the probability of i transmissions per frame?
  - A: It is the probability of
    - i-1 unsuccessful transmissions followed by
    - 1 successful transmission

# Problems with Stop-and-Wait (cont.)




- We also need to know the average number of transmissions per frame, $N_f$

$$N_f = \sum_{i=1}^{\infty} i a_i = \sum_{i=1}^{\infty} i P^{i-1}(1-P) = \frac{1}{1-P}$$

  - first $N_f - 1$ transmissions waste $(T + T_f)(N_f - 1) = \dfrac{(T+T_f)P}{(1-P)}$ seconds
  - last (successful) transmission takes $T_f + 2\tau$
  - total time, $T_t$ is the summation of the two times, so efficiency is

$$U = \frac{T_f}{\dfrac{(T+T_f)P}{(1-P)} + (T_f + 2\tau)}$$

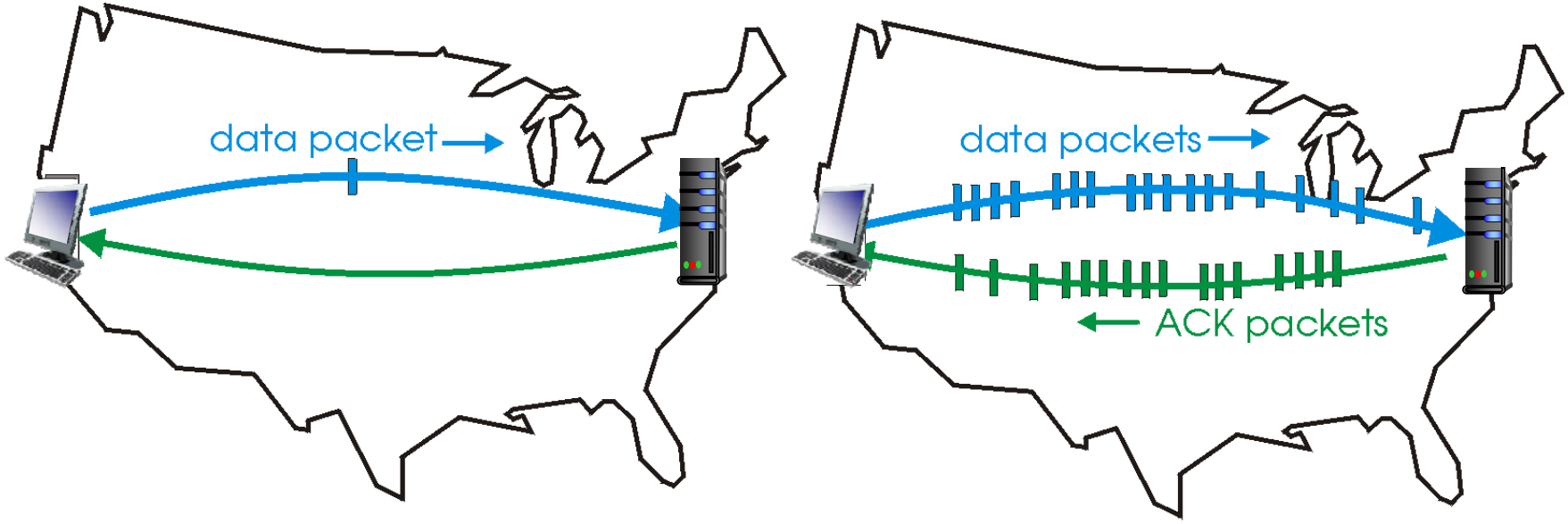
What if there are no errors (P=0)? $\longrightarrow$ $U = \dfrac{T_f}{T_f + 2\tau}$

# 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

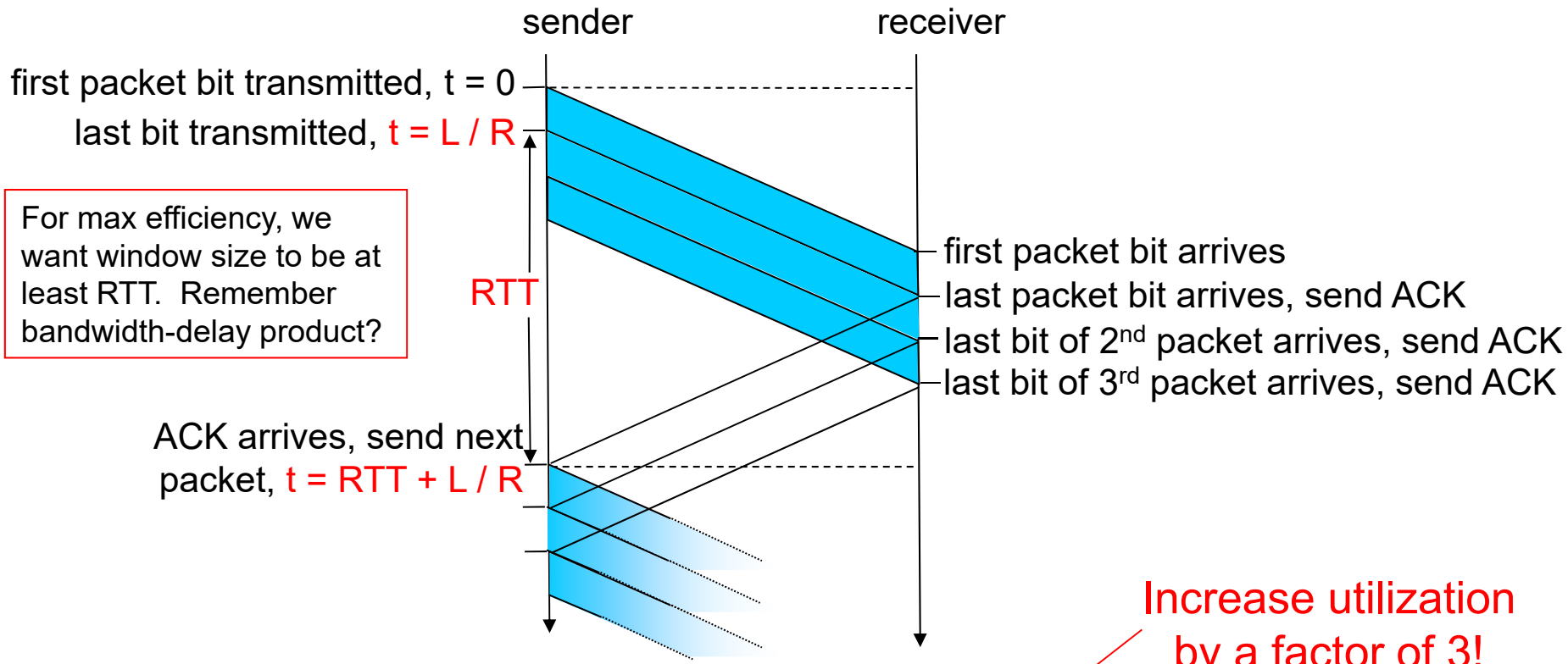
- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*




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

(b) a pipelined protocol in operation

# Pipelining: Increased Utilization




For max efficiency, we want window size to be at least RTT. Remember bandwidth-delay product?

$$U = \frac{3 * \frac{L}{R}}{\frac{L}{R} + RTT} = \frac{24 x 10^{-6}}{30.008 x 10^{-3}} = 0.0008$$

Increase utilization by a factor of 3!

$$U = \frac{3T_f}{T_f + 2\tau}$$

# Pipelined Protocols: Overview

Go-back-N:

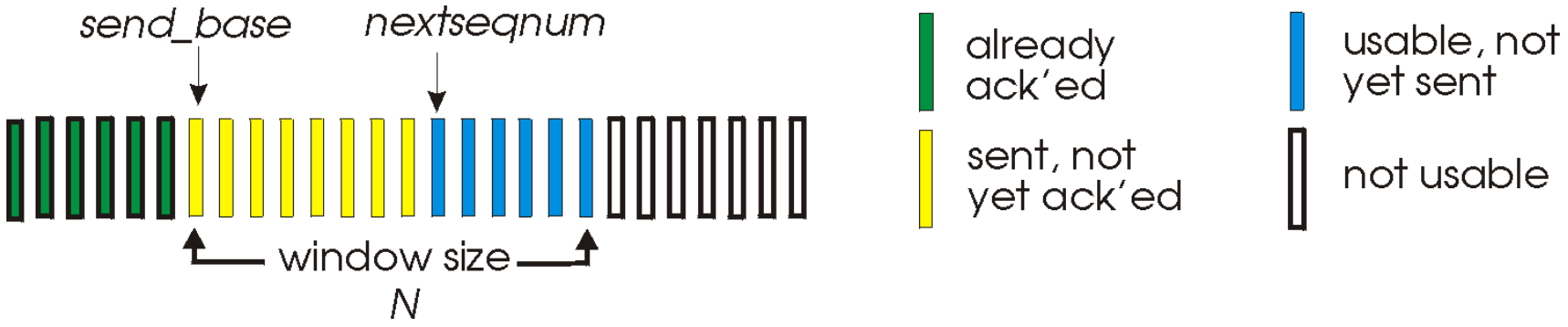
- Sender can have up to N unacked packets in pipeline
- Receiver sends *cumulative ack*
  - Doesn't ack packet if there's a gap (missing packet)
- Sender has timer for oldest unacked packet
  - When timer expires, retransmit *all* unacked packets

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 (Sliding Window Protocol) Sender

- k-bit seq # in pkt header → $2^k$ packets
- "Window" of up to N, consecutive unack'ed pkts allowed
  - With N = 1, Go-Back-N becomes Stop-and-Wait ARQ




- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
- Timer is used for in-flight packets
  - Really a timer for the oldest transmitted but not yet acked pkt (yellow packet)
- Upon *timeout(n)*: retransmit pkt n and all higher seq # pkts in window (retransmit all yellow packets)

# GBN: Sender Extended FSM


rdt_send(data)

```
if (nextseqnum < base+N) {
   sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum)
      start_timer
   nextseqnum++
   }
else
  refuse_data(data)
```

Λ

base=1
nextseqnum=1

Wait

timeout

start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

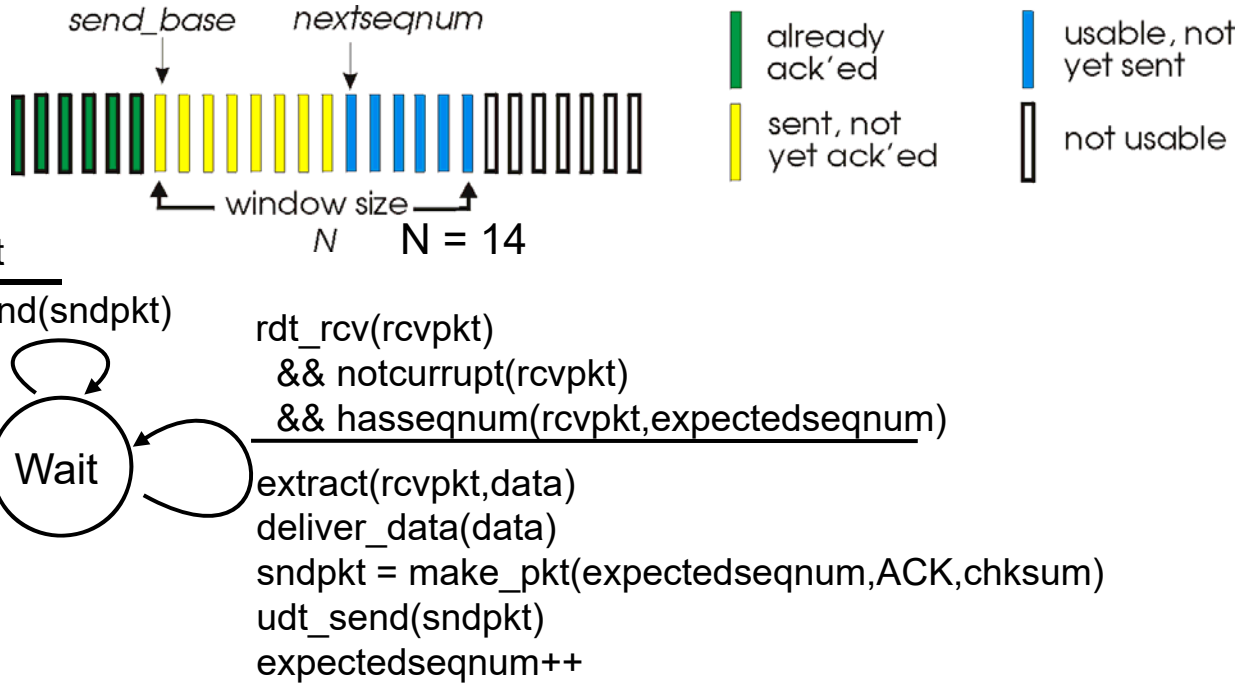
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

Λ

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
   stop_timer
  else
   start_timer
```

# GBN: Receiver Extended FSM




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

0 1 2 3 4 5 6 7 8

*sender* *receiver*

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

X *loss*

rcv pkt0, send ack0

rcv pkt1, send ack1

rcv pkt3,discard,(re)send ack1

rcv ack0, send pkt4

rcv ack1, send pkt5

rcv pkt4,discard,(re)send ack1

rcv pkt5,discard,(re)send ack1

ignore duplicate ACK

*pkt 2 timeout*

send pkt2

send pkt3

send pkt4

send pkt5

rcv pkt2, deliver, send ack2

rcv pkt3, deliver, send ack3

rcv pkt4, deliver, send ack4

rcv pkt5, deliver, send ack5

http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

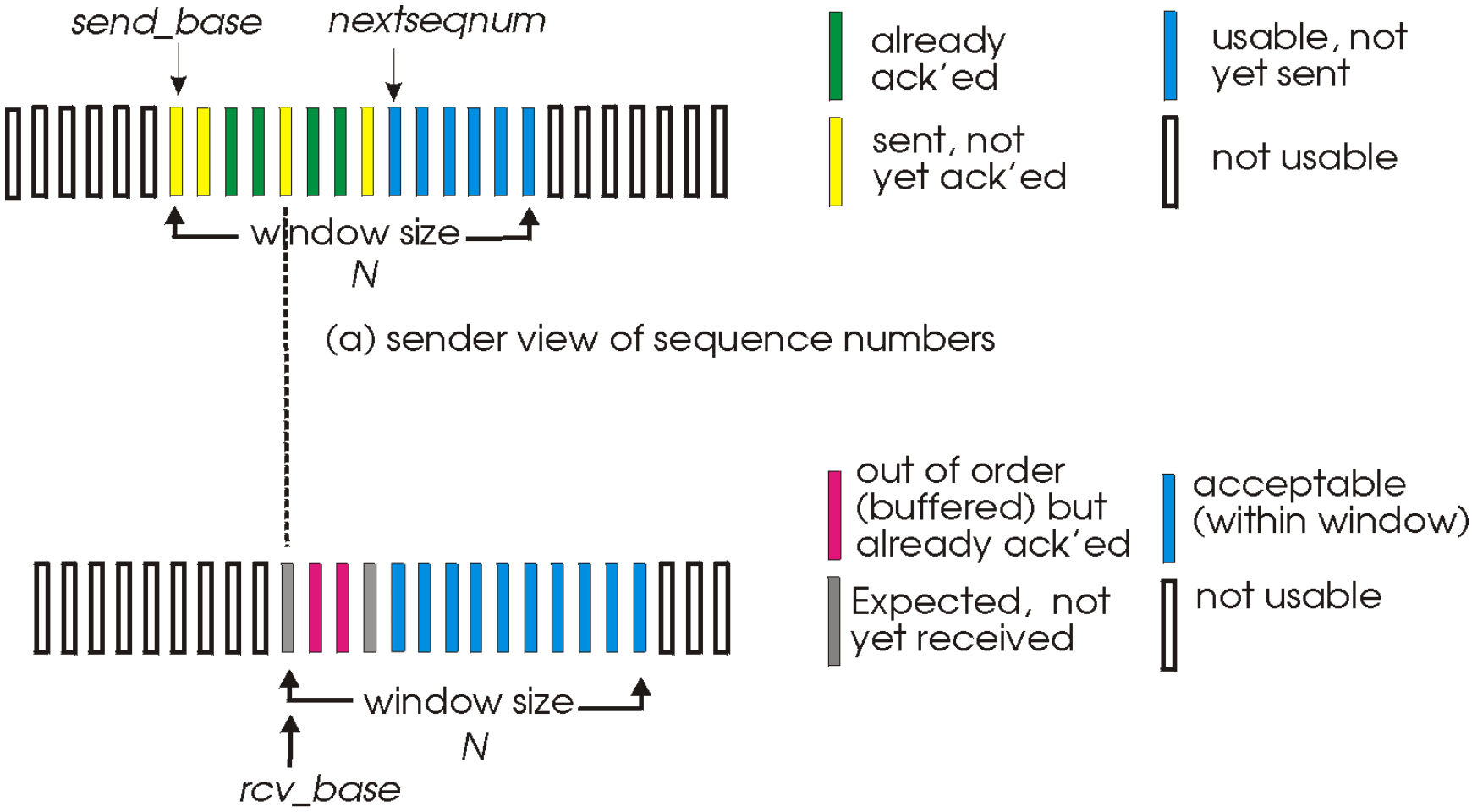
# Selective Repeat

- Receiver individually acknowledges all correctly received pkts
  - Buffers pkts for eventual in-order delivery to upper layer

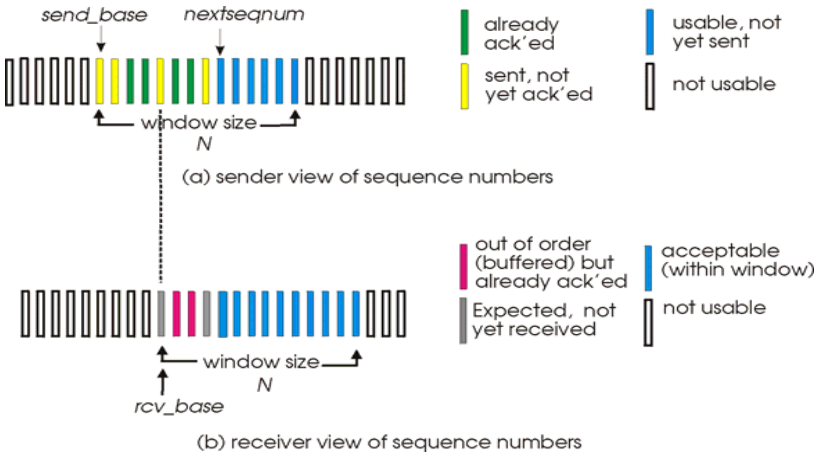
- Sender only resends pkts for which ACK not received
  - Sender has a timer for each unACKed pkt

- Sender window
  - N (window size) consecutive seq #'s
  - Again limits seq #s of sent, unACKed pkts

# Selective Repeat: Sender, Receiver Windows



# Selective Repeat




## Sender

data from above :

- if next available seq # is in window, send pkt

ACK(n) in [sendbase,sendbase+N-1]:

- mark pkt n as received
- if n is the smallest unACKed pkt, advance window base to next unACKed seq #

timeout(n):

- resend only pkt n, restart timer

## Receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- if out-of-order: buffer
- if in-order: deliver to app
  - also deliver buffered, in-order pkts
  - advance window to next not-yet-received pkt

pkt n in [rcvbase-N,rcvbase-1]

- ACK(n) again
- Covers the lost ACK case

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

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

record ack3 arrived

*pkt 2 timeout*

send pkt2

record ack4 arrived

record ack5 arrived

*receiver*

rcv pkt0, send ack0

rcv pkt1, send ack1

X *loss*

rcv pkt3, buffer, send ack3

rcv pkt4, buffer, send ack4

rcv pkt5, buffer, send ack5

rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

*Q: what happens when ack2 arrives?*

http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

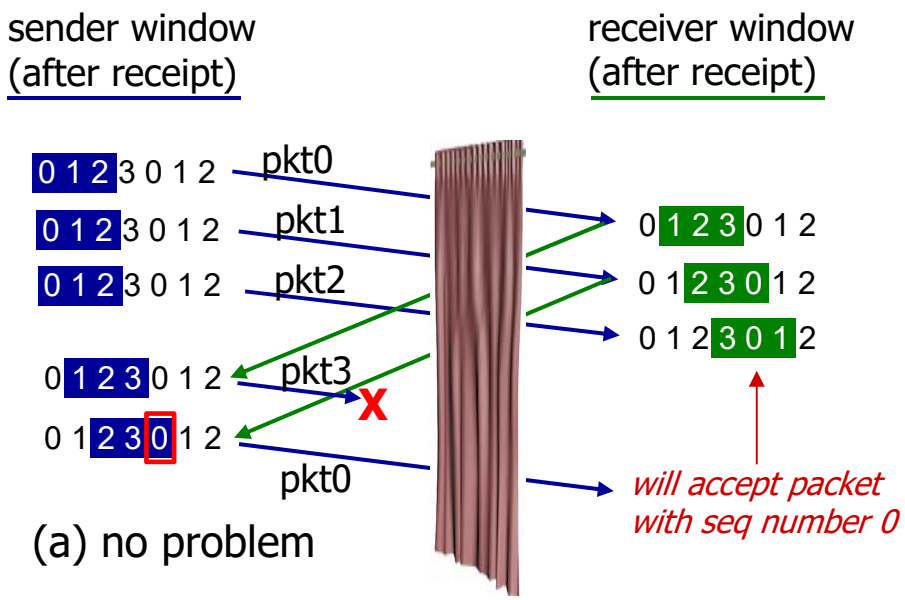
# Selective Repeat: Dilemma

Example:

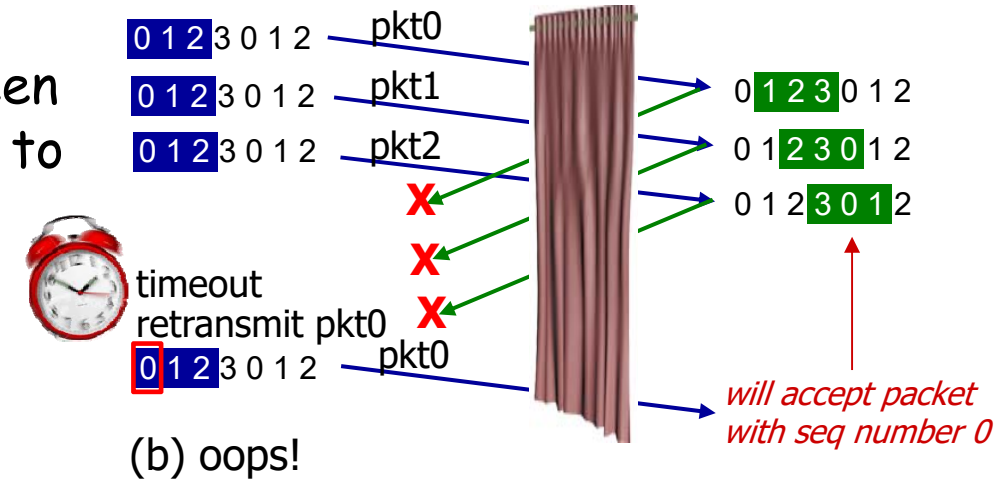
- seq #'s: 0, 1, 2, 3
- Window size = 3

- Receiver sees no difference in two scenarios!
- Duplicate data accepted as new data in (b)

- Q: what relationship between seq # size and window size to avoid problem in (b)?
  - Seq # > 2*window




*receiver can't see sender side.*
*receiver behavior identical in both cases!*
*something's (very) wrong!*

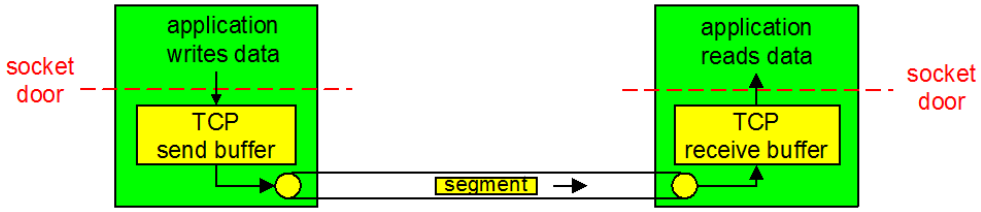


# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

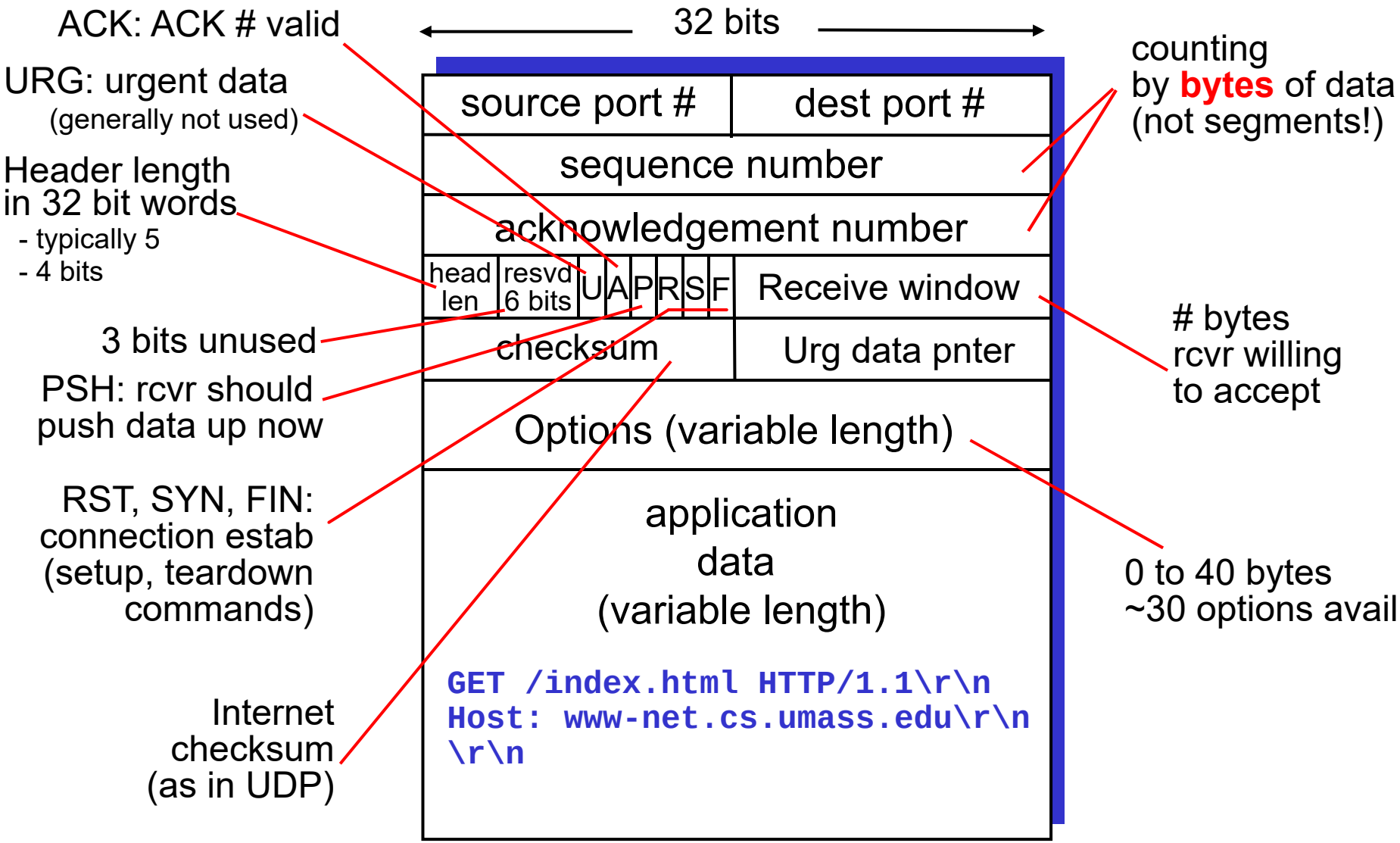
# TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point:
  - One sender, one receiver
- Reliable, in-order byte stream
- Pipelined (uses windowing):
  - TCP congestion and flow control set window size
- Send & receive buffers




- Full duplex data:
  - Bi-directional data flow in same connection
  - MSS: maximum segment size (data only)
- Connection-oriented:
  - Handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- Flow controlled:
  - Sender will not overwhelm receiver

# TCP Segment Structure



# TCP Seq. #'s and ACKs

Seq. #'s:

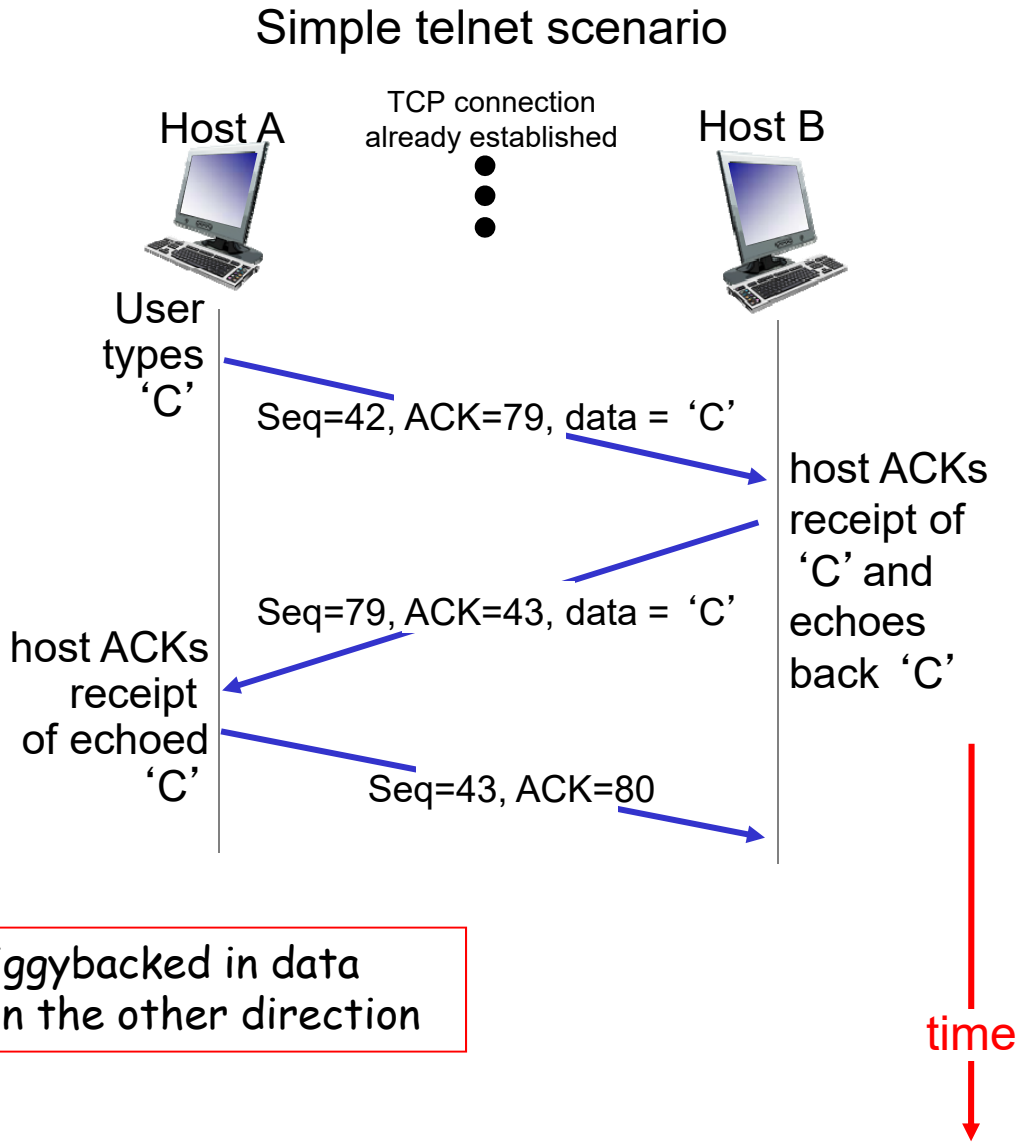
- Number of first byte in segment's data

ACKs:

- Seq # of next byte expected from other side
- Cumulative ACK

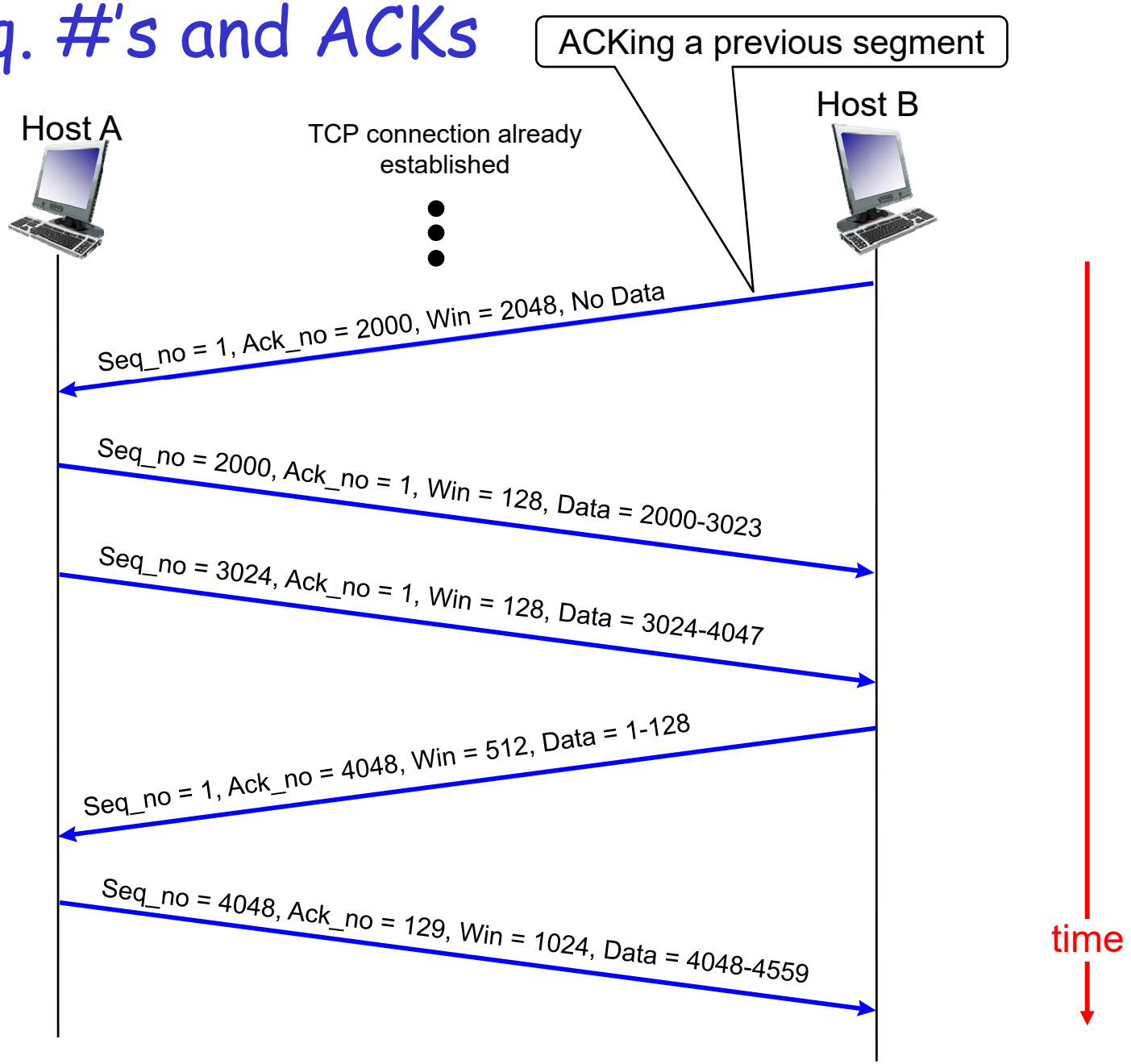
Q: How does receiver handle out-of-order segments?

- A: TCP spec doesn't say - up to implementer




ACKs piggybacked in data segments in the other direction

# TCP Seq. #'s and ACKs



# TCP Retransmission

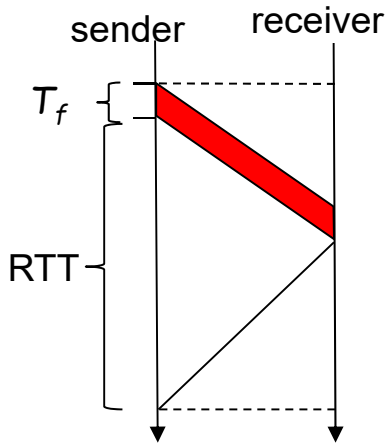
- TCP calculates round trip time (RTT) for a segment and its ack
- From the RTT, TCP can guess how long it should wait before timing out on the next segment
- When a segment remains unacknowledged for a period of time, TCP assumes it is lost and retransmits it
  - Timer for retransmitted segments is doubled for each retransmission of that segment with a maximum timeout of 240 seconds
- What happens if retransmissions of a segment continue to fail?
  - **TcpMaxDataRetransmissions** registry value indicates the maximum number of retransmissions that can be sent on an existing connection
  - Default is 3

```
Registry location:
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters
```

# TCP RTT and 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 end of segment transmission until ACK receipt
  - Ignore retransmissions
- SampleRTT will vary over time
  - We want an estimated RTT "smoother" to avoid short-term spikes
  - Average several recent measurements, not just current SampleRTT

# TCP RTT and Timeout

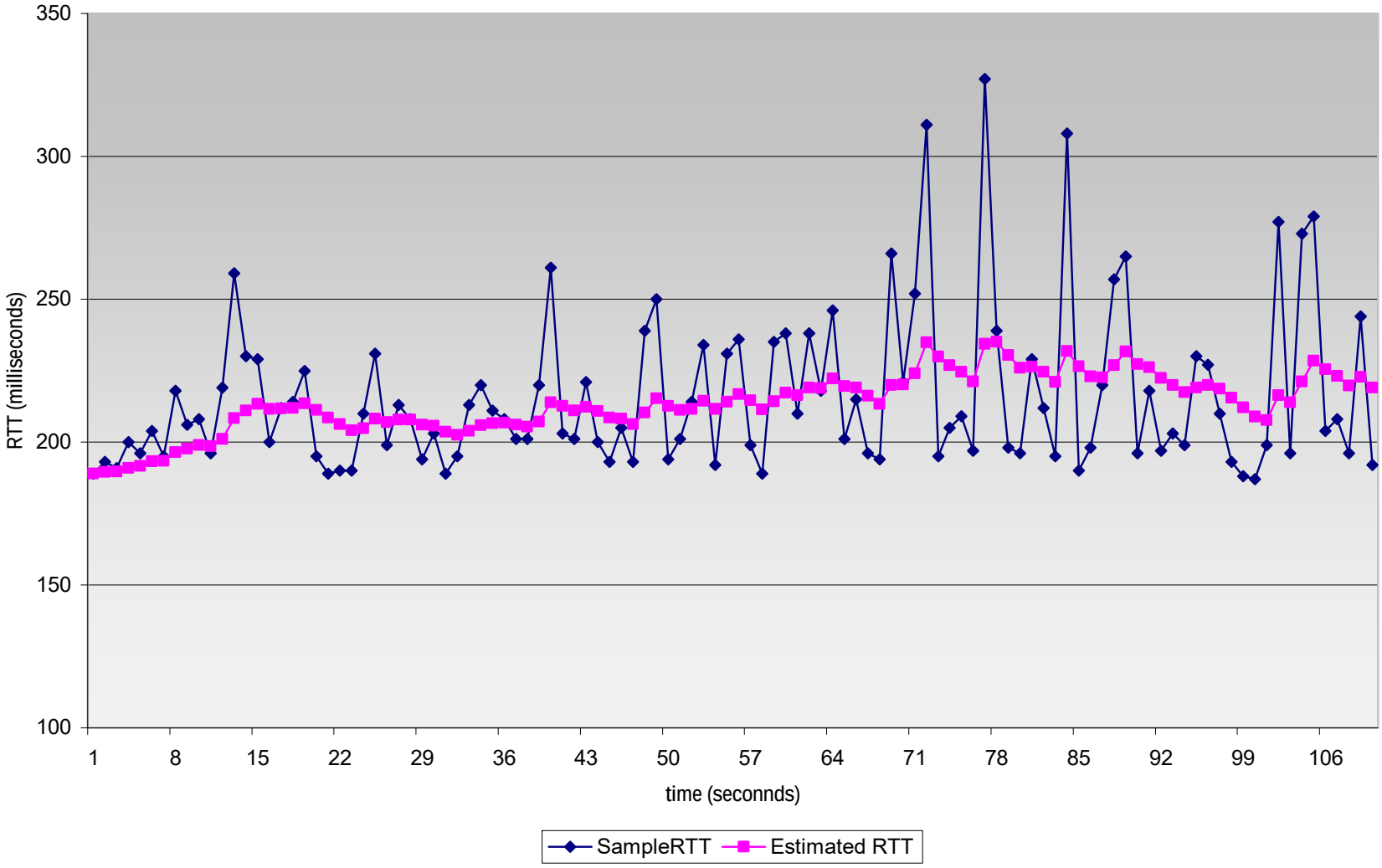
- Very first retransmission timeout value (1st segment)
  - **`TimeoutInterval`** set to 3 sec
  - Can be changed in Windows registry: **`TcpInitialRtt`**
- Subsequent timeout values are calculated based on sampled RTTs
- Typical value: $\alpha = 0.125$

**Registry location:**
**HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters**

**EstimatedRTT = (1 - $\alpha$)*EstimatedRTT + $\alpha$*SampleRTT**
**EstimatedRTT = (0.875)*EstimatedRTT + 0.125*SampleRTT**

- **`EstimatedRTT = SampleRTT`** for first calculation (2nd segment)
- "Exponential weighted moving average"
  - Influence of past sample decreases exponentially fast

# Example RTT Estimation



# TCP RTT and Timeout

Setting the timeout

- **EstimatedRTT** plus "safety margin"
  - Large variation in **EstimatedRTT** → larger safety margin

- 1st estimate of how much **SampleRTT** deviates from **EstimatedRTT:**

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

**DevRTT = 0.75 * DevRTT + 0.25 * |SampleRTT-EstimatedRTT|**

typically, β = 0.25

- **DevRTT = [SampleRTT / 2]** for first calculation (2nd segment)
- Then set timeout interval:

**TimeoutInterval = EstimatedRTT + 4*DevRTT**

Estimated RTT

"Safety margin"

# TCP RTT and TOI Example

First TimeoutInterval ($TOI_1$) = 3

For first calculation:

$EstimateRTT_2$ ($ERTT_2$) = $SampleRTT_1$ ($SRTT_1$)

$DevRTT_2 = [SRTT_1 / 2]$

$TOI_2 = ERTT_2 + 4*DevRTT_2$

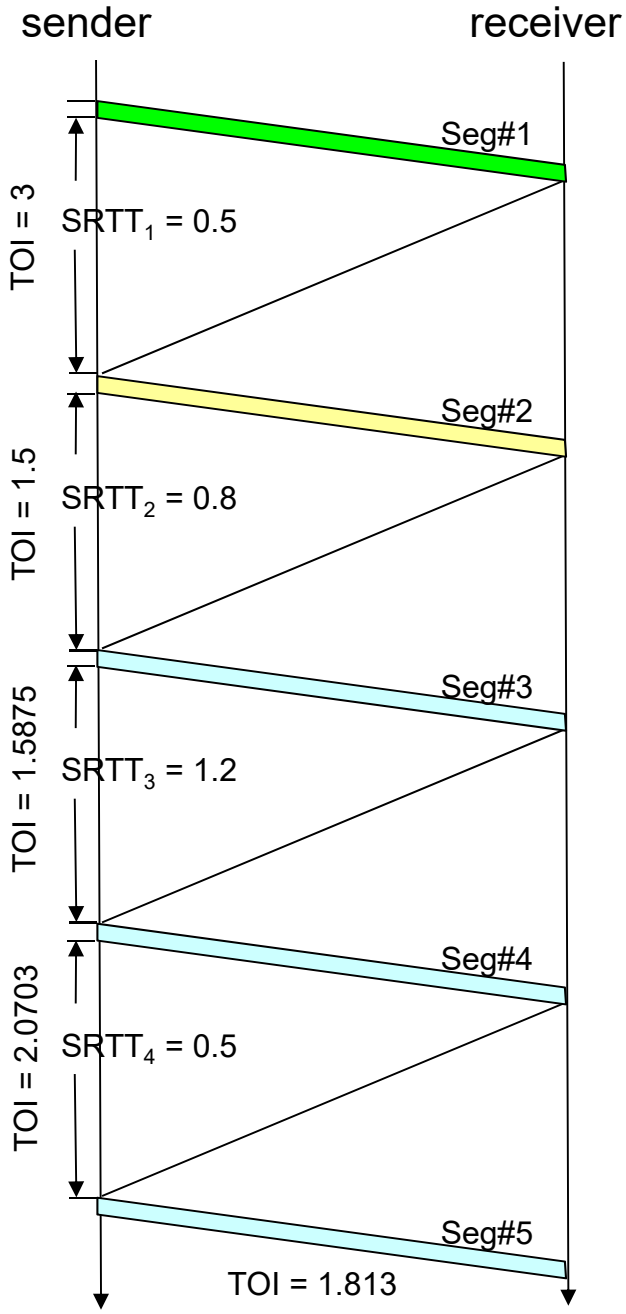
For all subsequent calculations (compute in this order):

$DevRTT_n = 0.75*DevRTT_{n-1} + 0.25*|SRTT_{n-1}-ERTT_{n-1}|$

$ERTT_n = 0.875*ERTT_{n-1} + 0.125*SRTT_{n-1}$

$TOI_n = ERTT_n + 4*DevRTT_n$

| Seg# | **DevRTT** Calculated | **ERTT** Calculated | **TOI** Calculated | **SRTT** Measured |
|---|---|---|---|---|
| 1 | | | 3 | 0.5 |
| 2 | 0.25 | 0.5 | 1.5 | 0.8 |
| 3 | 0.2625 | 0.5375 | 1.5875 | 1.2 |
| 4 | 0.3625 | 0.6203 | 2.0703 | 0.5 |
| 5 | 0.30195 | 0.6053 | 1.813 | |




SRTT times are made up

# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# TCP Reliable Data Transfer

- TCP creates reliable data transfer service on top of IP's unreliable service
  - Pipelined segments
  - Cumulative acks
  - Single retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Three duplicate acks
- Initially consider simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control

# TCP Sender Events:

Data rcvd from app:

- Create TCP segment with seq #
  - Seq # 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 rcvd:

- If ack acknowledges previously unacked segments
  - Update what is known to be acked
  - Start timer if there are outstanding segments

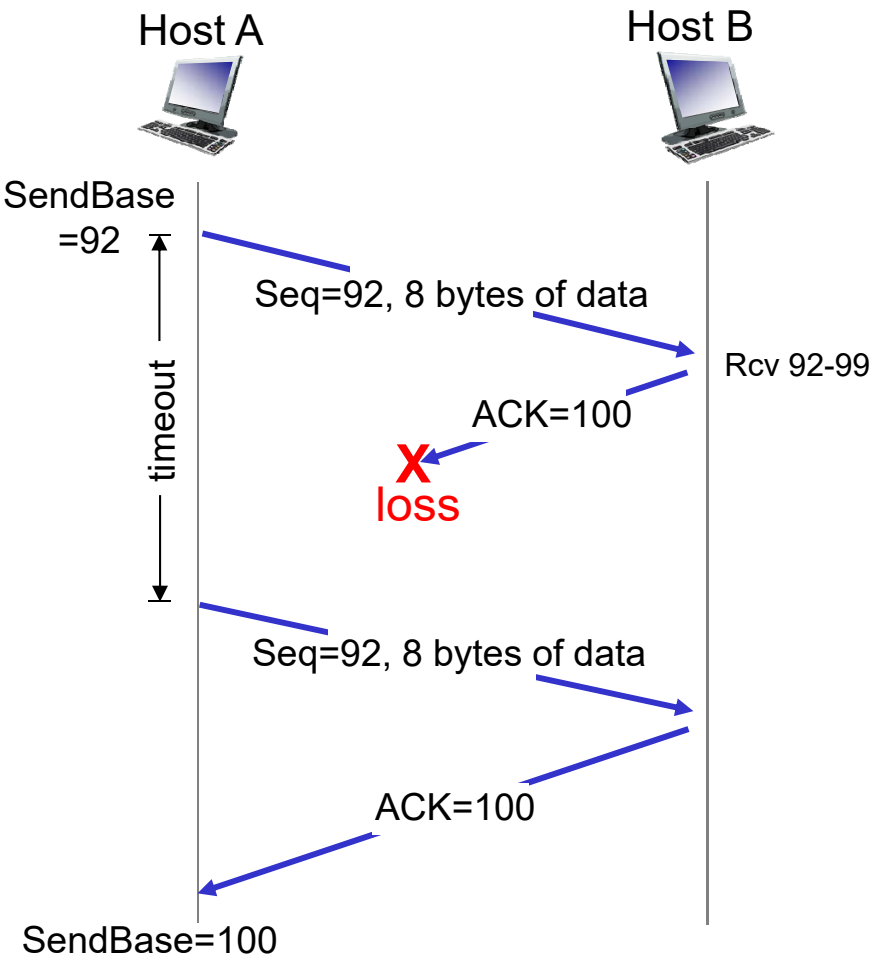
```
NextSeqNum = InitialSeqNum
SendBase   = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
          smallest sequence number
      start timer

  event: ACK received, with ACK value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
              start timer
          }

      }  /* end of loop forever */
```

# TCP Sender (simplified)

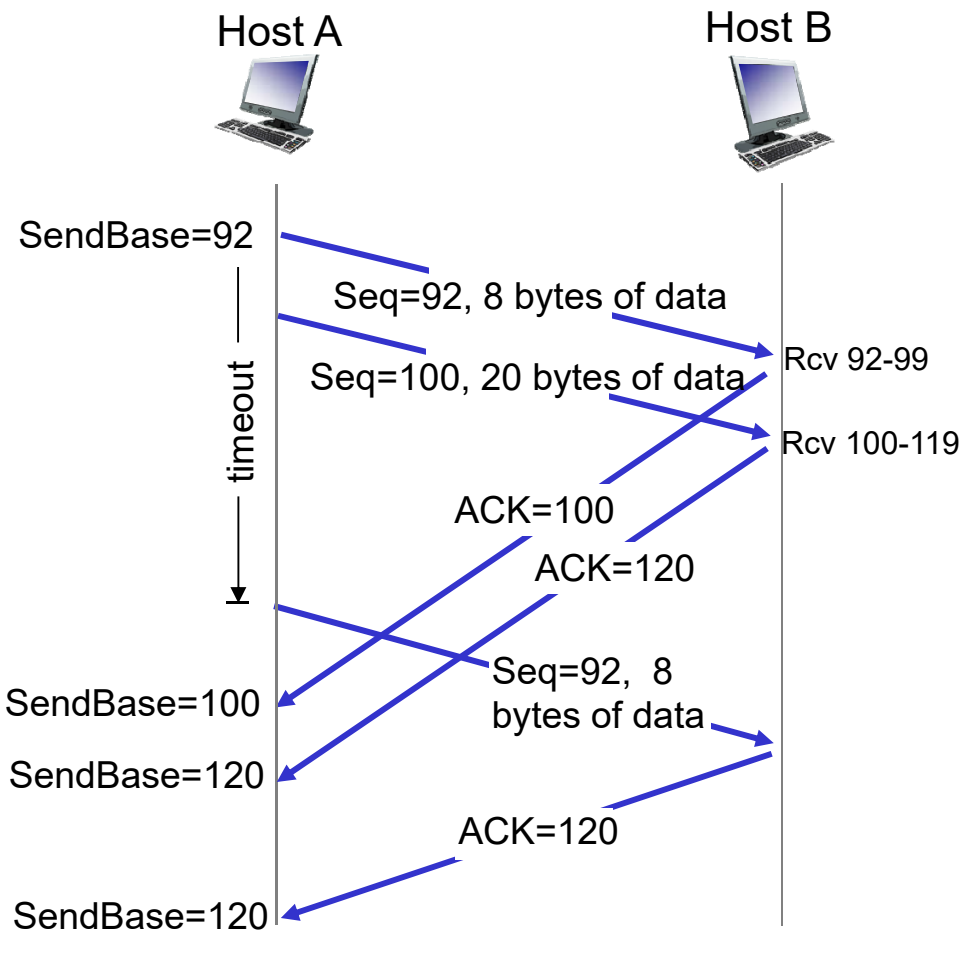
Comment:

SendBase-1: last cumulatively ack'ed byte

# TCP: Retransmission Scenarios

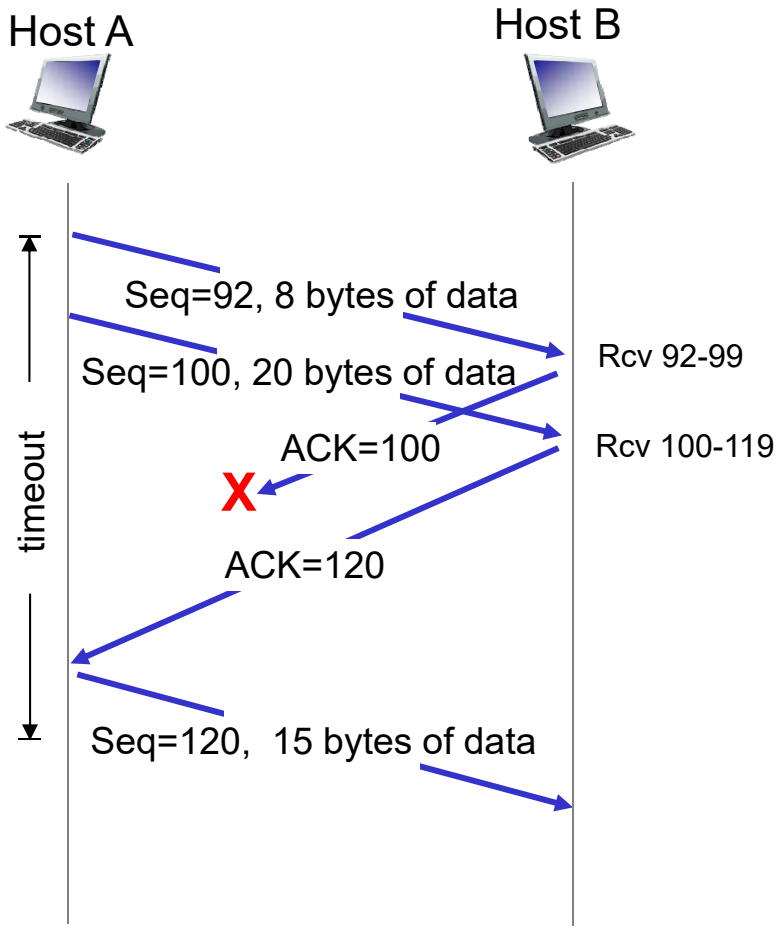



Lost ACK scenario




Premature timeout
segment 100 not retransmitted

# TCP: Retransmission Scenarios




Cumulative ACK

# TCP ACK Generation
## [RFC 1122, RFC 2581]

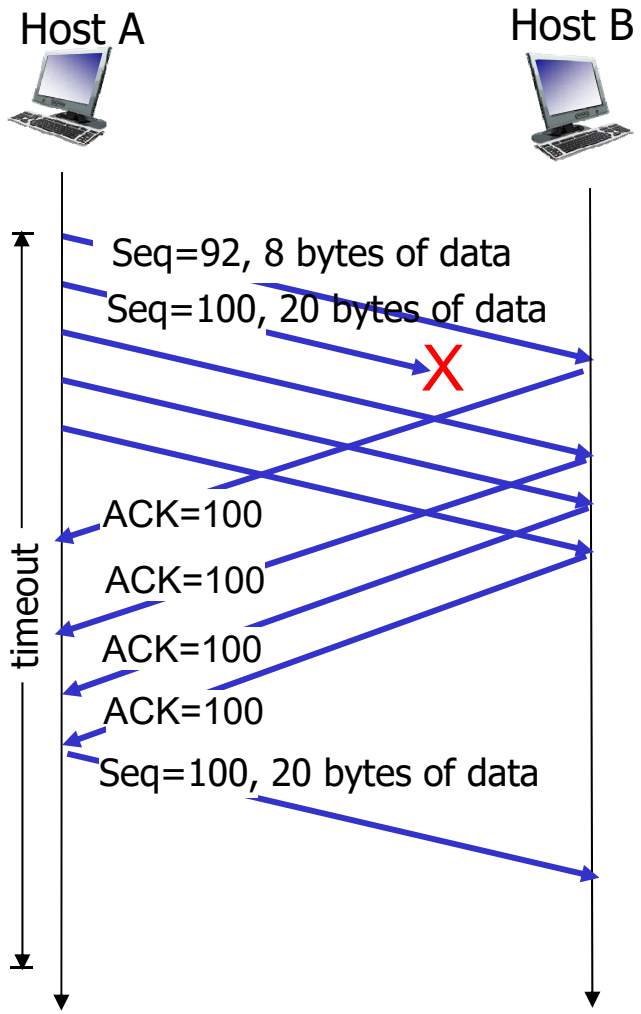
The hope is, within the delay, the receiver will have data ready to be sent to the receiver. Then, the ACK can be piggybacked with a data segment

| Event at Receiver | TCP Receiver Action |
|---|---|
| Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed | Delayed ACK. Wait up to 200 ms for next segment. If no next segment, send ACK. Registry: **`TcpAckFrequency`** |
| Arrival of in-order segment with expected seq #. One other segment has an 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 duplicate ACK, 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 |

**Registry location:**
**`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters`**

# Fast Retransmit

- Timeout period can be 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
- If sender receives 3 **duplicate** ACKs for the same data, it supposes that segment after ACKed data was lost:
  - Fast retransmit: resend segment before timer expires

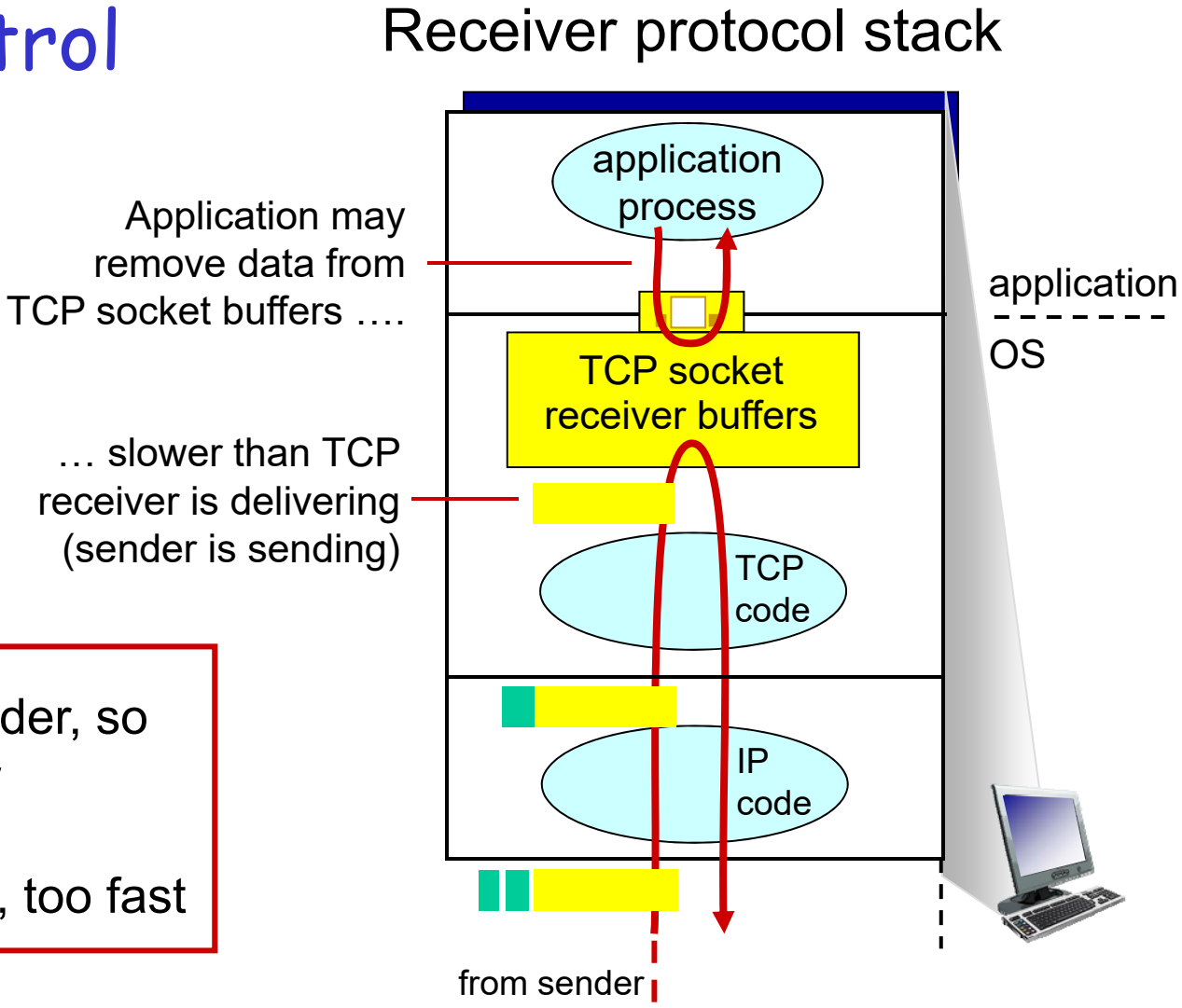



Fast retransmit after receipt of 3rd duplicate ACK

# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# TCP Flow Control



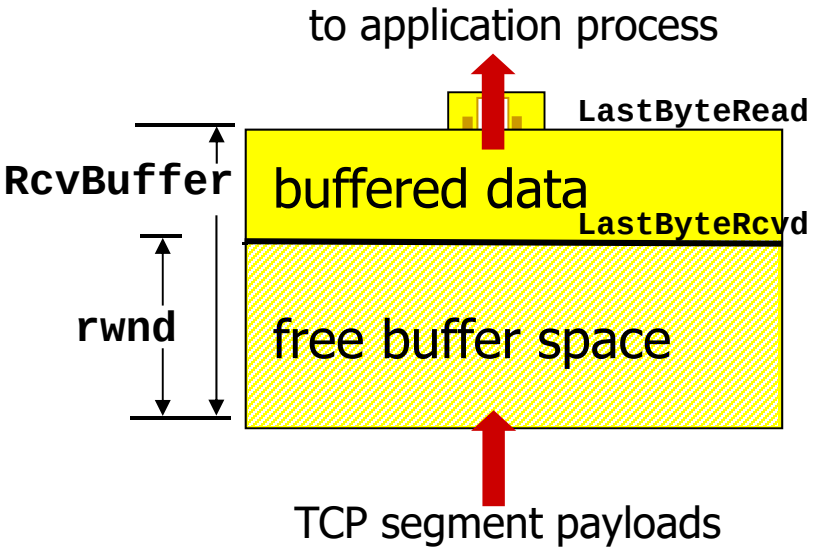

*flow control*

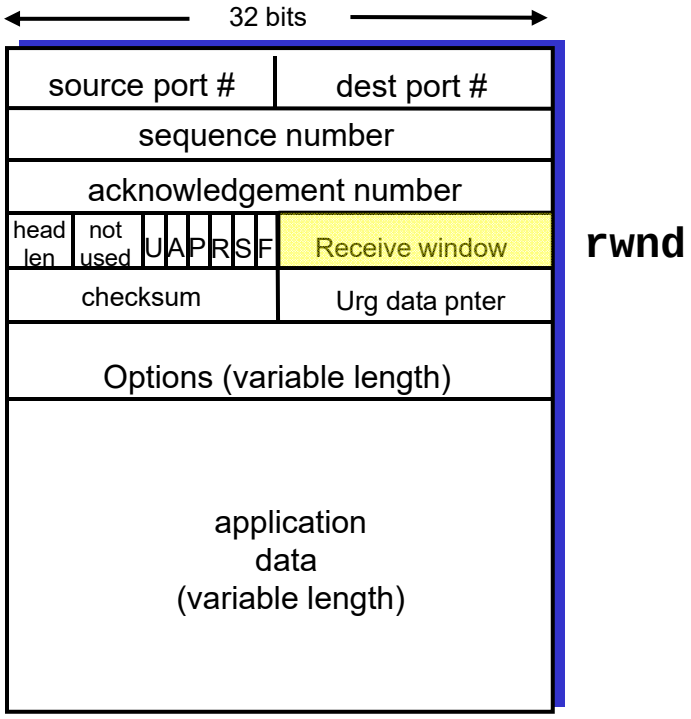
Receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

# 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)
- Sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- Guarantees receive buffer will not overflow
- Spare room in buffer

**rwnd = RcvBuffer-**
**[LastByteRcvd –**
**LastByteRead]**



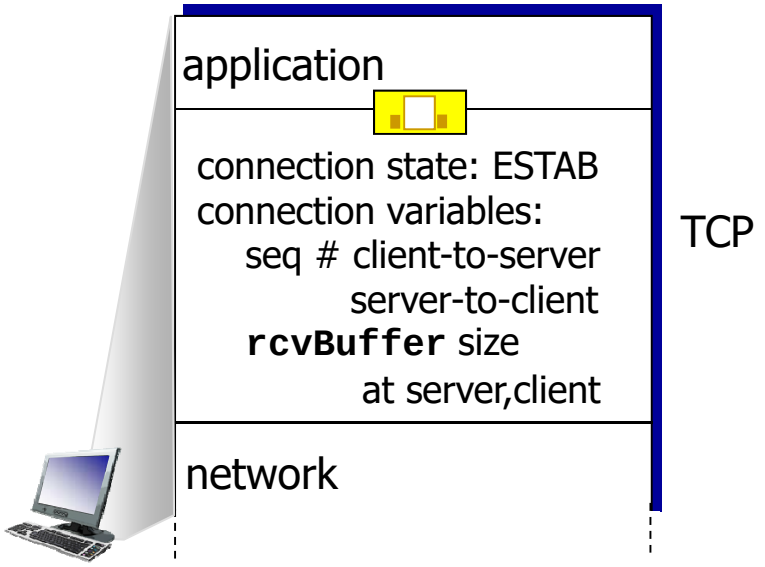
# Chapter 3 Outline

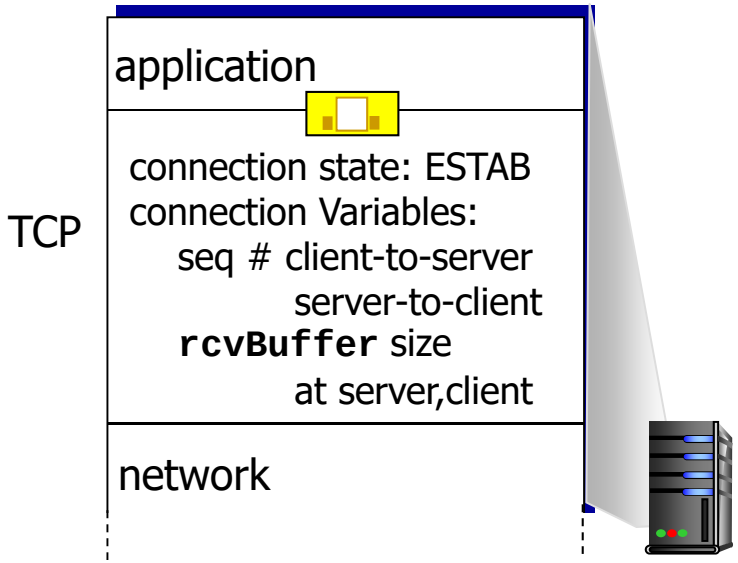
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# Connection Management

Before exchanging data, sender/receiver "handshake"

- Agree to establish connection
  - Each knowing the other willing to establish connection
- Agree on connection parameters

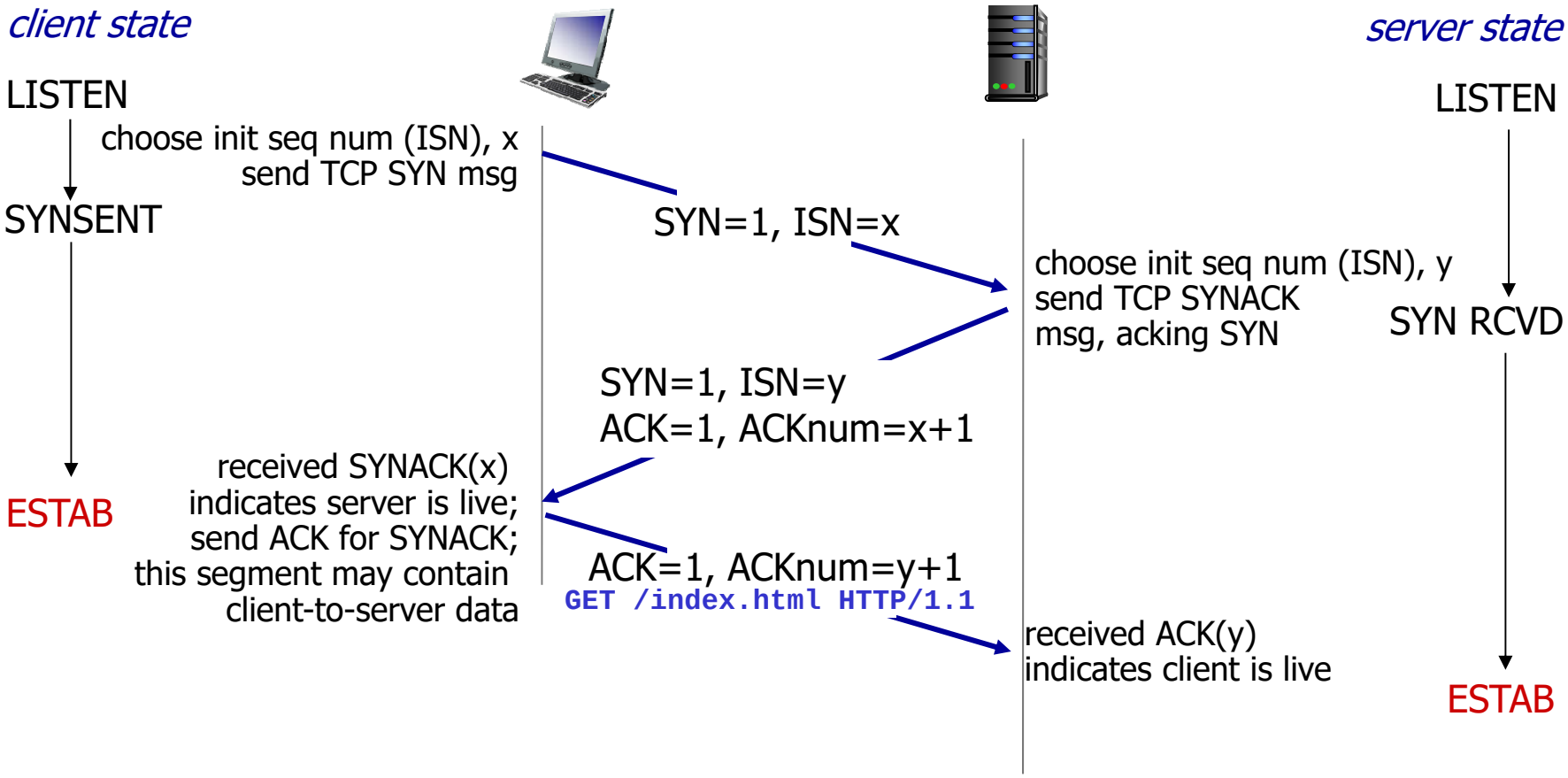


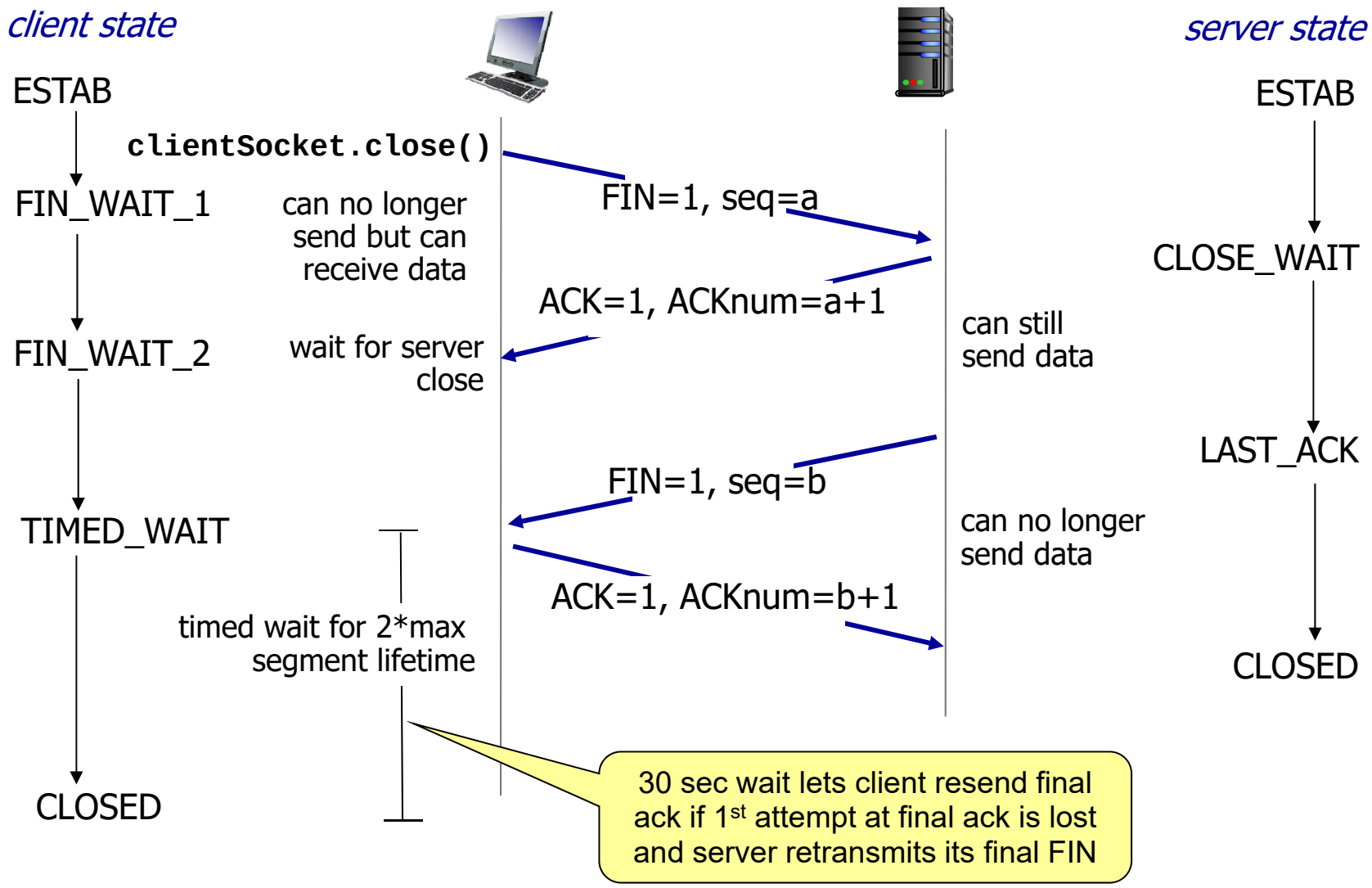

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
```

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

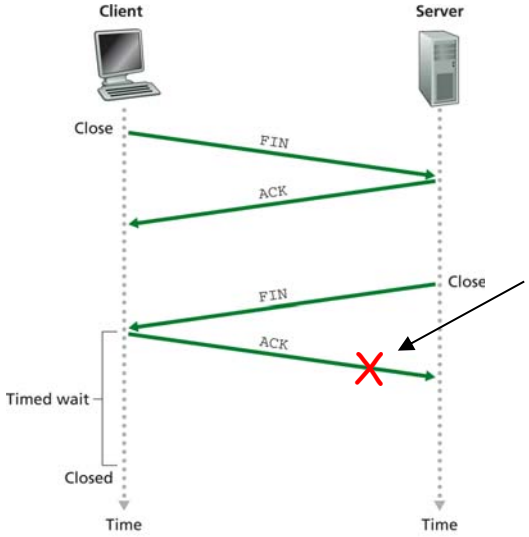
# TCP 3-way Handshake



# TCP: Closing A Connection

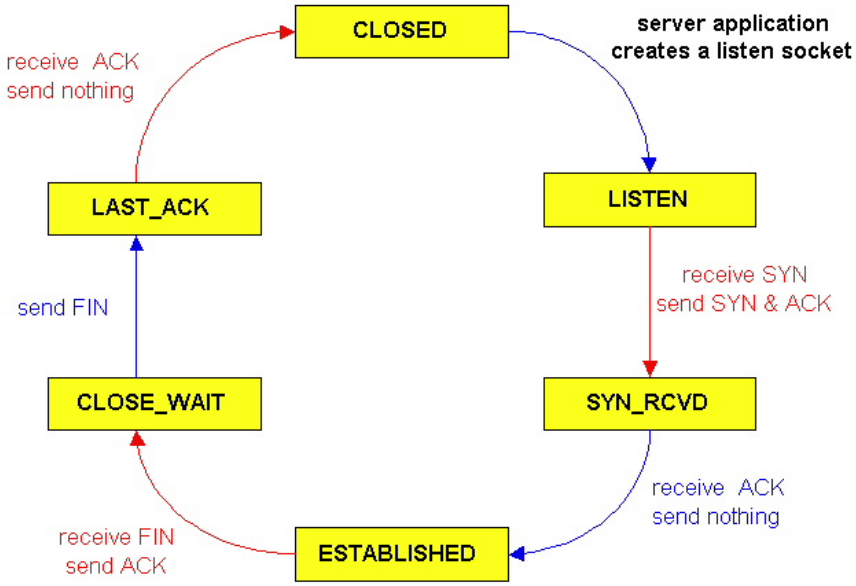


# TCP Connection Lifecycles

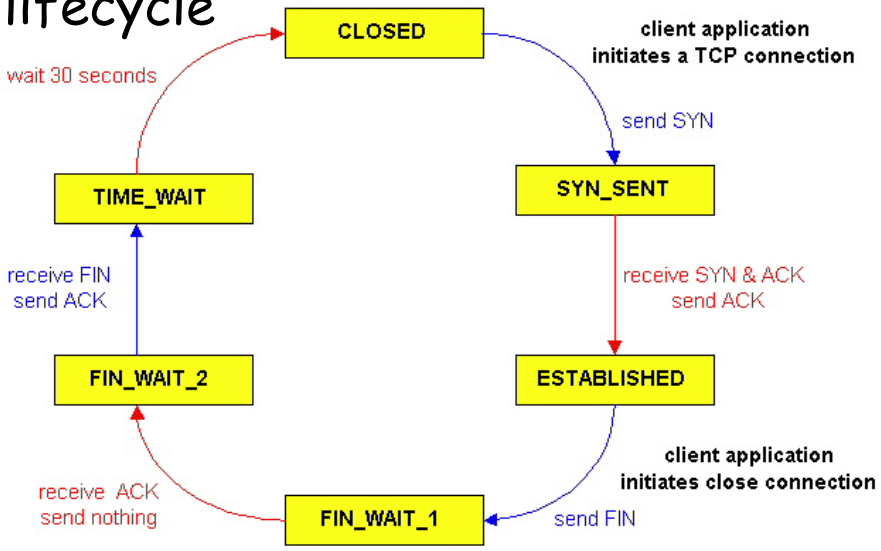


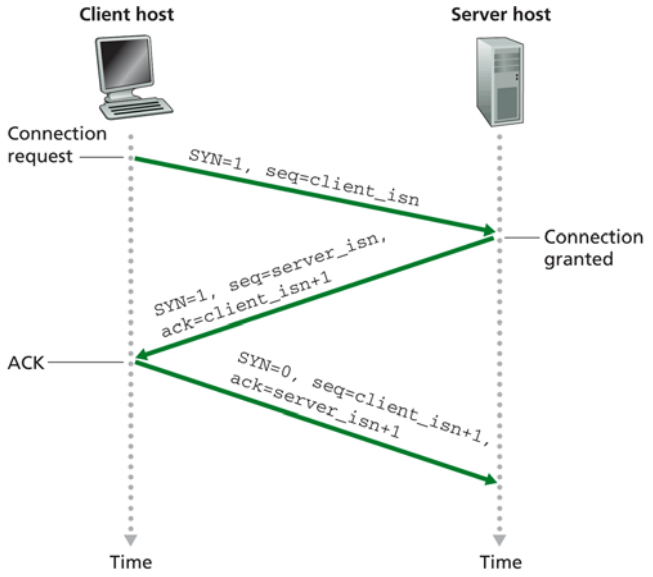

30 sec wait lets client resend final ack if 1st attempt at final ack is lost and server retransmits its final FIN

TCP server lifecycle




TCP client lifecycle



# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# Principles of Congestion Control

Congestion:

- Informally: "too many sources sending too much data too fast for the network to handle"
  - "Network" is not referring to the network layer but the network between sender and receiver

- Different from flow control!

- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queuing in router buffers)

# Chapter 3 Outline

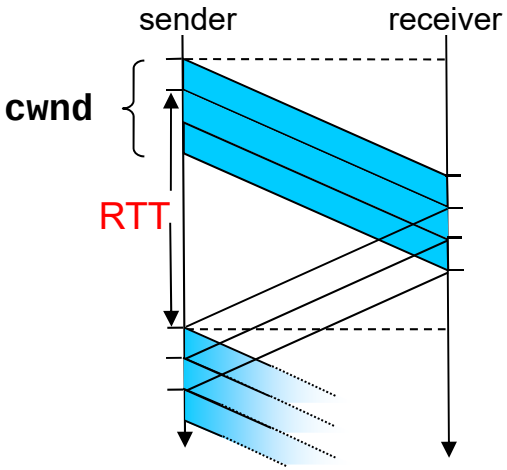
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# TCP Congestion Control

- End-end control (no network assist)
- Congestion window (**cwnd**) is dynamic function of perceived network congestion
  - Ideally make **cwnd** as large as possible
    - Transmit as fast as possible
- Sender limits transmission

  **LastByteSent-LastByteAcked**
  **≤ min{cwnd, rwnd}**
- Roughly,

$$\text{rate} = \frac{\textbf{cwnd}}{\text{RTT}} = \frac{\text{W*MSS}}{\text{RTT}} \quad \text{Bytes/sec}$$




W = window size in segments
MSS = max segment size (data only)

## How does sender perceive congestion?

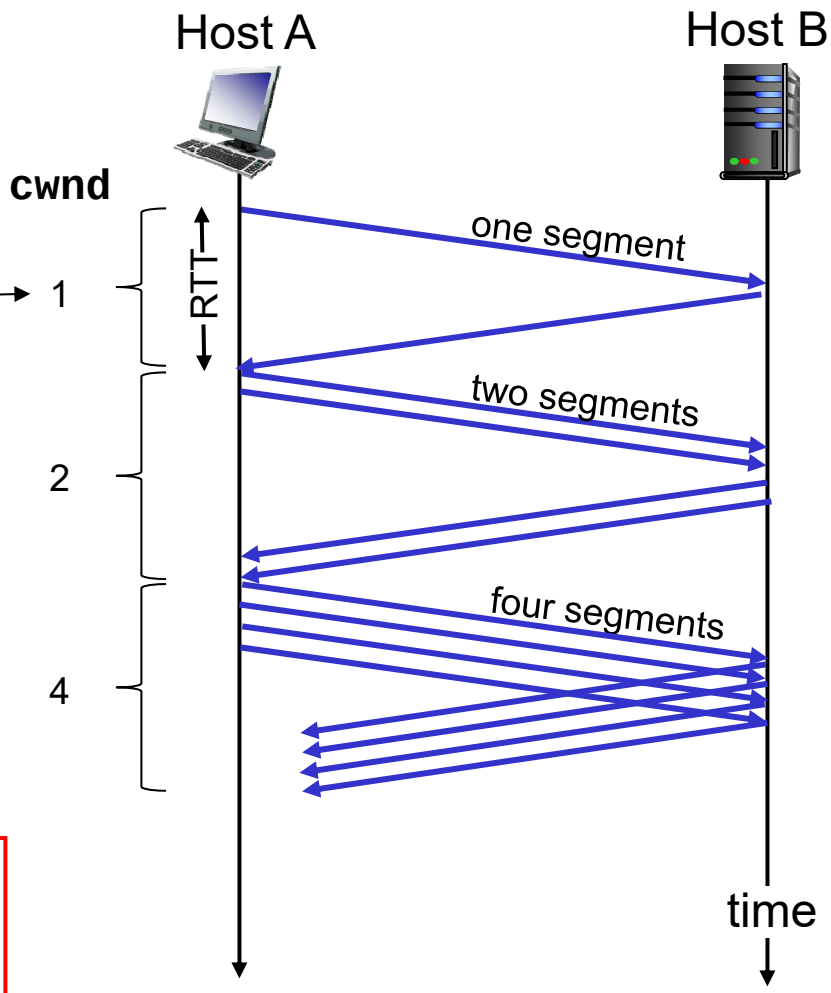
- Loss event = timeout or 3 duplicate acks
- TCP sender reduces rate (**cwnd**) after loss event
- Acks regulate the send rate
  - Slower ack rate = slower increase in **cwnd**

## TCP Congestion Control Algorithm has 2 phases:

- Slow start
- Congestion Avoidance
  - AIMD (Additive Increase Multiplicative Decrease)

# 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

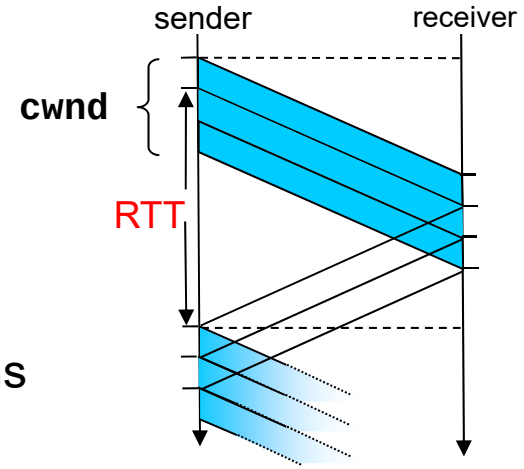


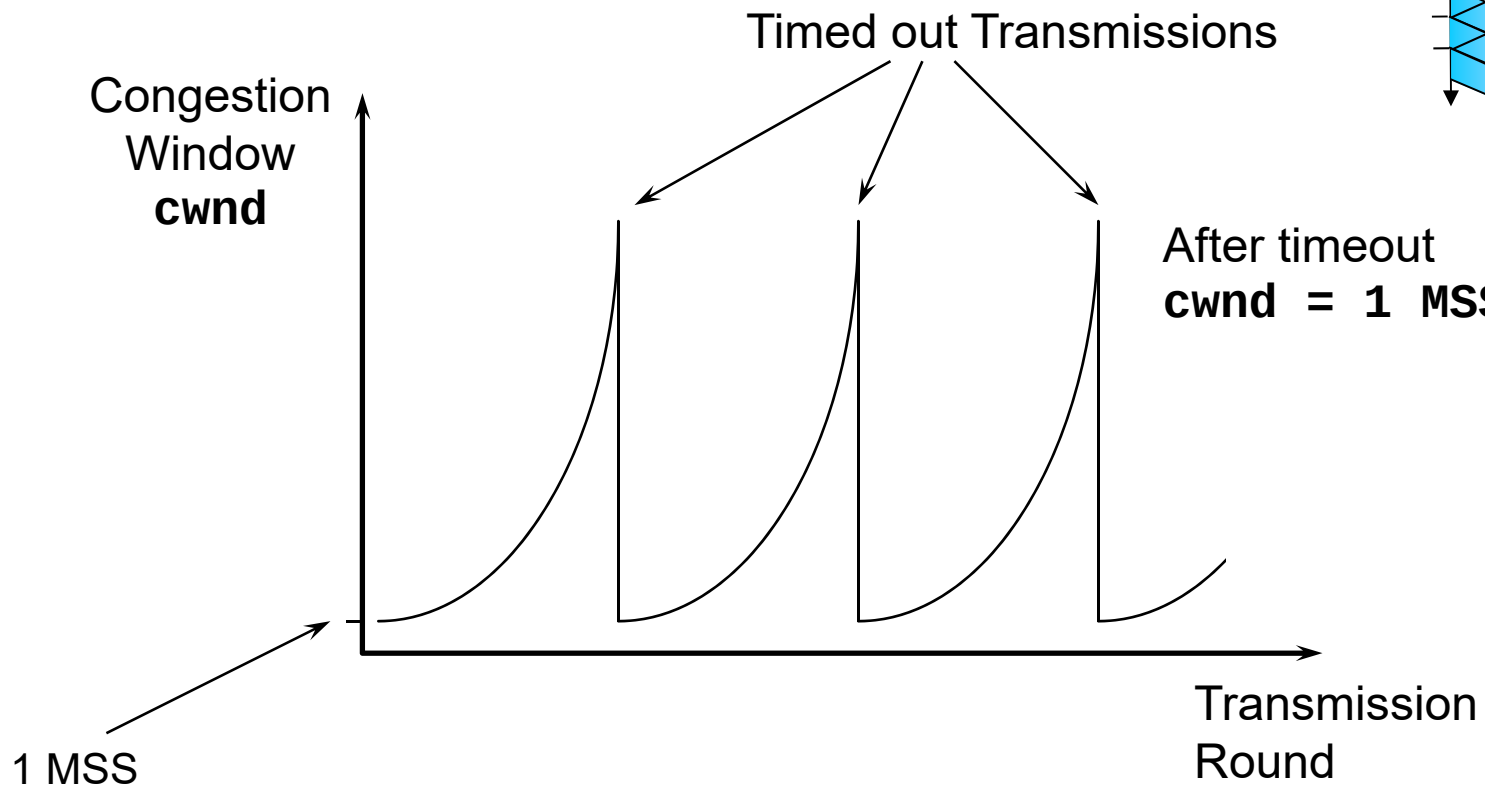

**Slow Start algorithm**

initialize: cwnd = 1 MSS
for (each segment ACKed)
    cwnd++
until (loss event OR cwnd > threshold)

# TCP Slow Start

- It works but performance is not all that great



# Refinement

- After timeout event:
  - **cwnd** set to 1 MSS
  - Window then grows exponentially (slow start)…
  - …to a threshold, then grows linearly (congestion avoidance)

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- Timeout before 3 dup ACKs is "more alarming"

- After 3 dup ACKs: (TCP Reno)
  - **cwnd** is cut in half
  - Window then grows linearly starting at the threshold
    - "Fast recovery" (optional) sets the new **cwnd** to threshold + 3
      - We will not use Fast recovery
- Note: TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)

# TCP AIMD (Congestion Avoidance)

## Additive increase

Increase **cwnd** a little each time an ACK is received with the goal of increasing **cwnd** by 1 MSS every RTT in the absence of loss events: *probing*

**cwnd = cwnd + MSS * (MSS/cwnd)**

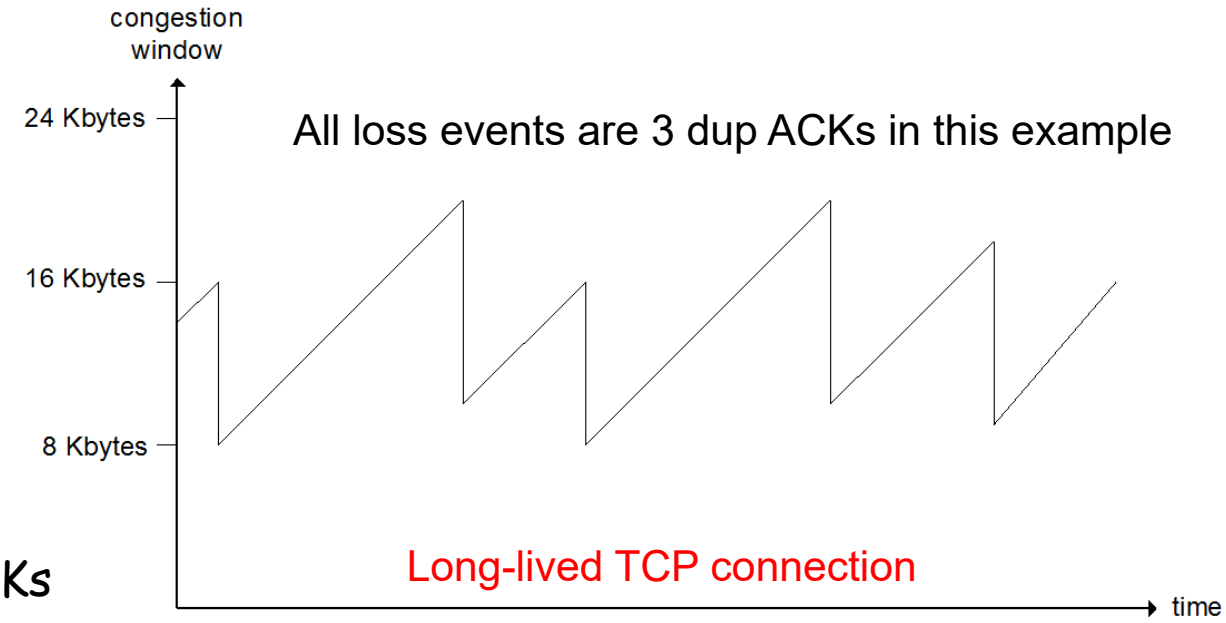
## Multiplicative decrease

Cut **cwnd** in half after 3 dup acks

Example:
MSS = 1,460 bytes

If window = 10,
**cwnd** = 14,600 bytes

**cwnd** would increase
by ~146 bytes per ACK
or one MSS after 10 ACKs

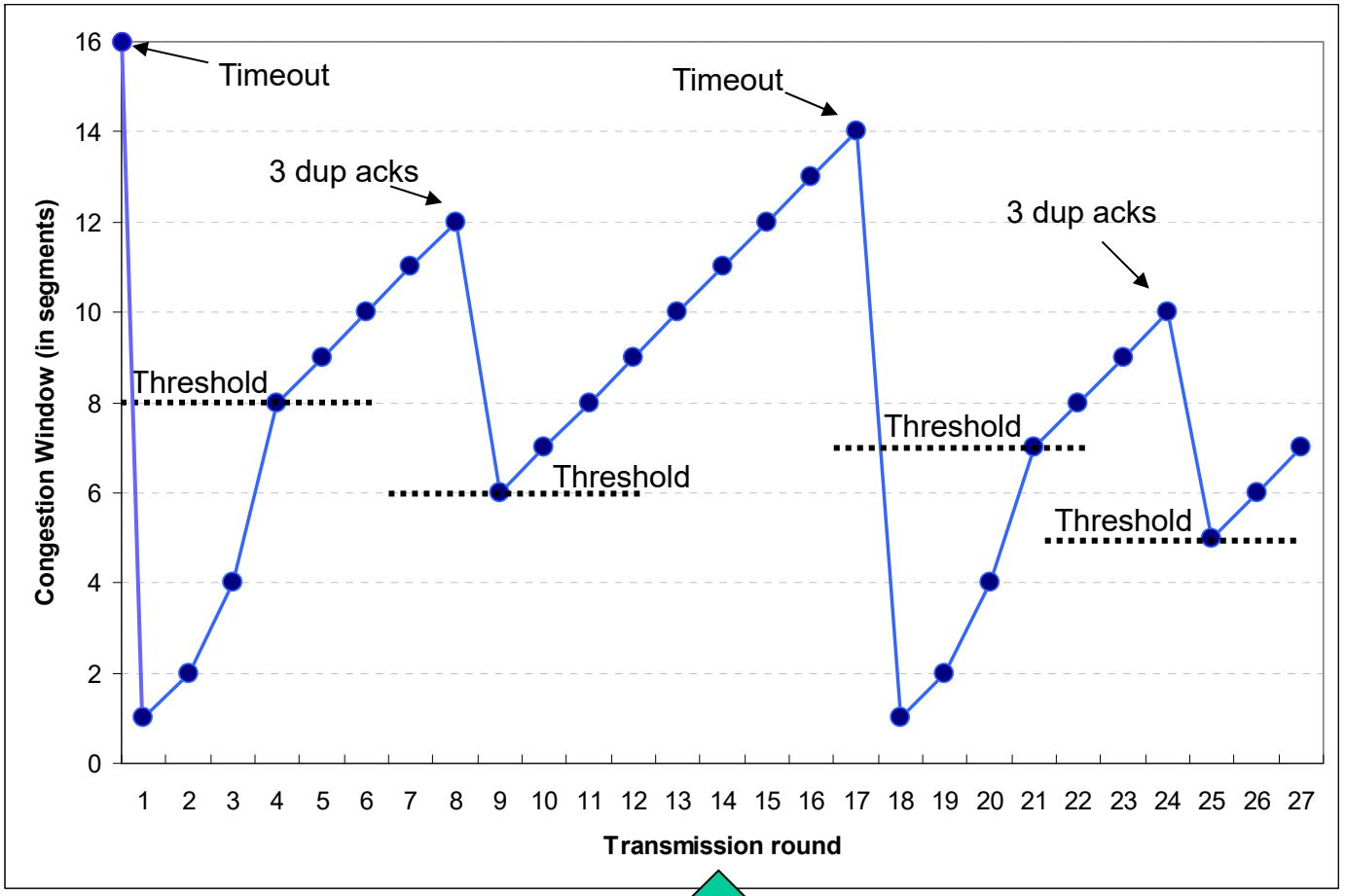


# TCP Reno Performance

This trace is not the beginning of a connection but somewhat into it

Q: When should the exponential increase (SS) switch to linear (Additive increase)?

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

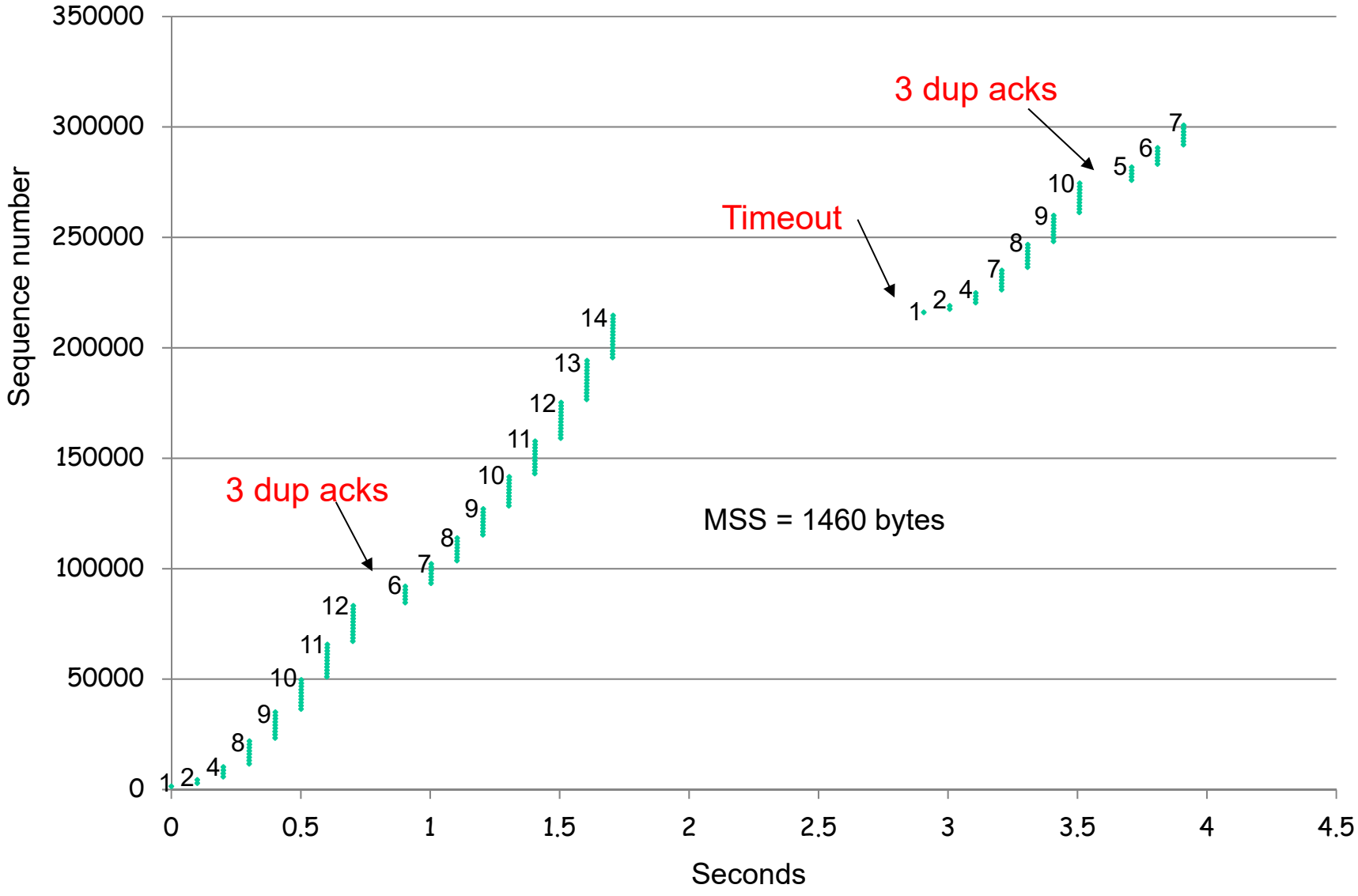



Note: This is NOT time!

## Implementation:

- Variable Threshold
- Initial threshold set to 65 Kbytes
- At loss event, threshold is set to 1/2 of **cwnd** just before loss event

# TCP Reno Performance - Stevens Graph



# Summary: TCP Congestion Control

- When **cwnd** is below Threshold, sender in slow-start phase, window grows exponentially.
- When **cwnd** is above Threshold, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, Threshold set to **cwnd/2** and **cwnd** set to Threshold. Start *CA*.
- When timeout occurs, Threshold set to **cwnd/2** and **cwnd** is set to 1 MSS. Start SS.

# TCP Sender Congestion Control

| State | Event | TCP Sender Action | Commentary |
|---|---|---|---|
| Slow Start (SS) | ACK receipt for previously unacked data | cwnd = cwnd + MSS,<br>If (cwnd > Threshold)<br>set state to “Congestion Avoidance” | Resulting in a doubling of cwnd every RTT |
| Congestion Avoidance (CA) | ACK receipt for previously unacked data | cwnd = cwnd+MSS * (MSS/cwnd) | Additive increase, resulting in increase of cwnd by 1 MSS every RTT |
| SS or CA | Loss event detected by triple duplicate ACK | Threshold = cwnd/2,<br>cwnd = Threshold,<br>Set state to “Congestion Avoidance” | Fast recovery, implementing multiplicative decrease. cwnd will not drop below 1 MSS. |
| SS or CA | Timeout | Threshold = cwnd/2,<br>cwnd = 1 MSS,<br>Set state to “Slow Start” | Enter slow start |
| SS or CA | Duplicate ACK | Increment duplicate ACK count for segment being acked | cwnd and Threshold not changed |