COMPUTER NETWORKS

Animesh Giri

Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri
Department of Computer Science & Engineering

Transport Layer – 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

In this segment

- Transport layer goals
- Transport layer services
- Transport services & protocols
- Transport vs Network layer
- Transport layer actions
- Internet transport layer protocols

Transport Layer - Goals

- Understand principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

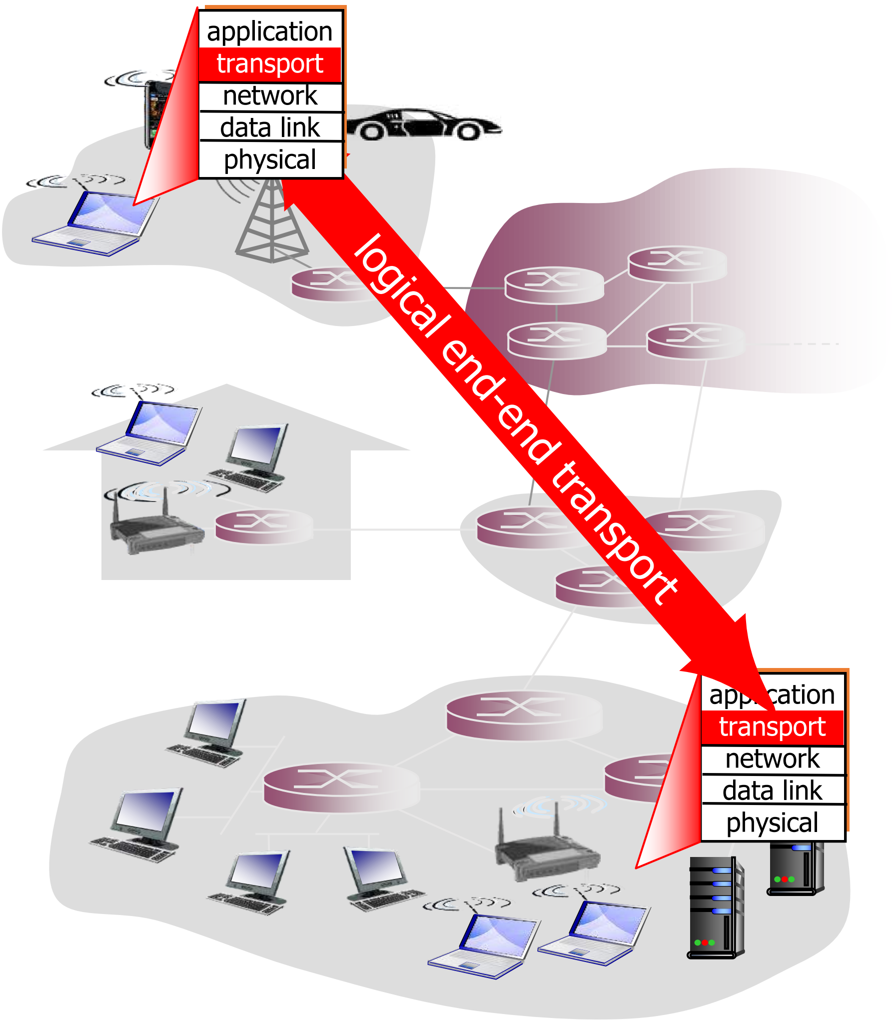
COMPUTER NETWORKS

Transport Layer Services

Animesh Giri
Department of Computer Science & Engineering

Transport Services & protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - **send side:** breaks app messages into *segments*, passes to network layer
  - **rcv side:** reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. Network Layer

- *Network layer:* logical communication between hosts
- *Transport layer:* logical communication between processes
  - relies on, enhances, network layer services

household analogy:

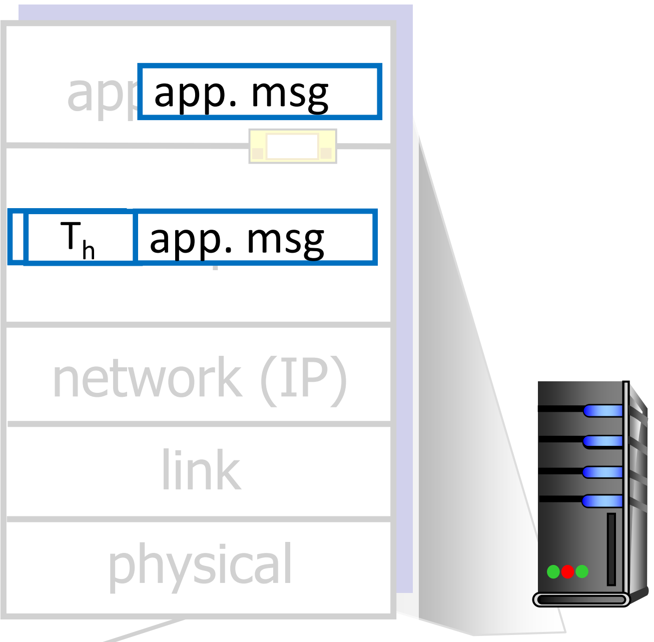
12 kids in Ann's house sending letters to 12 kids in Bill's house:

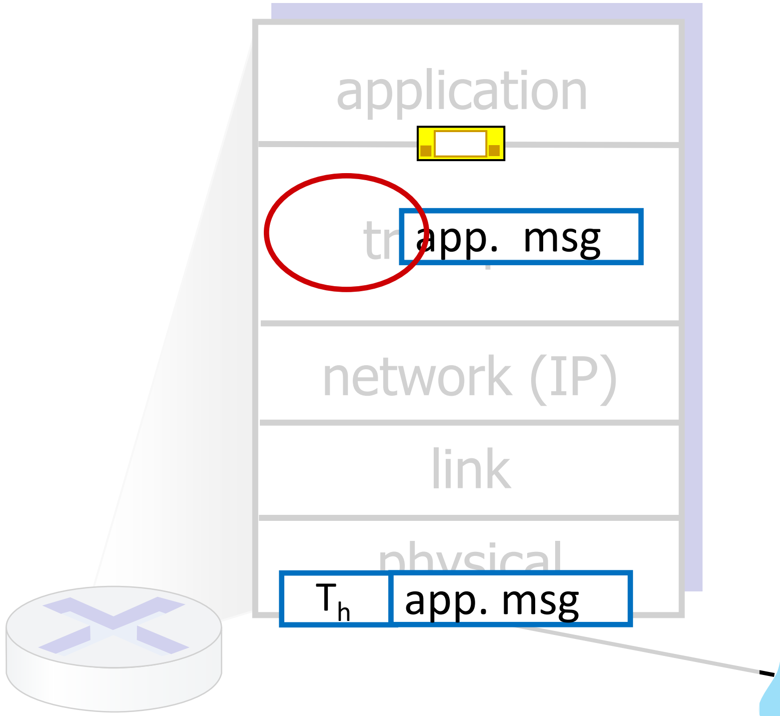
- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Transport-layer Actions

Sender:

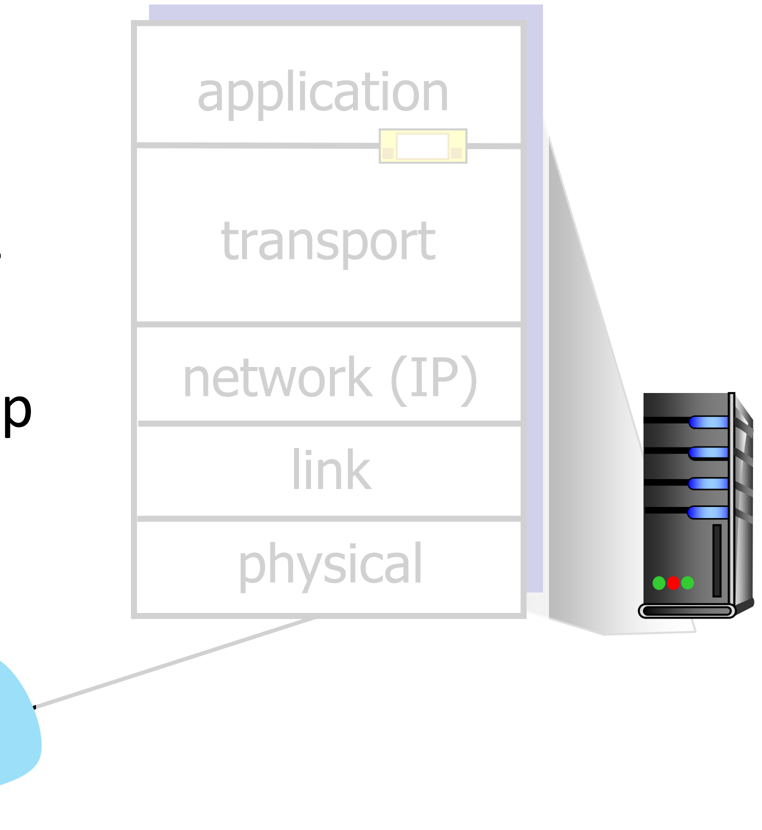
- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP





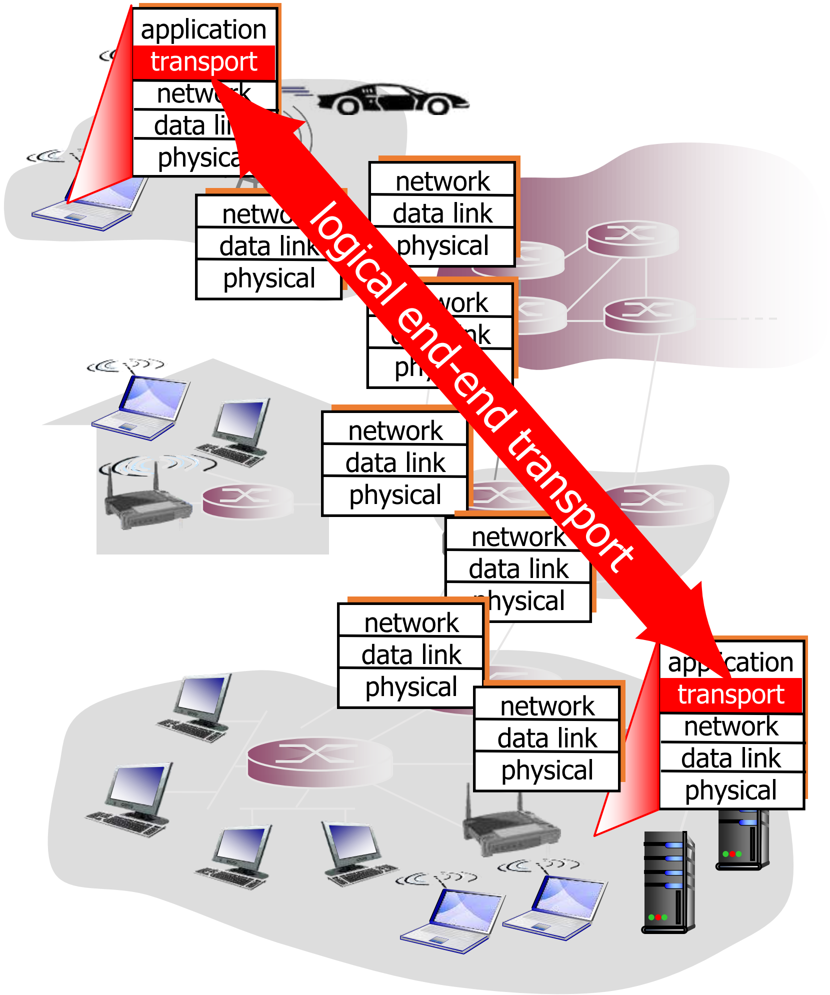
Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket



Internet Transport-layer protocols

- **TCP:** Transmission Control Protocol
  - reliable, connection oriented
  - in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, connectionless
  - unordered delivery
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603

COMPUTER NETWORKS

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Multiplexing & Demultiplexing

Animesh Giri
Department of Computer Science & Engineering

In this segment

What is transport-layer multiplexing and demultiplexing

- How demultiplexing works
  - TCP / UDP segment format
- Connectionless demultiplexing - UDP
- Connectionless demux: Example
- Connection-oriented demux - TCP
- Connection-oriented demux: Example

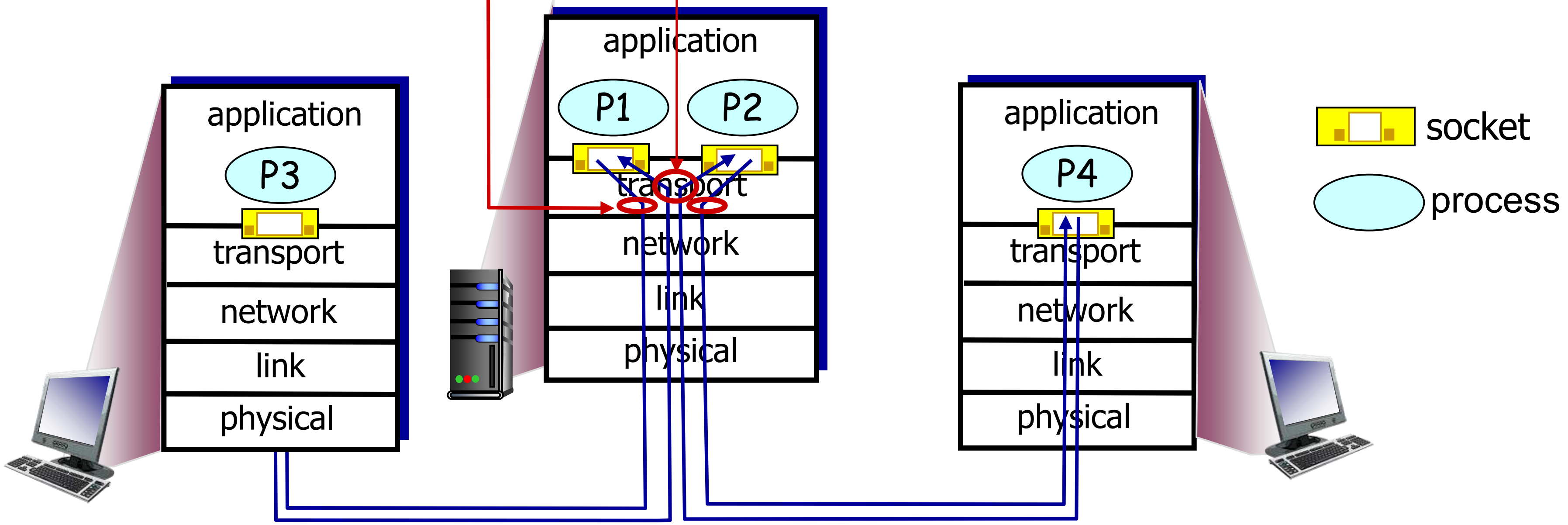
Multiplexing / demultiplexing

multiplexing at sender:

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

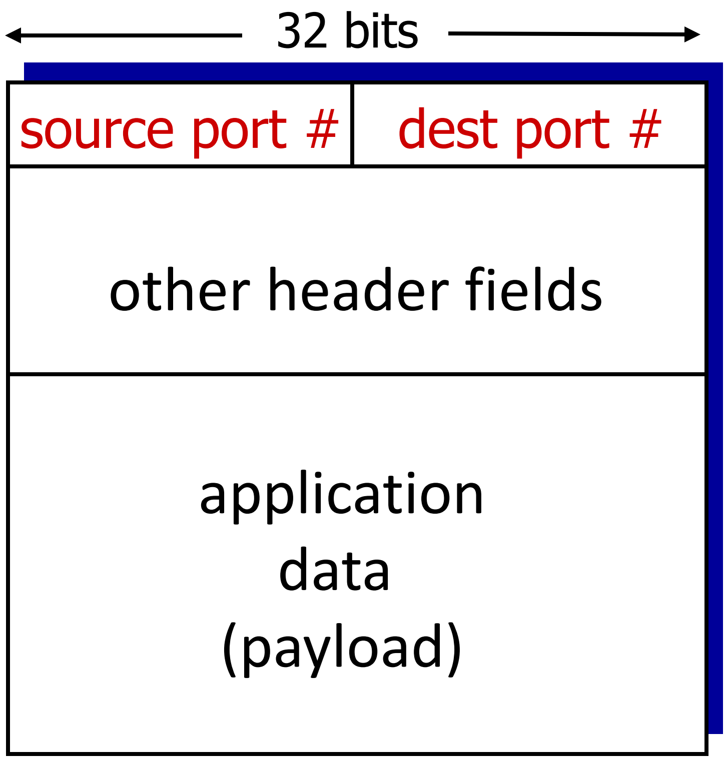
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

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



TCP/UDP segment format

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

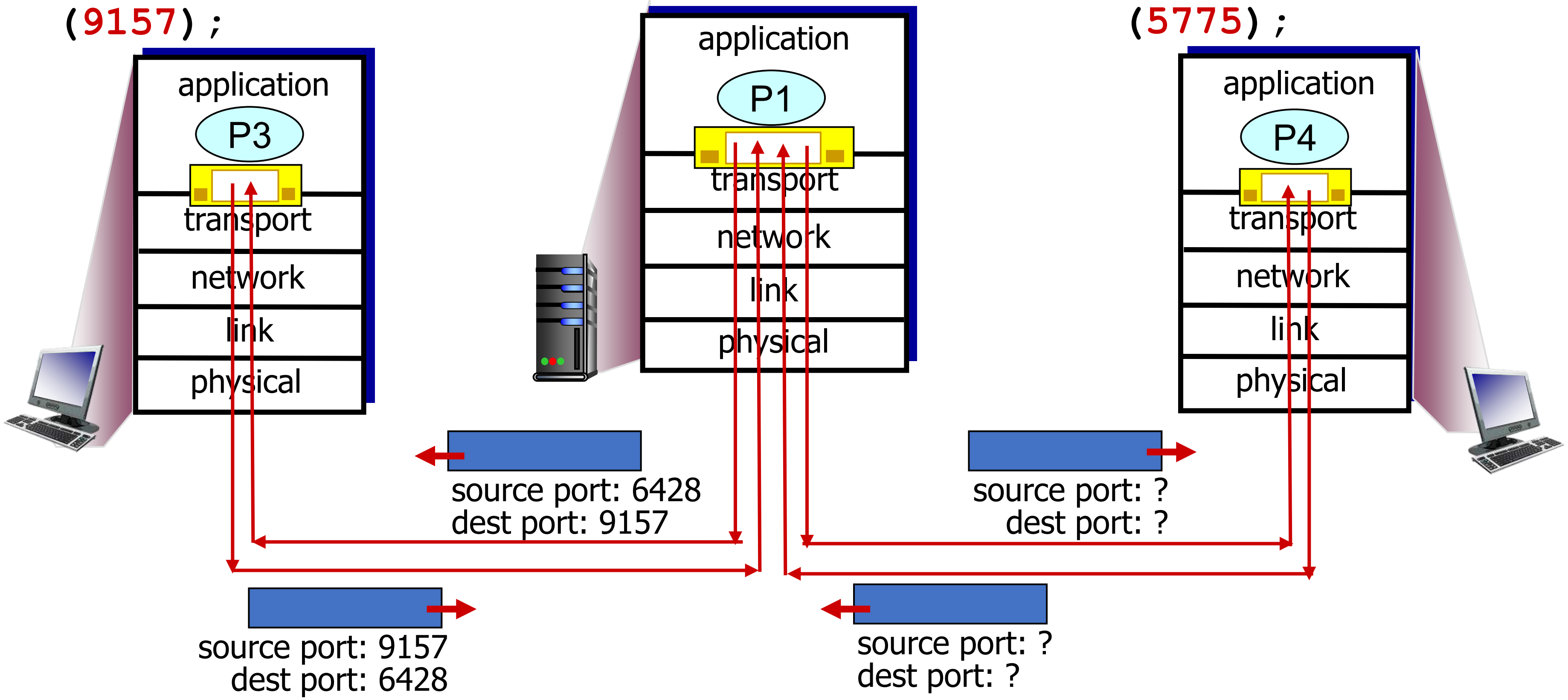
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

```
DatagramSocket serverSocket
  = new DatagramSocket
 (6428);
```

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```

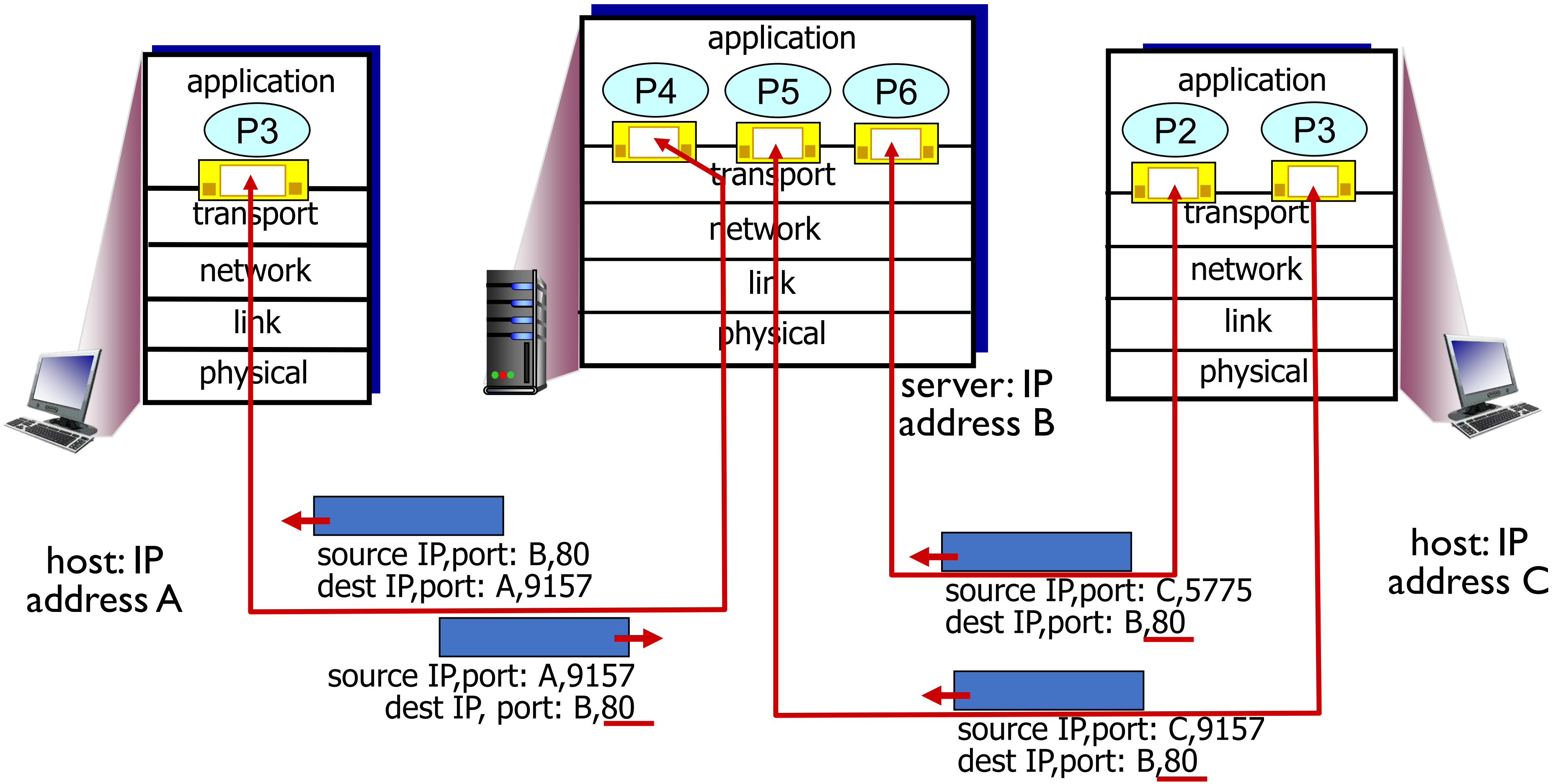


Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values (4-tuple) to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux : example



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

THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603

COMPUTER NETWORKS

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri

Department of Computer Science & Engineering

COMPUTER NETWORKS

Connectionless Transport: UDP

Animesh Giri

Department of Computer Science & Engineering

In this segment

- UDP: User Datagram Protocol [RFC 768]
- UDP: segment header
- UDP Checksum
  - Internet Checksum: example
  - Internet Checksum: weak protection!
- Summary

UDP: User Datagram Protocol [RFC 768]

PES UNIVERSITY ONLINE

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

why is there a UDP?

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

UDP: User Datagram Protocol [RFC 768]

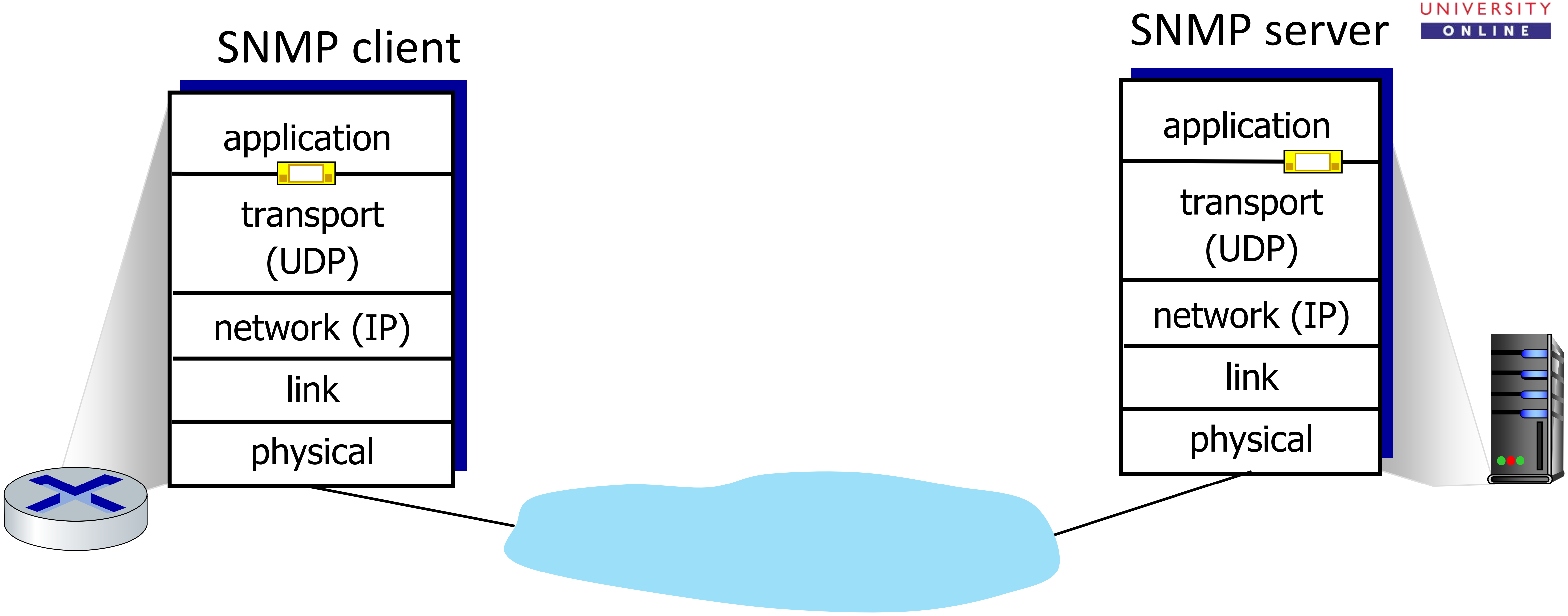
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP

- If reliable transfer needed over UDP:
  - add needed reliability at application layer
  - application-specific error recovery!
  - add congestion control at application layer

UDP: User Datagram Protocol [RFC 768]

```
                                                  INTERNET STANDARD
RFC 768                                                   J. Postel
                                                                ISI
                                                     28 August 1980


                         User Datagram Protocol
                         ----------------------

Introduction
------------

This User Datagram  Protocol  (UDP)  is  defined  to  make  available  a
datagram   mode  of  packet-switched   computer   communication  in  the
environment  of  an  interconnected  set  of  computer  networks.   This
protocol  assumes  that the Internet  Protocol  (IP)  [1] is used as the
underlying protocol.

This protocol  provides  a procedure  for application  programs  to send
messages  to other programs  with a minimum  of protocol mechanism.  The
protocol  is transaction oriented, and delivery and duplicate protection
are not guaranteed.  Applications requiring ordered reliable delivery of
streams of data should use the Transmission Control Protocol (TCP) [2].

Format
------


                  0      7 8     15 16    23 24    31
                 +--------+--------+--------+--------+
                 |     Source      |   Destination   |
                 |      Port       |      Port       |
                 +--------+--------+--------+--------+
                 |                 |                 |
                 |     Length      |    Checksum     |
                 +--------+--------+--------+--------+
                 |
                 |          data octets ...
                 +---------------- ...
```

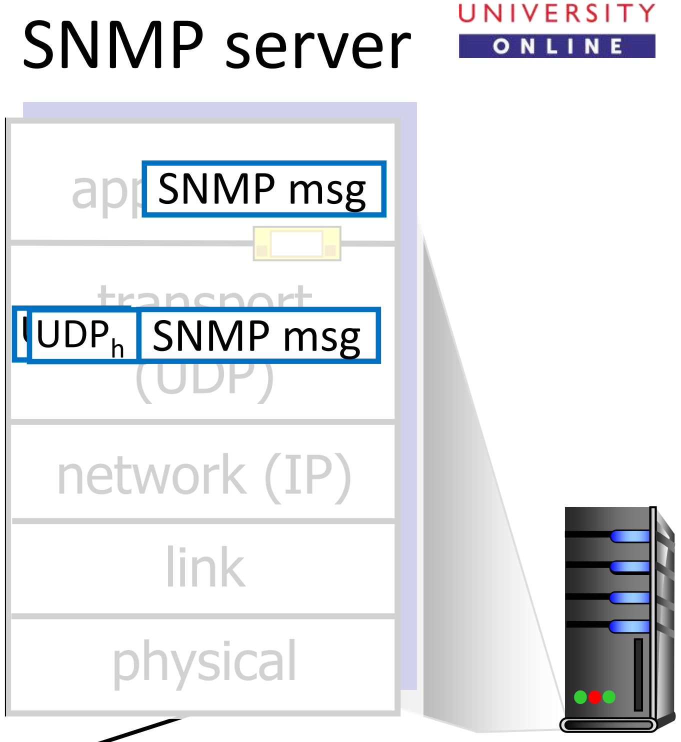
UDP: Transport Layer Actions



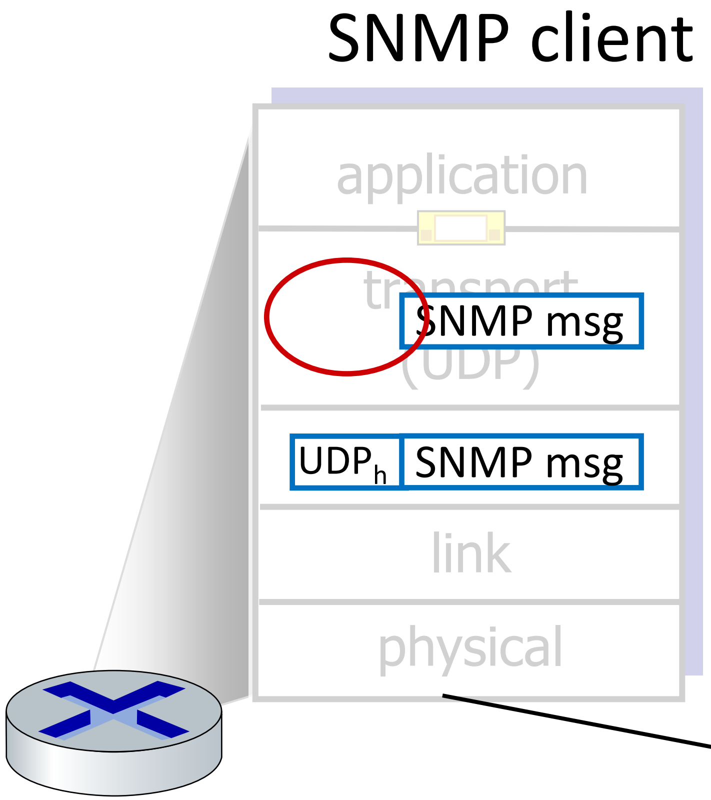
UDP: Transport Layer Actions

UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

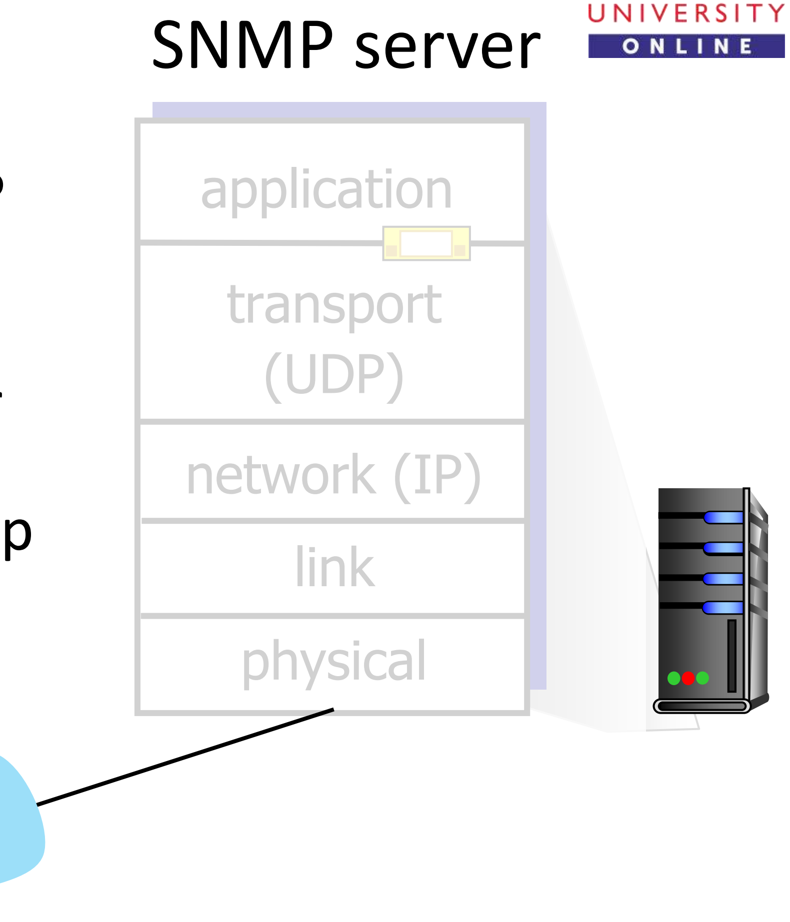


UDP: Transport Layer Actions

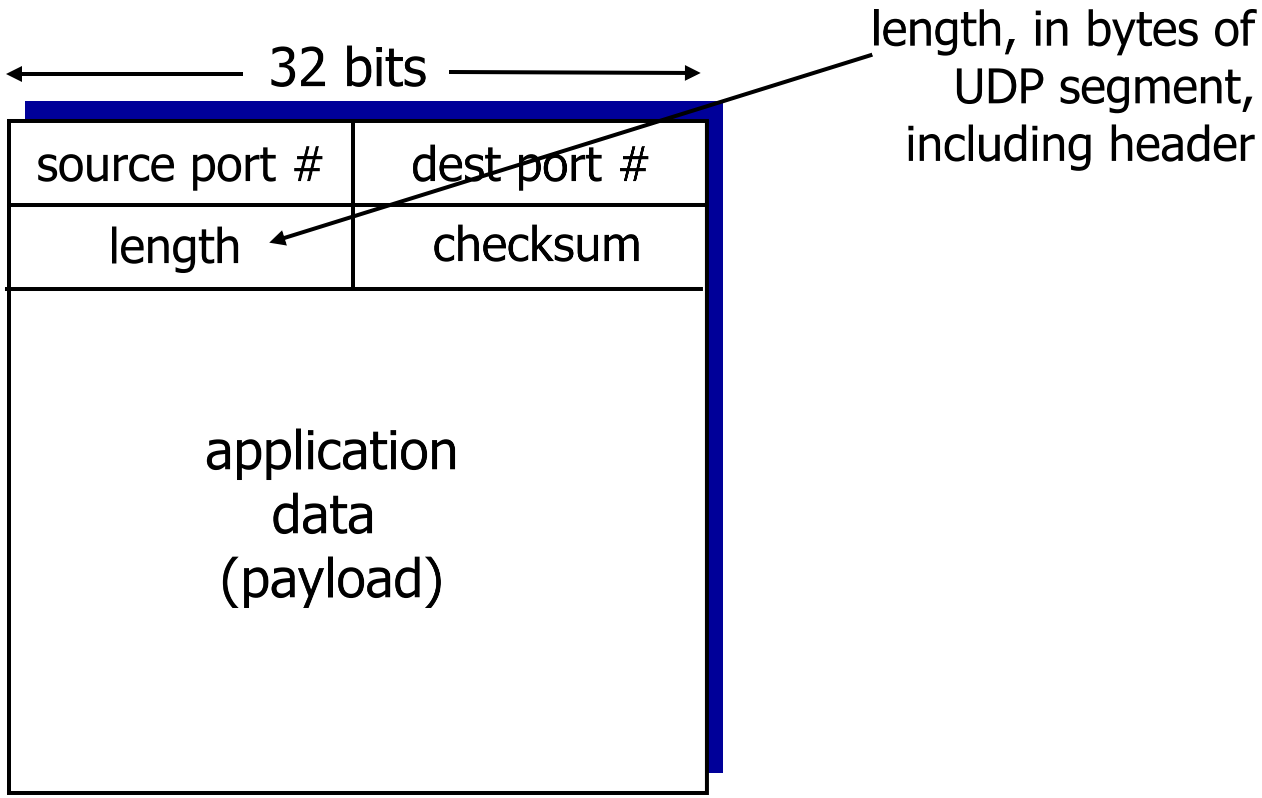


UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket



UDP: segment header



UDP segment format

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

	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11

receiver-computed checksum $\neq$ sender-computed checksum (as received)

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 maybe errors nonetheless?* More later …

Internet Checksum: example

example: add two 16-bit integers

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

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

Internet Checksum: Weak protection!

example: add two 16-bit integers

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

Even though numbers have changed (bit flips), *no* change in checksum!

Summary

- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”

- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)

- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603

COMPUTER NETWORKS

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Principles of reliable date transfer

Animesh Giri

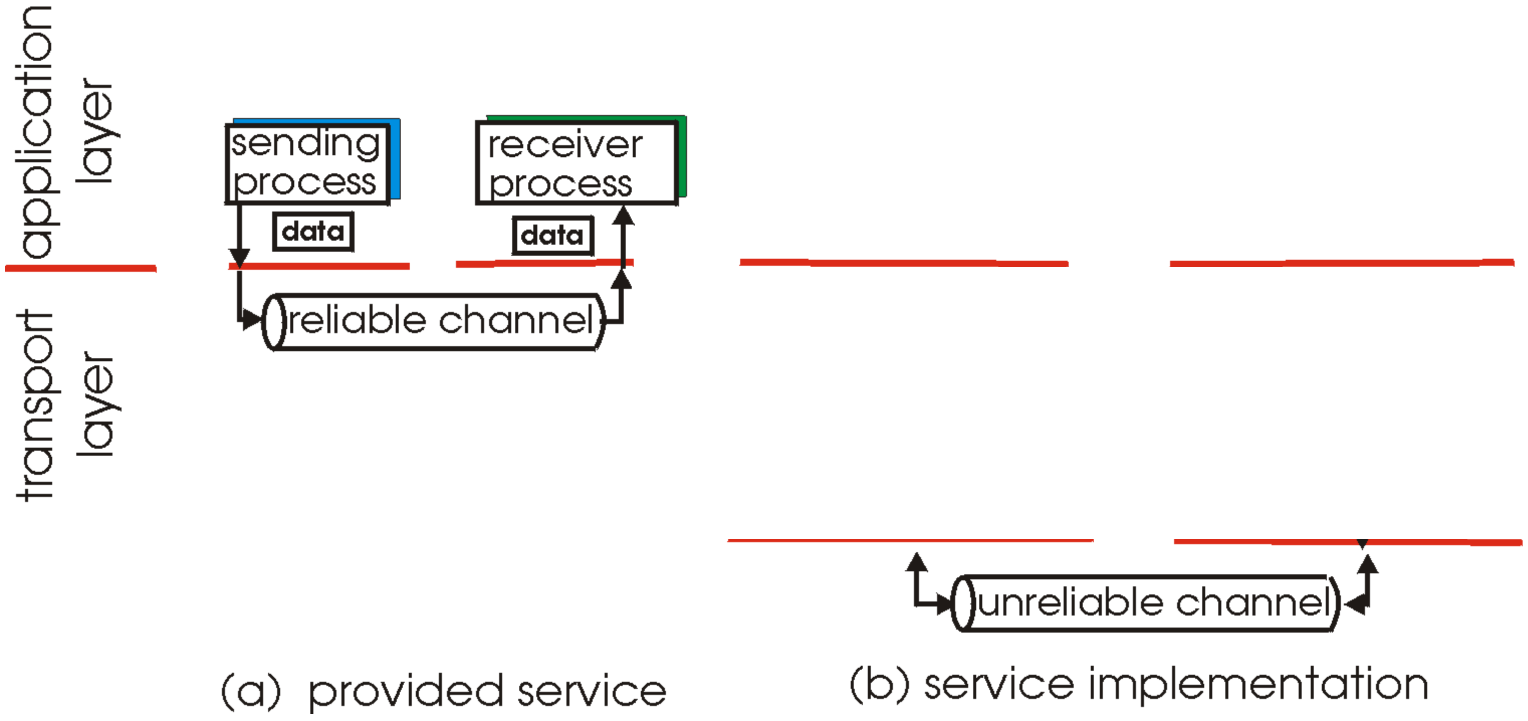
Department of Computer Science & Engineering

In this segment

- Principles of reliable data transfer
- Reliable data transfer: getting started
- rdt1.0: reliable transfer over a reliable channel
- Summary

Principles of reliable data transfer

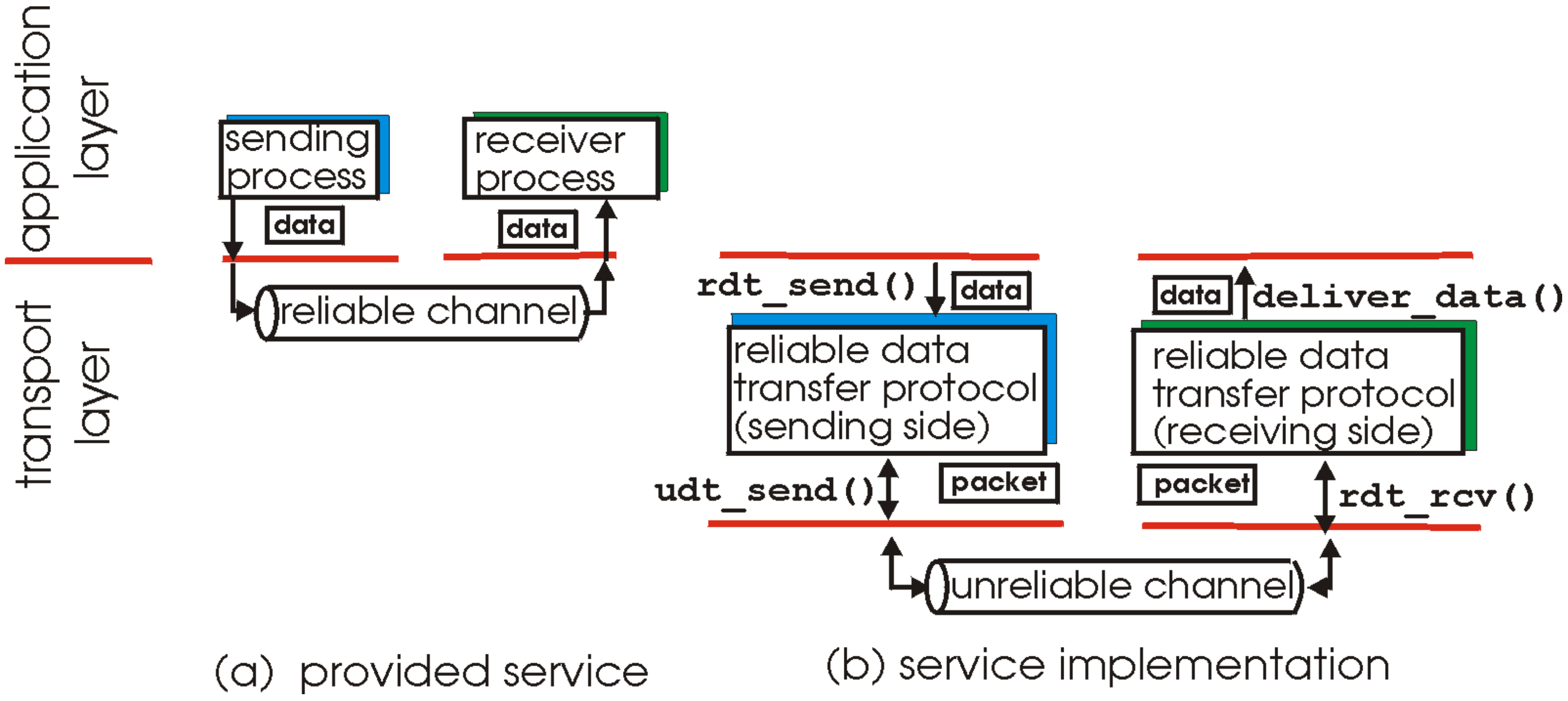
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

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

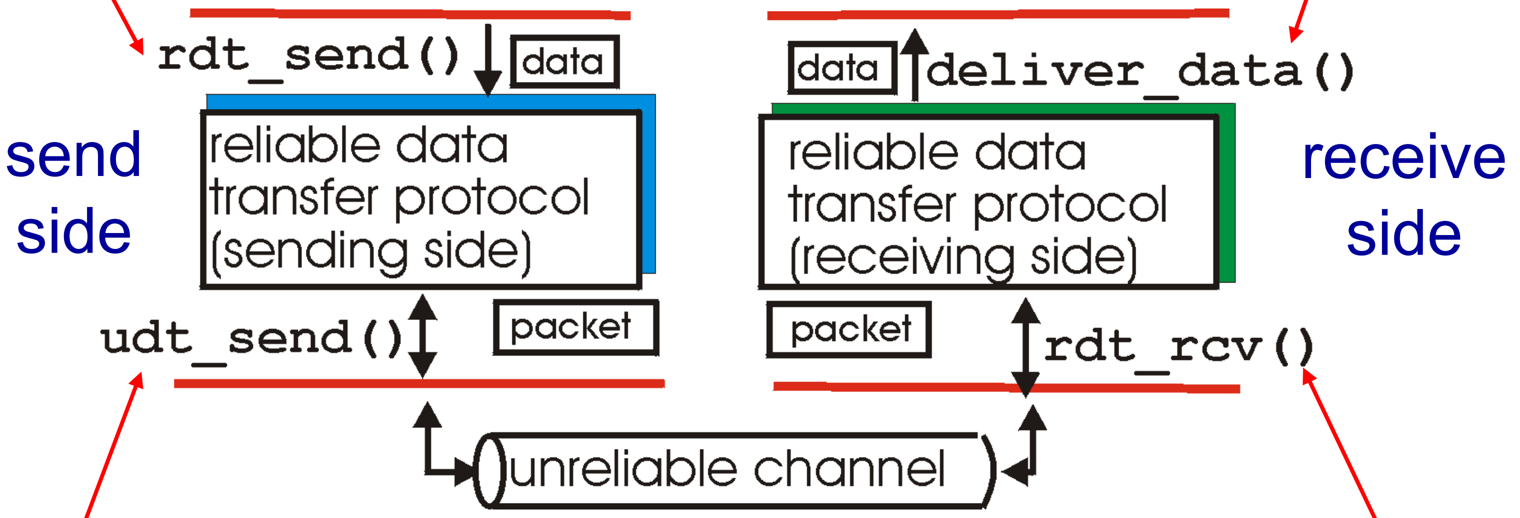


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



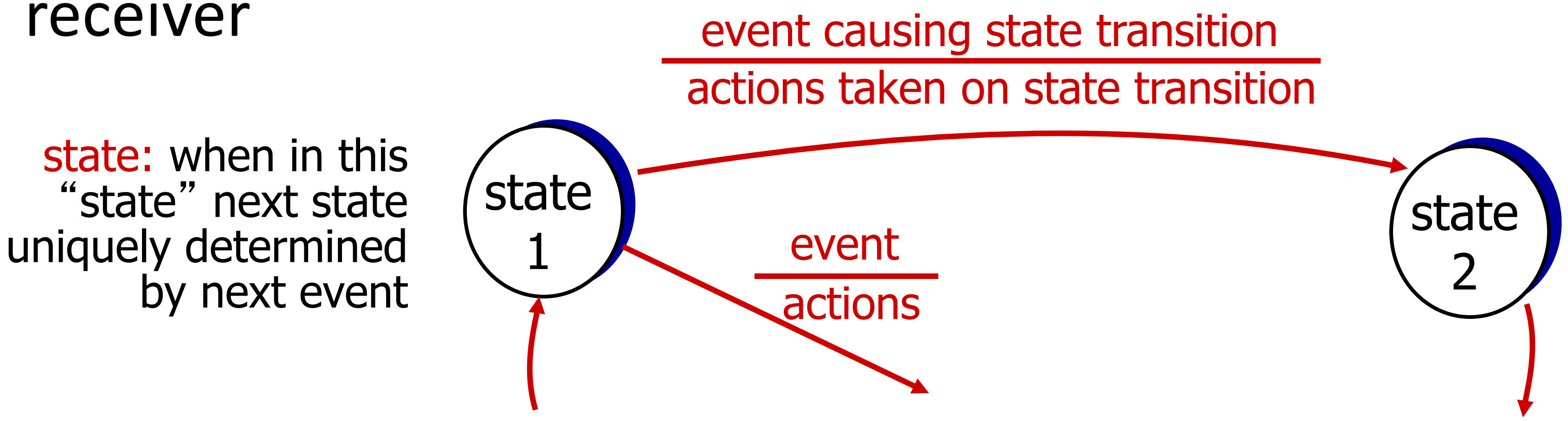
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

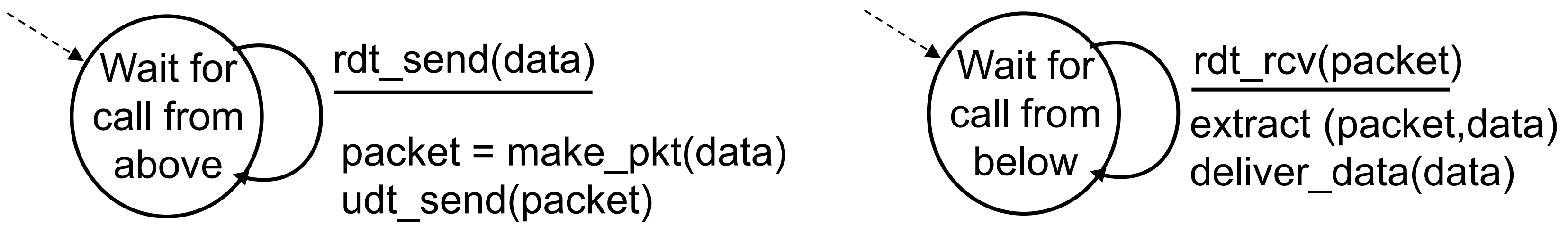
we'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver



rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel



sender

receiver

Summary

PES UNIVERSITY ONLINE

THANK YOU

Animesh Giri

Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603

COMPUTER NETWORKS

Transport Layer

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Principles of reliable date transfer

Animesh Giri

Department of Computer Science & Engineering

In this segment

- rdt2.0: channel with bit errors
- rdt2.0: FSM specification
- rdt2.0: operation with no errors
- rdt2.0: error scenario
- rdt2.0 has a fatal flaw!
- rdt2.1: sender, handles garbled ACK/NAKs
- rdt2.1: discussion
- rdt2.2: a NAK-free protocol
- rdt2.2: sender, receiver fragments
- Summary

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from “errors” during conversation?

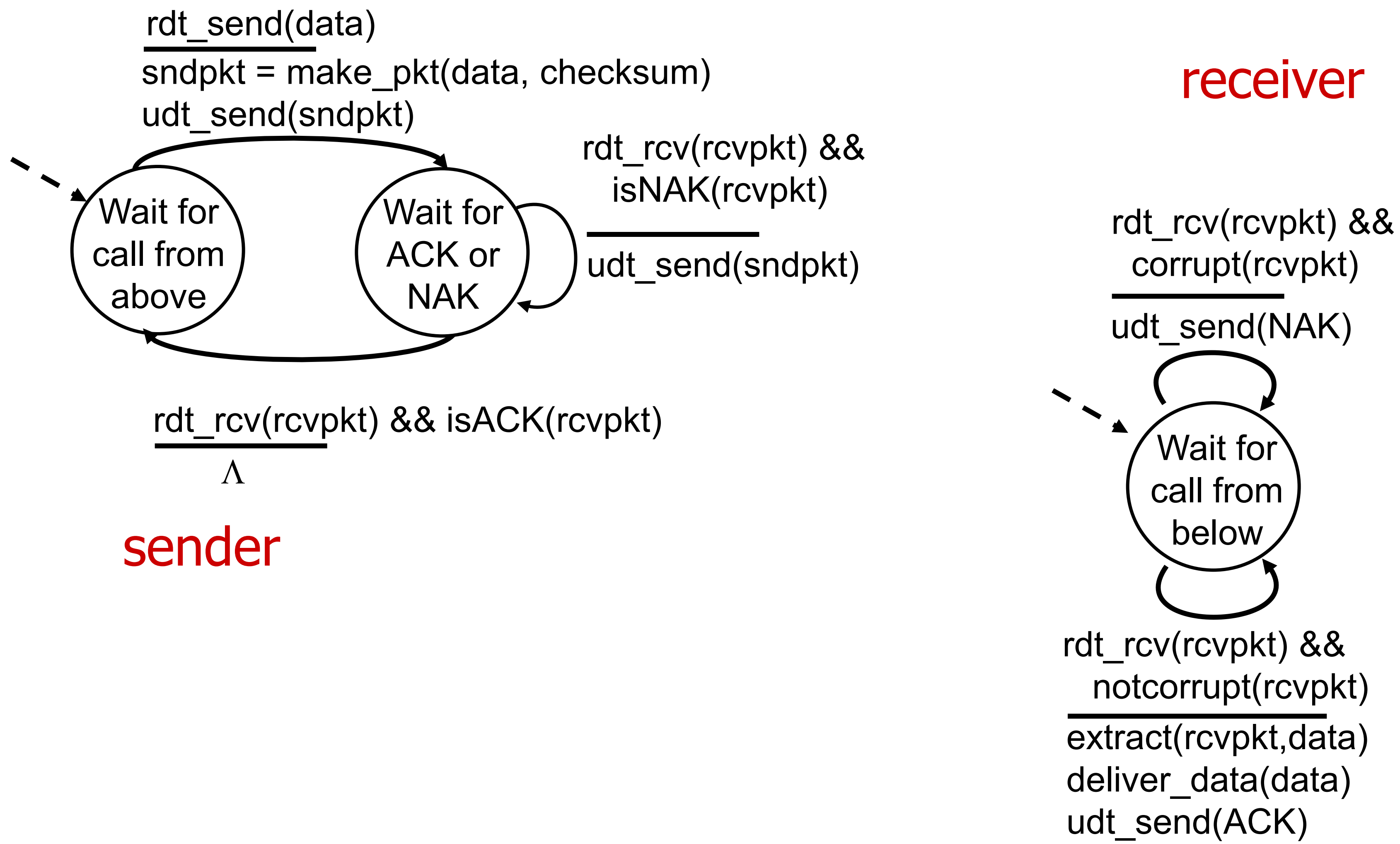
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

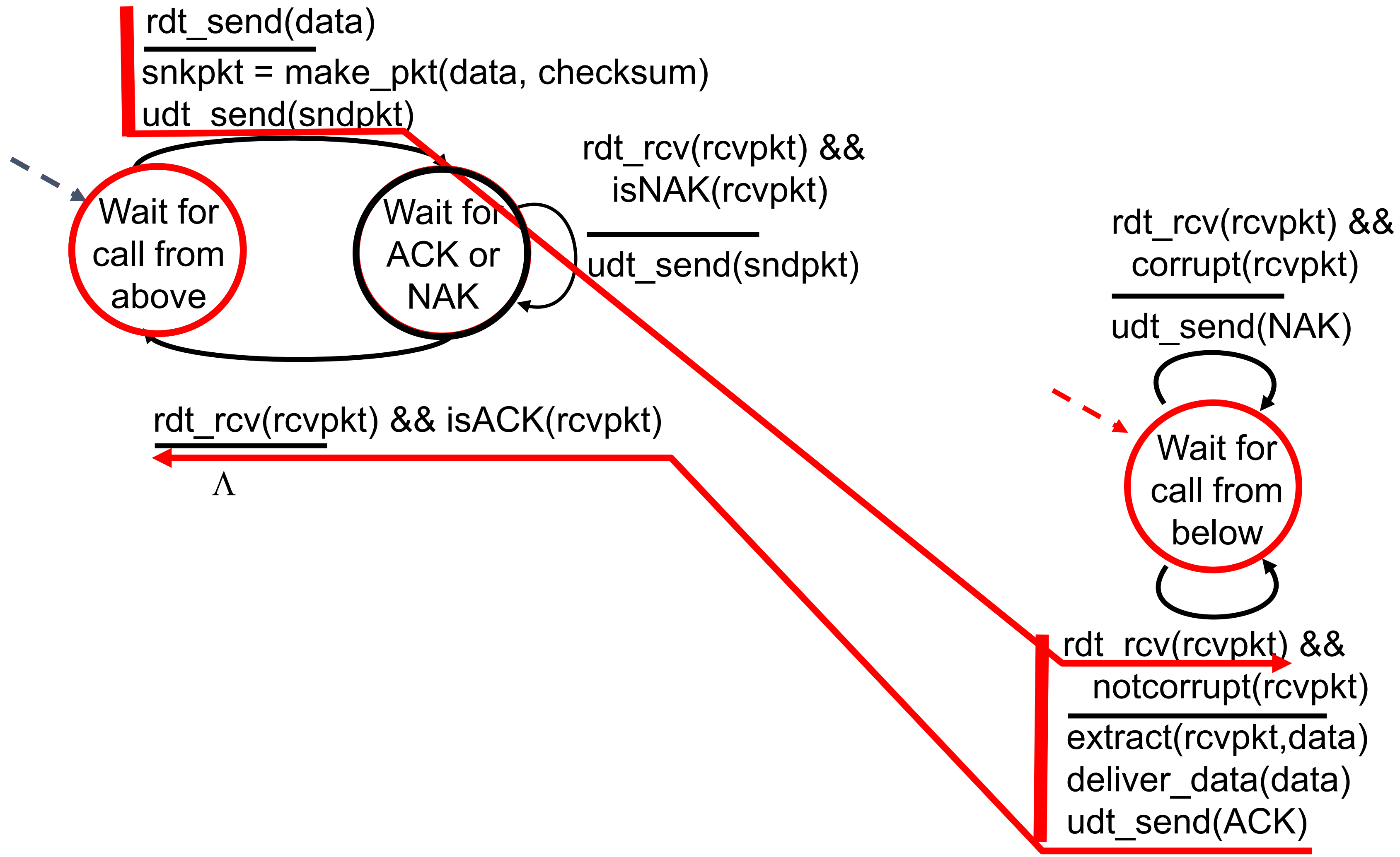
stop and wait

sender sends one packet, then waits for receiver response

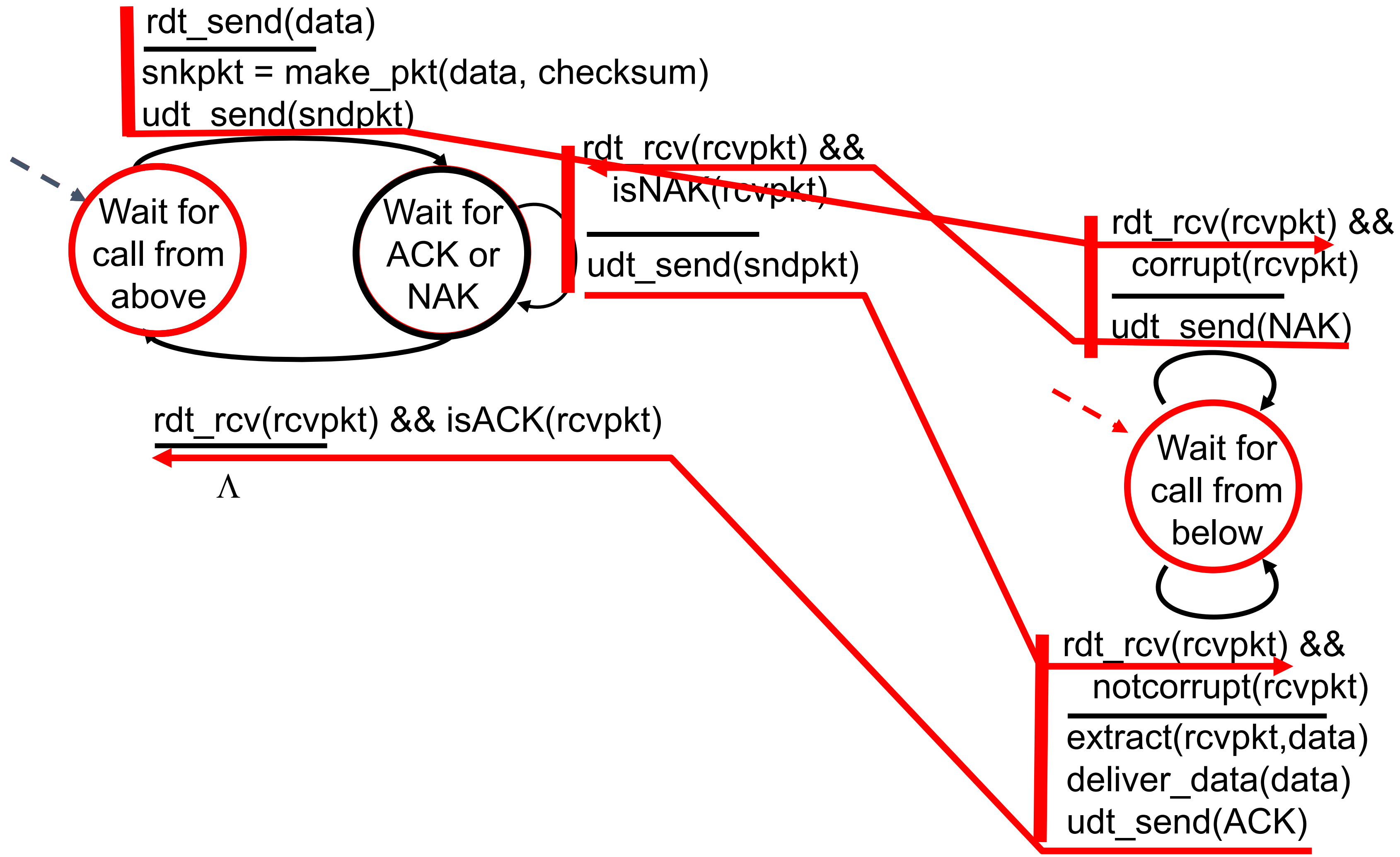
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario.



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

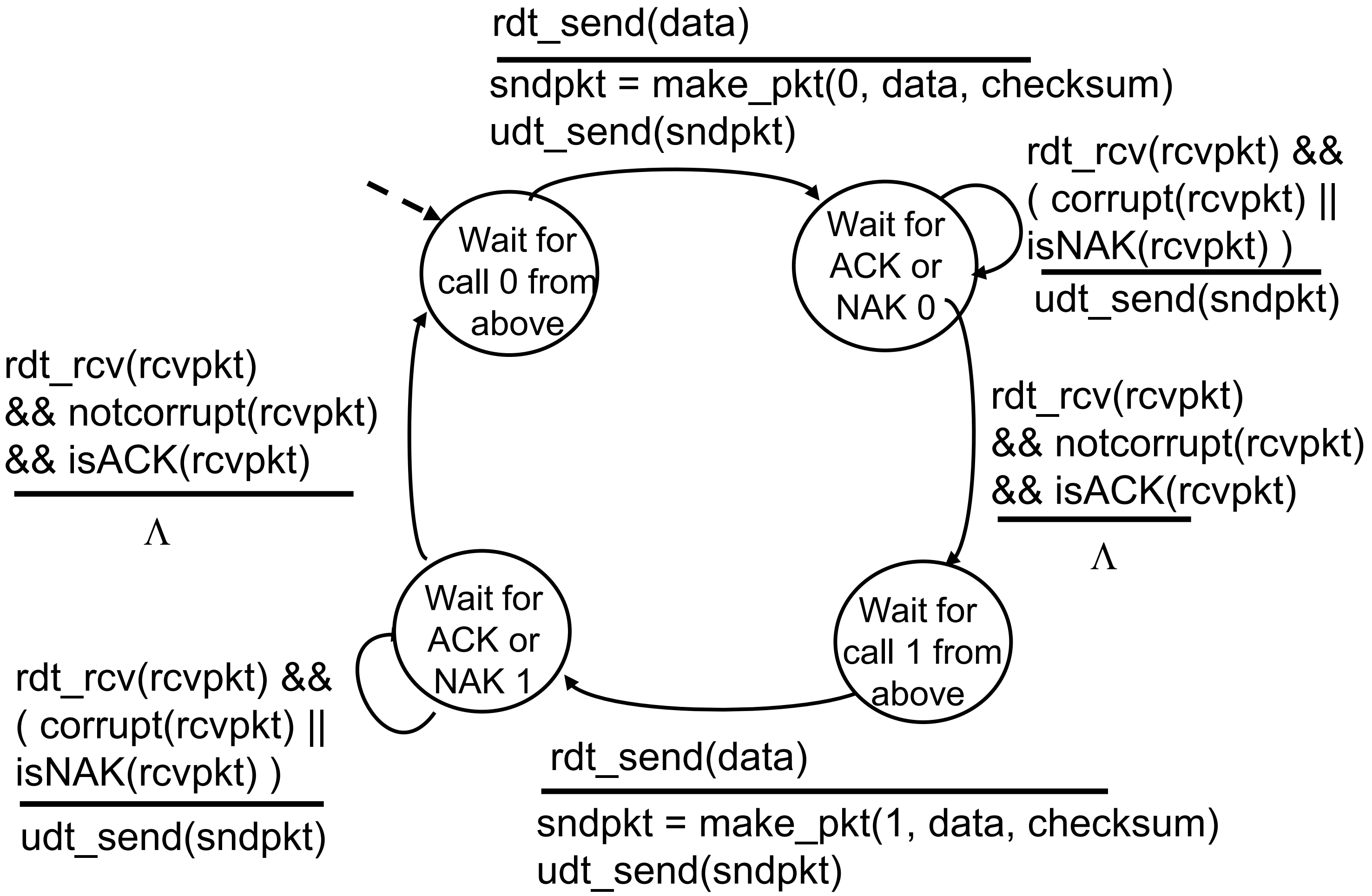
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds sequence number to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

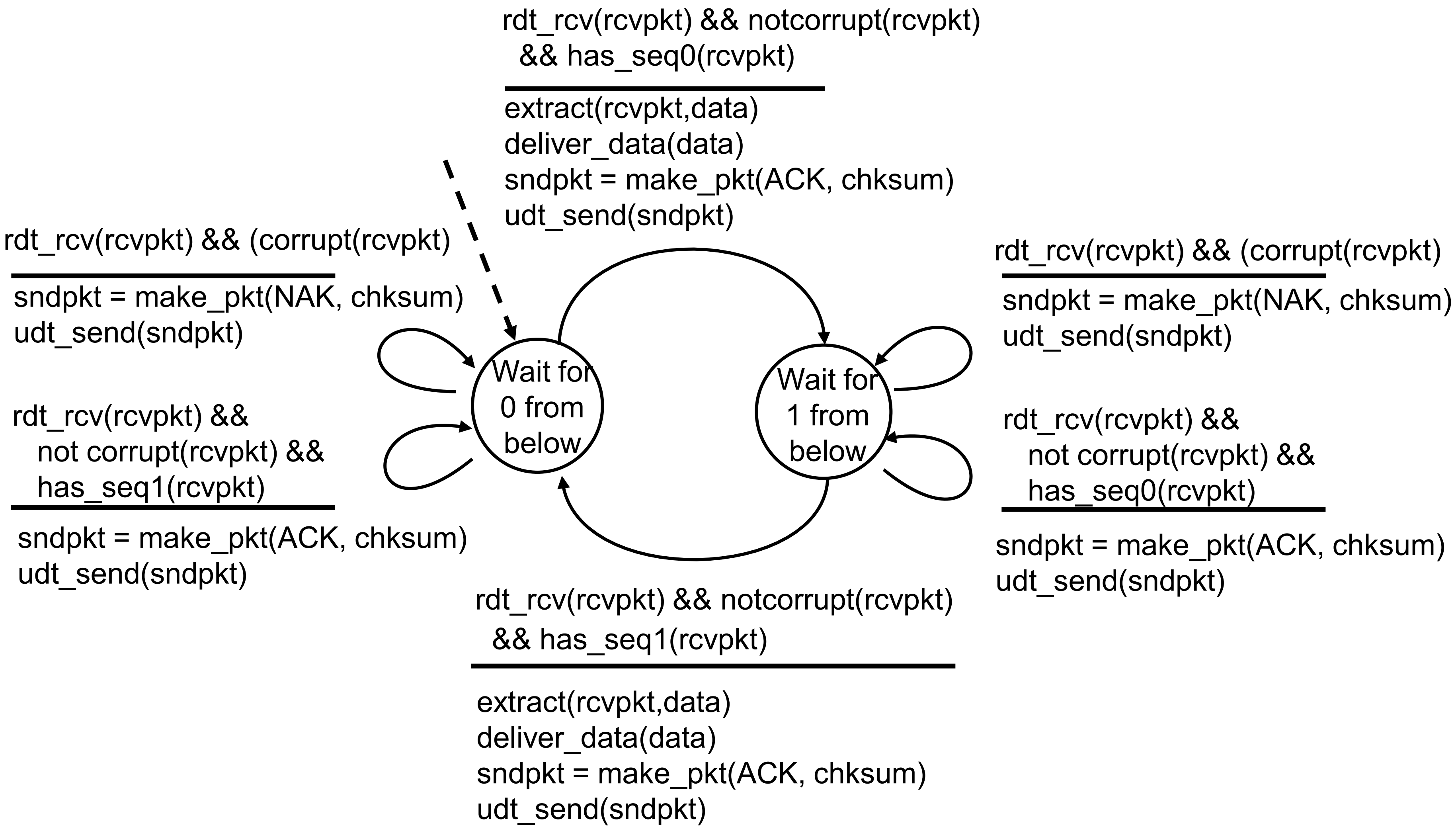
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



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 "expected" pkt should have seq # of 0 or 1

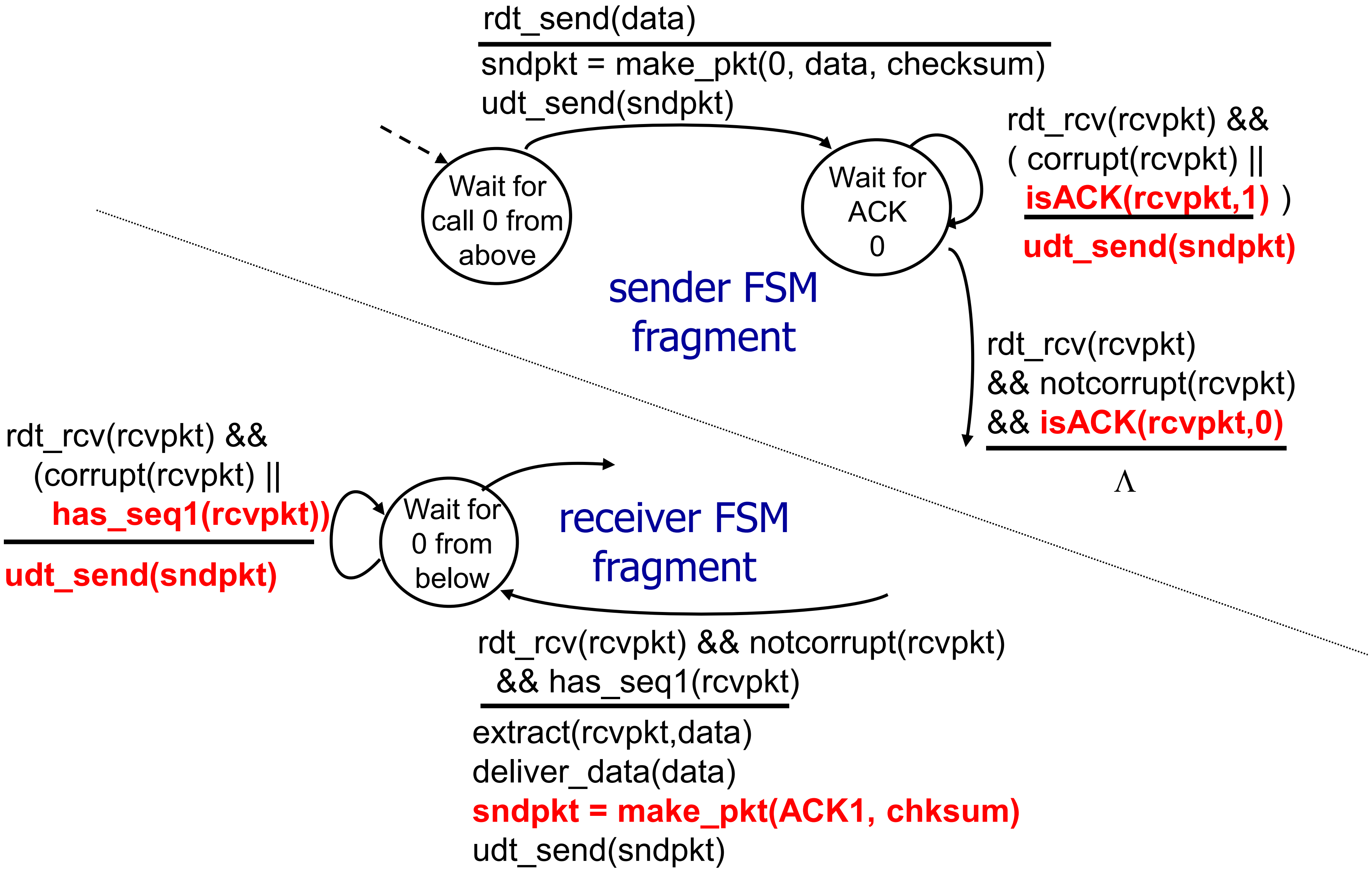
receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

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

rdt2.2: sender, receiver fragments



Summary

THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603

COMPUTER NETWORKS

Transport Layer

Animesh Giri

Department of Computer Science & Engineering

COMPUTER NETWORKS

Principles of reliable date transfer

Animesh Giri

Department of Computer Science & Engineering

In this segment

- rdt3.0: channels with errors and loss
- rdt3.0 sender
- rdt3.0 in action
- Performance of rdt3.0
- rdt3.0: stop-and-wait operation

rdt3.0: channels with errors and loss

new assumption: underlying channel can also lose packets (data, ACKs)

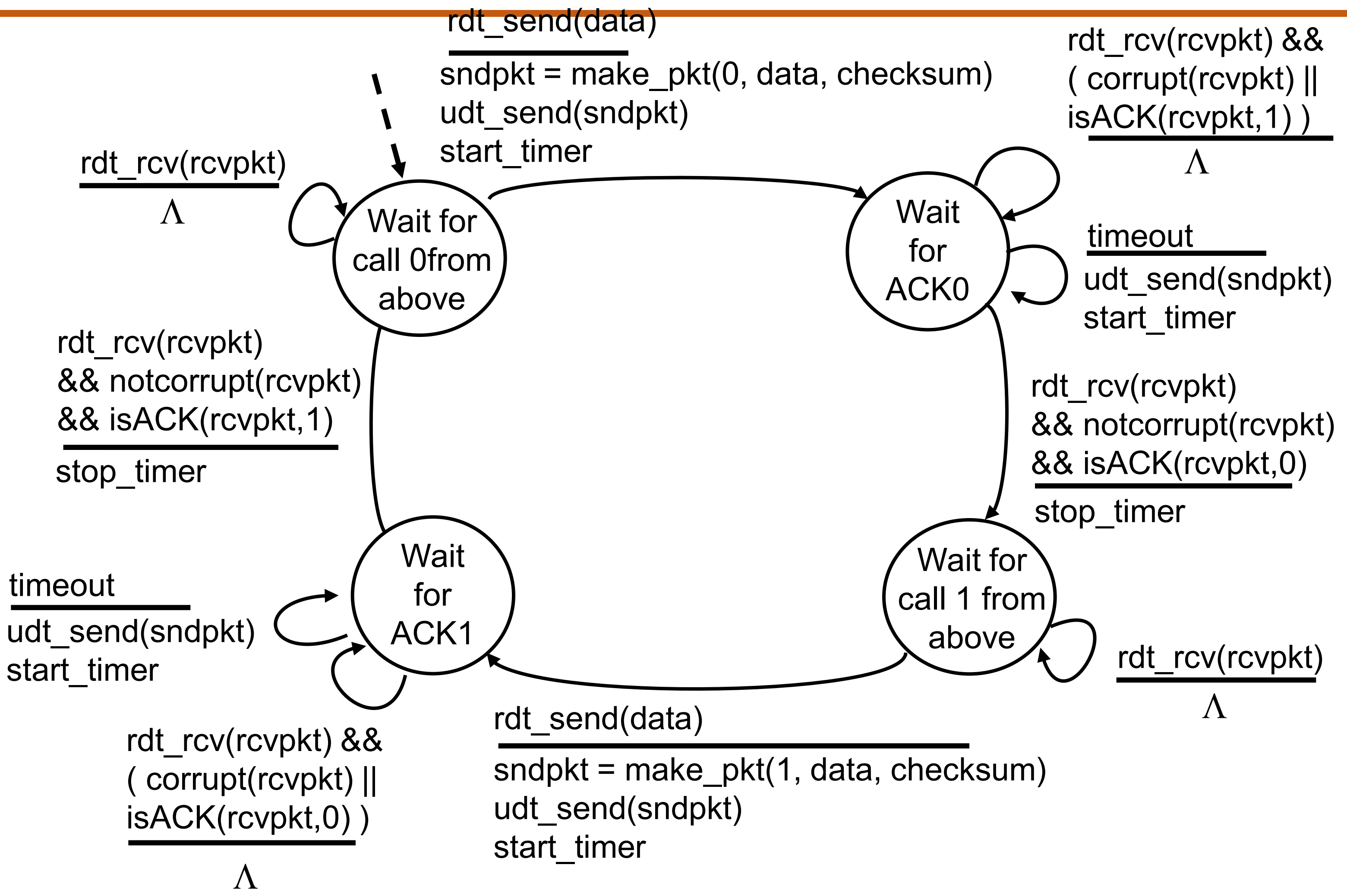
- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

timeout

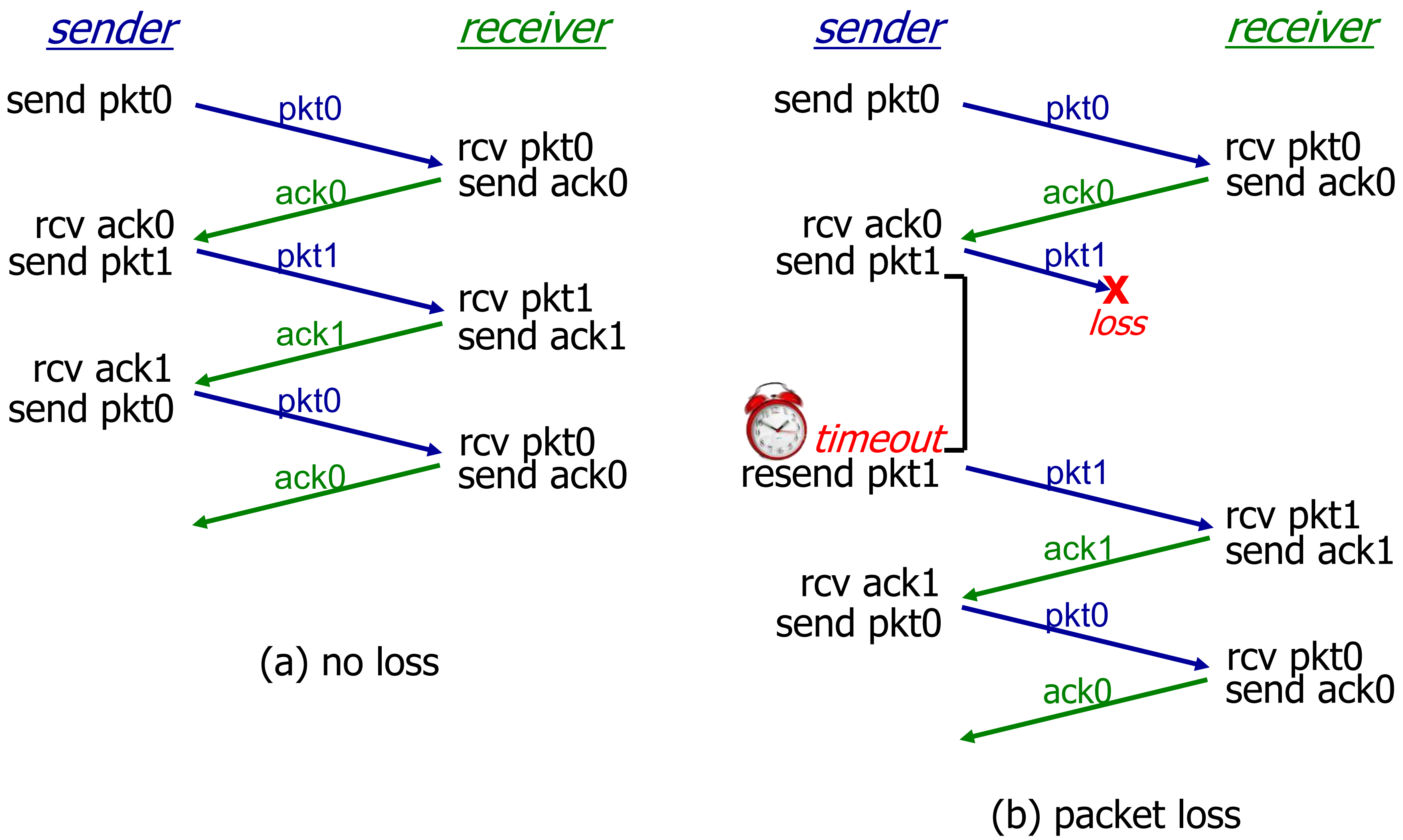
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

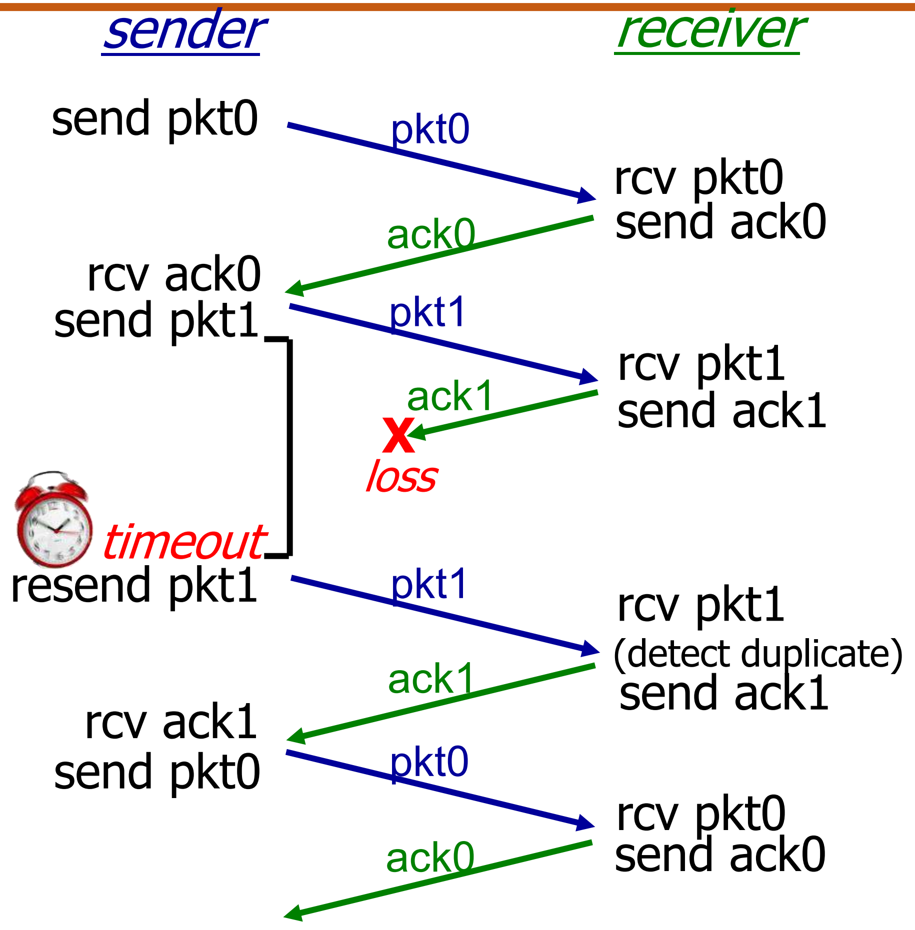
rdt3.0 sender



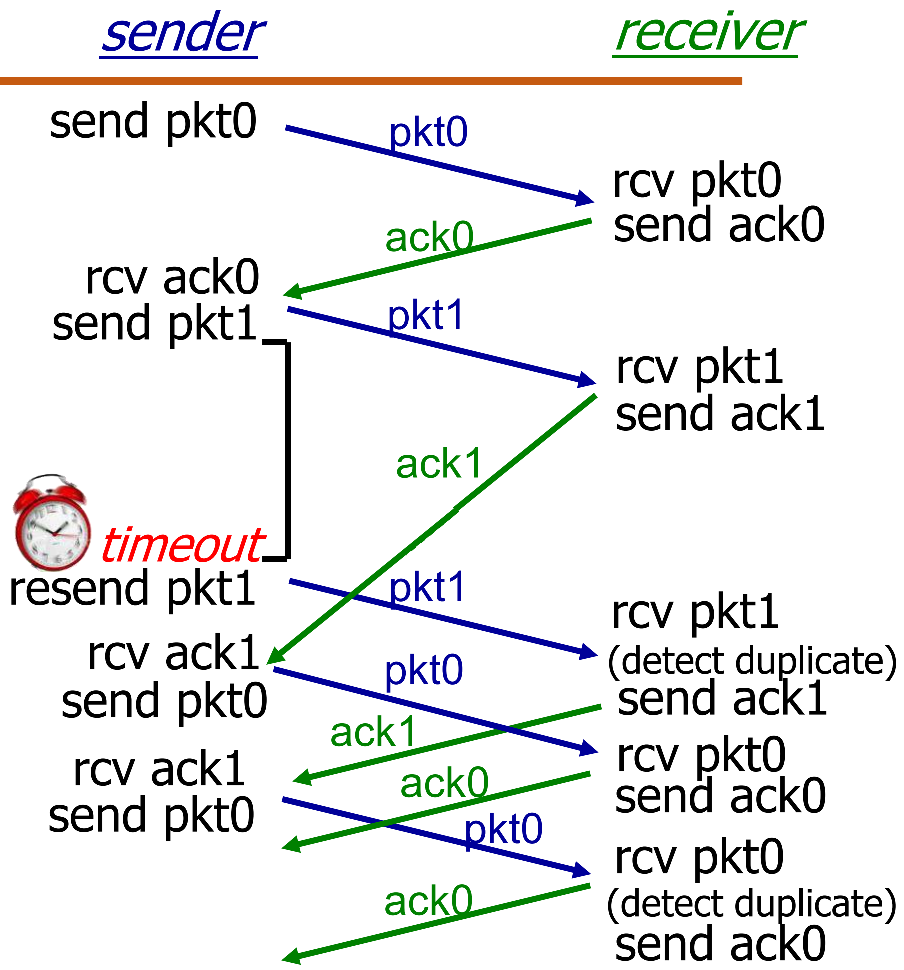
rdt3.0 in action



rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

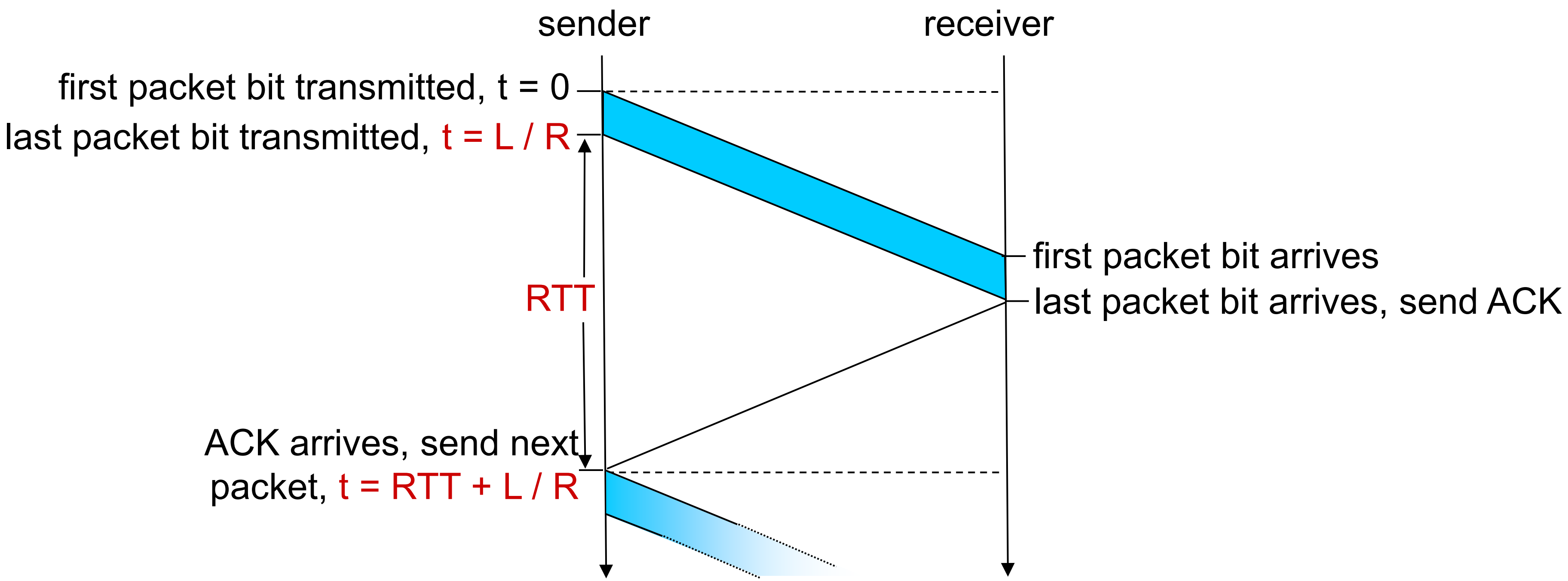
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U_{sender}: *utilization* – fraction of time sender busy sending

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

  - if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation



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

THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603

COMPUTER NETWORKS

Transport Layer

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Pipelined protocols

Animesh Giri

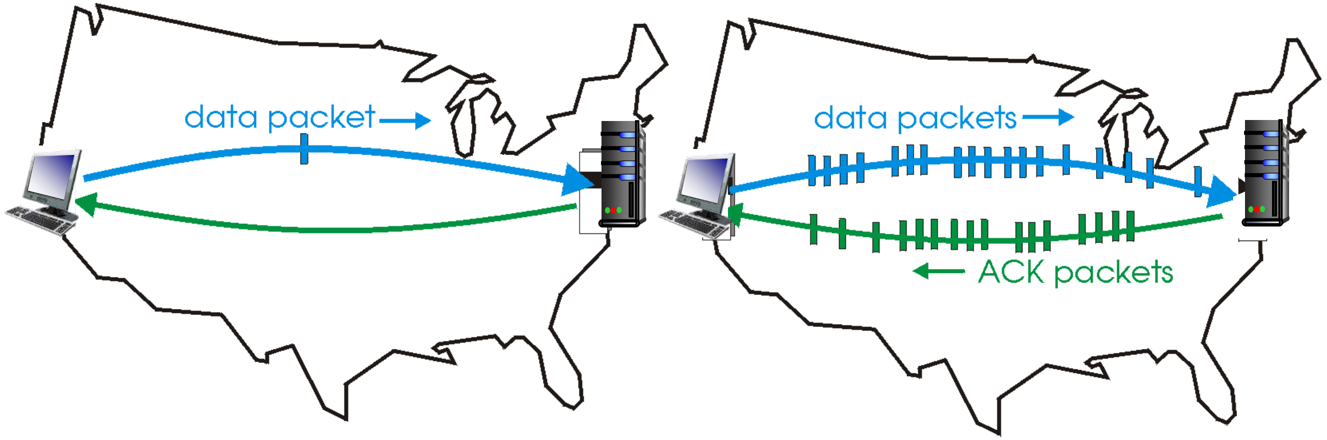
Department of Computer Science & Engineering

In this segment

- Pipelined protocols
- Pipelining: increased utilization
- Pipelined protocols: overview
- Go-Back-N: sender
- GBN: sender extended FSM
- GBN: receiver extended FSM
- Selective repeat
- Selective repeat: sender, receiver windows
- Selective repeat in action
- Selective repeat: dilemma

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

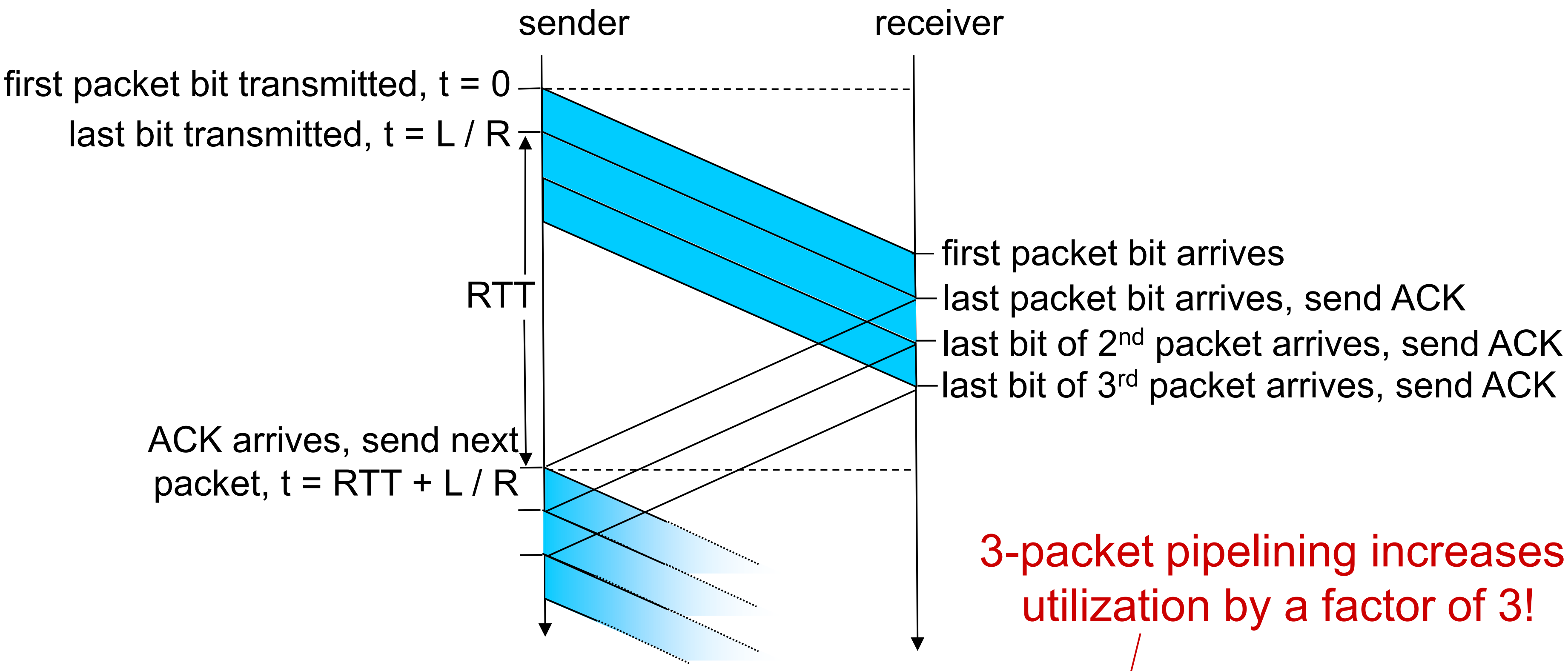


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: increased utilization



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:

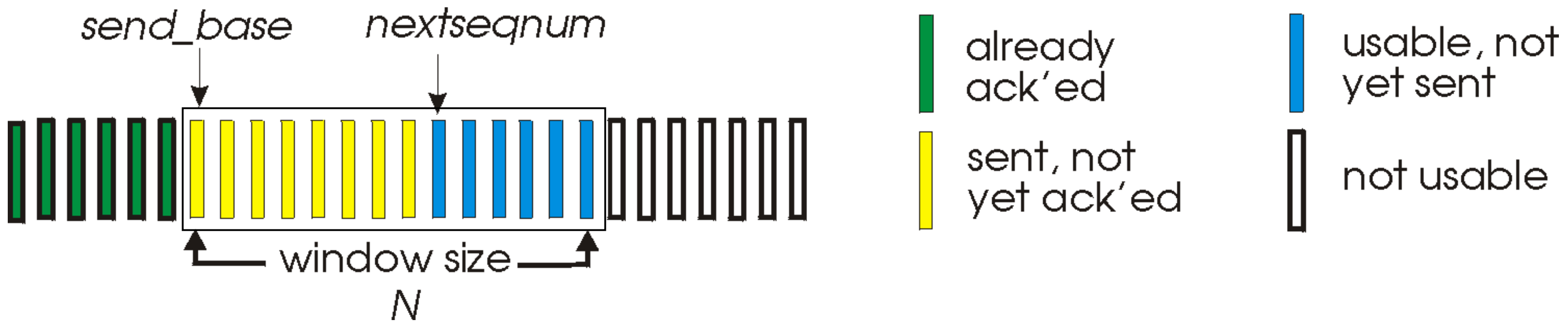
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Go-Back-N: sender

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

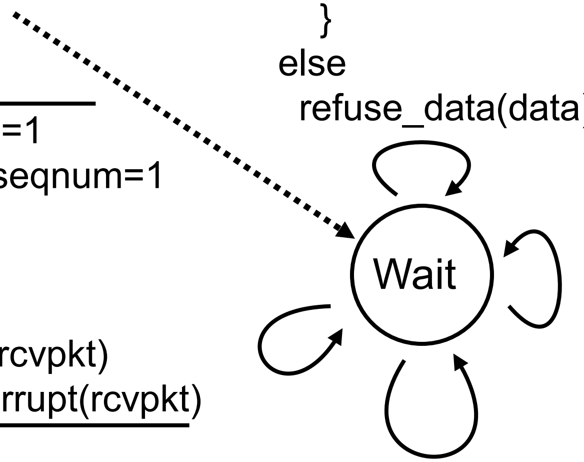


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window

GBN: 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
```

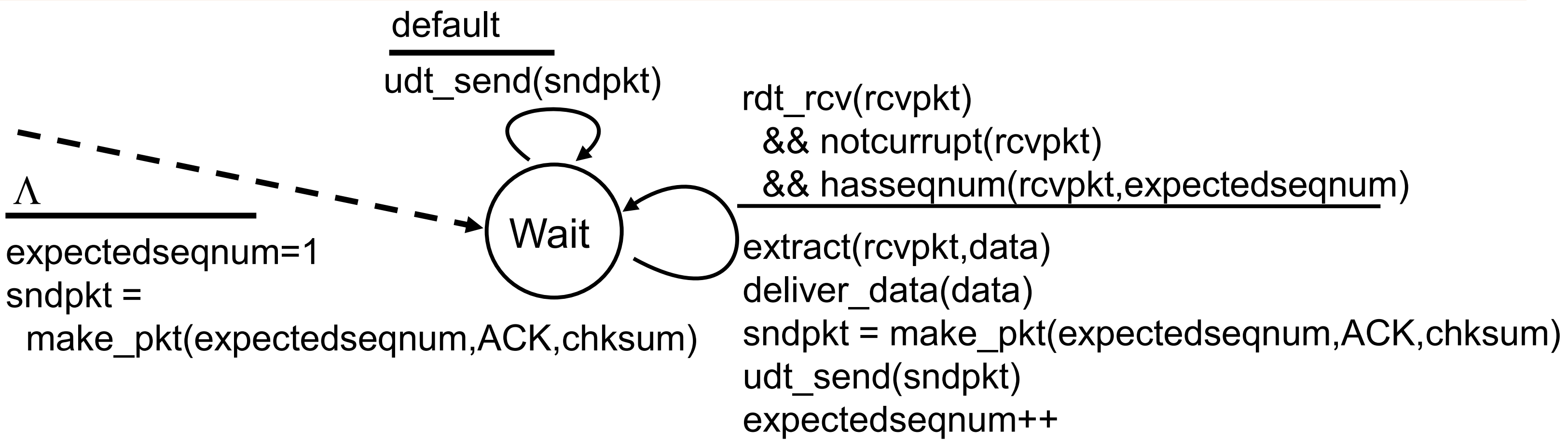



```
timeout
_________________________
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

```
rdt_rcv(rcvpkt)
   && corrupt(rcvpkt)
_________________________
```

```
rdt_rcv(rcvpkt) &&
   notcorrupt(rcvpkt)
__________________________
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

GBN: receiver extended FSM

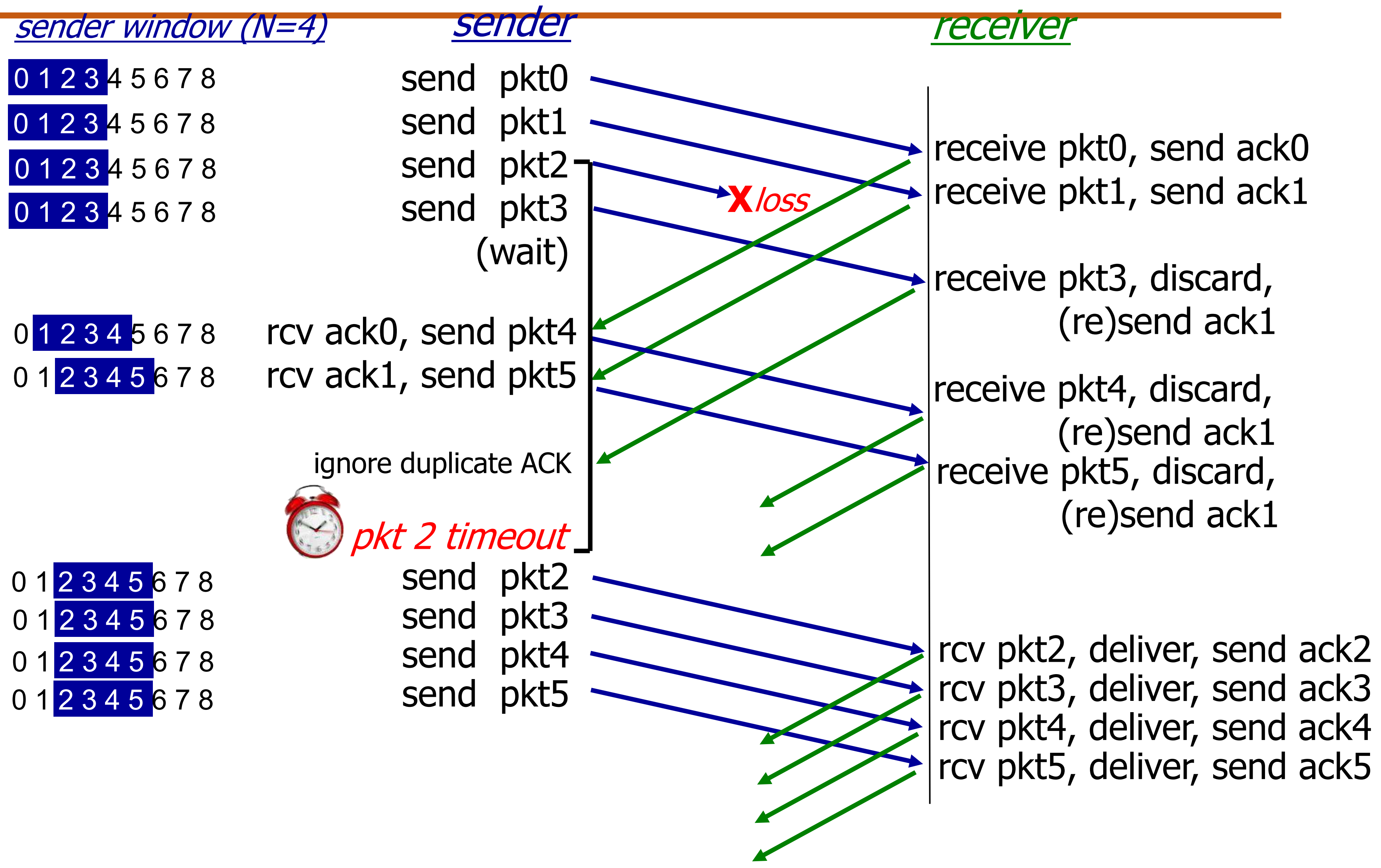


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #

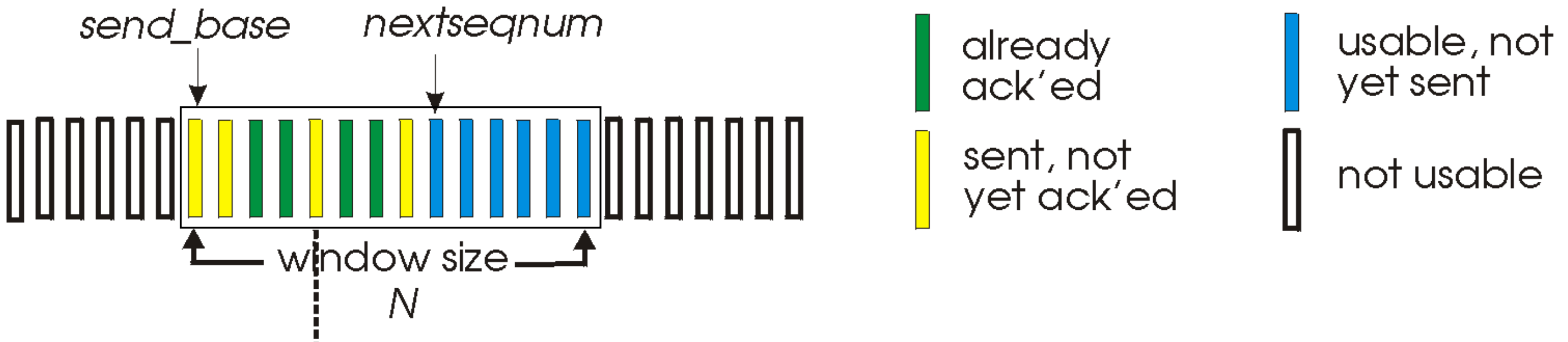
GBN in action



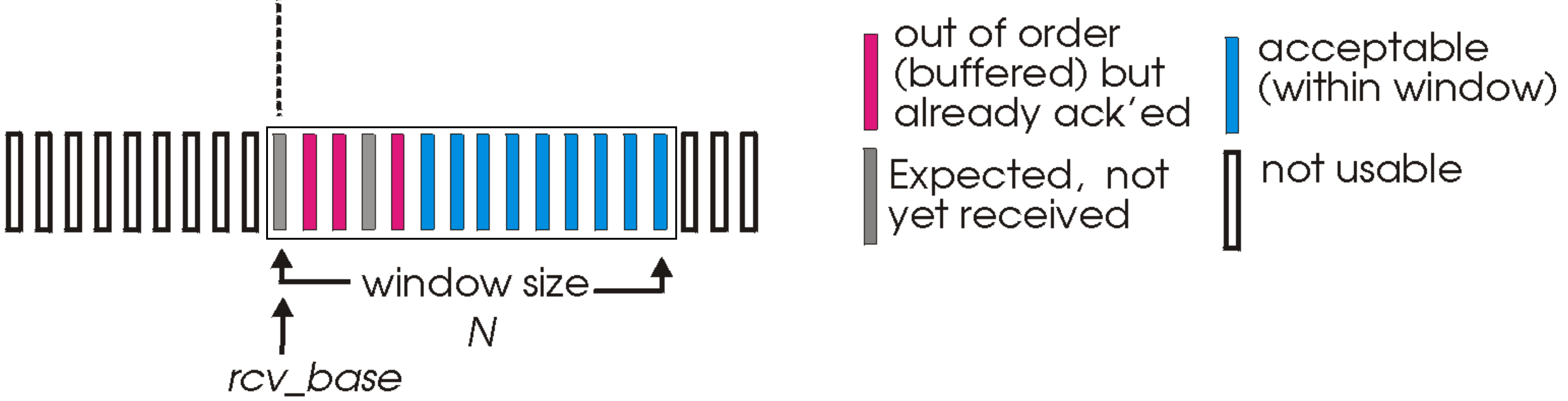
Selective repeat

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

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

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

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

receiver

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

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

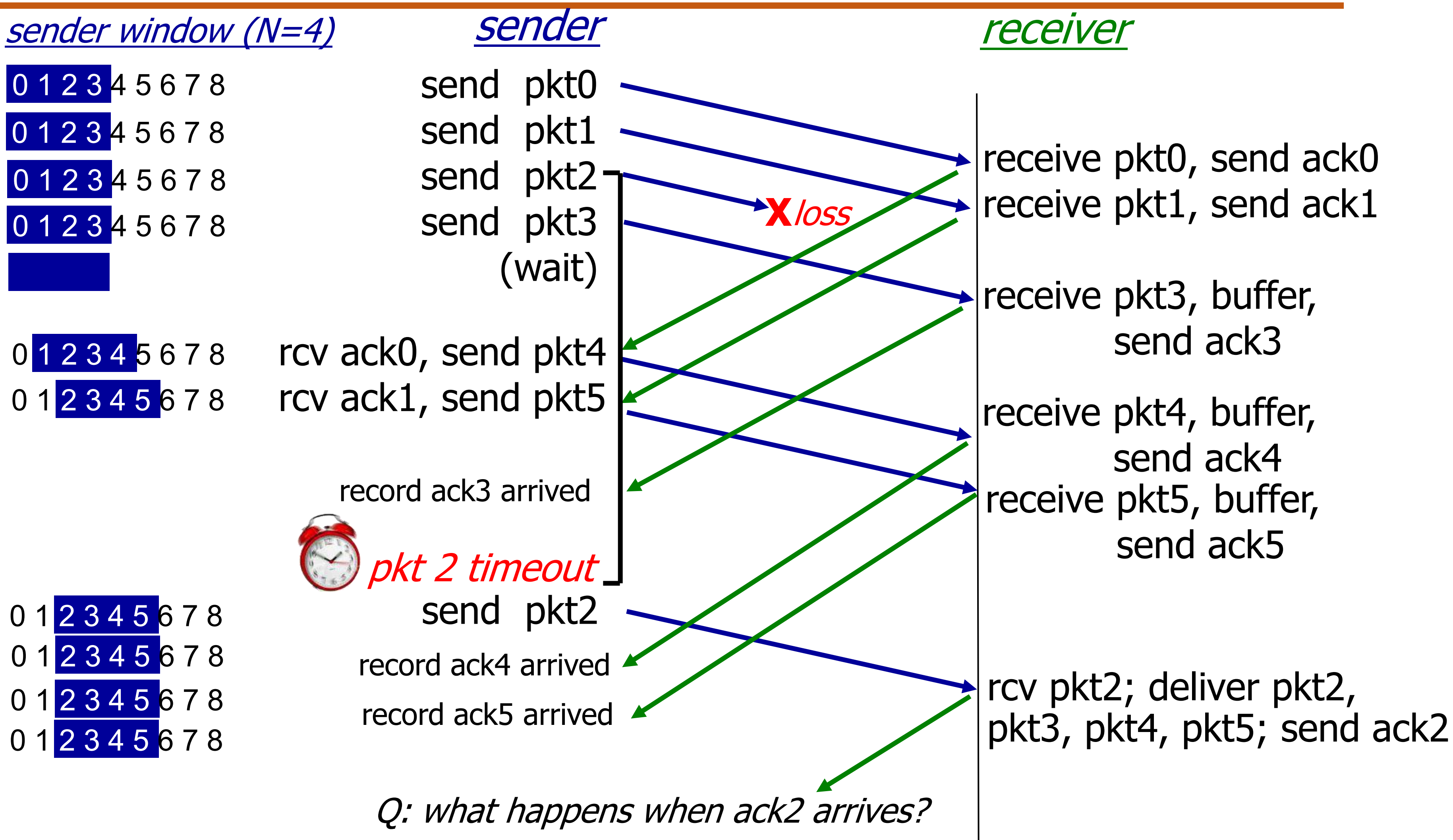
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

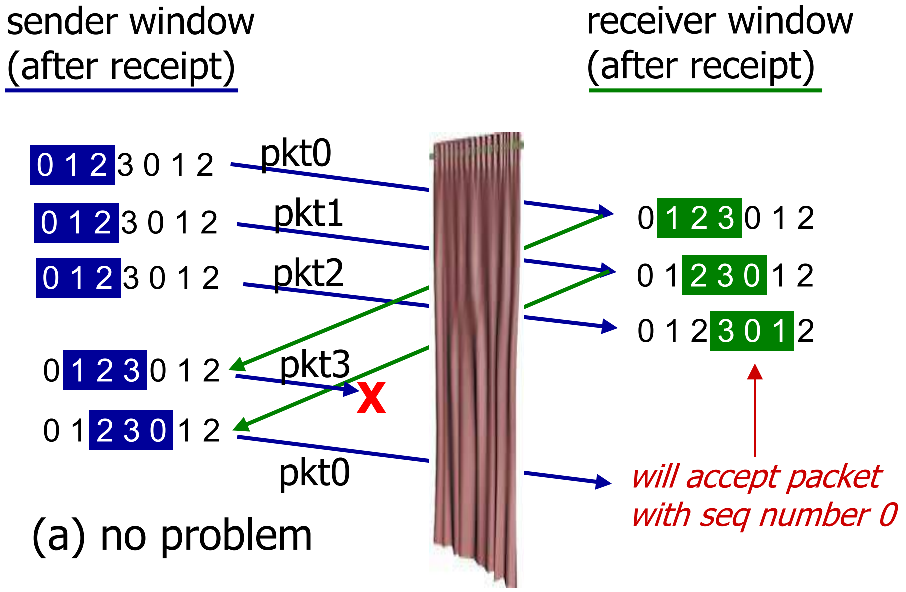
Selective repeat in action



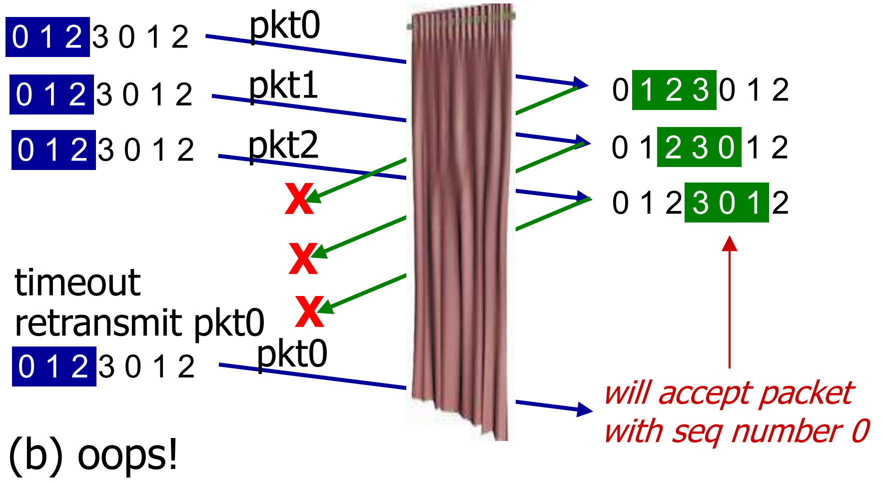
Selective repeat: dilemma

example:

- seq #'s: 0, 1, 2, 3
- window size=3
  - receiver sees no difference in two scenarios!
  - duplicate data accepted as new in (b)



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



Q: what relationship between seq # size and window size to avoid problem in (b)?

THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 66186603

COMPUTER NETWORKS

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri

Department of Computer Science & Engineering

COMPUTER NETWORKS

Connection-oriented transport: TCP

Animesh Giri

Department of Computer Science & Engineering

In this segment

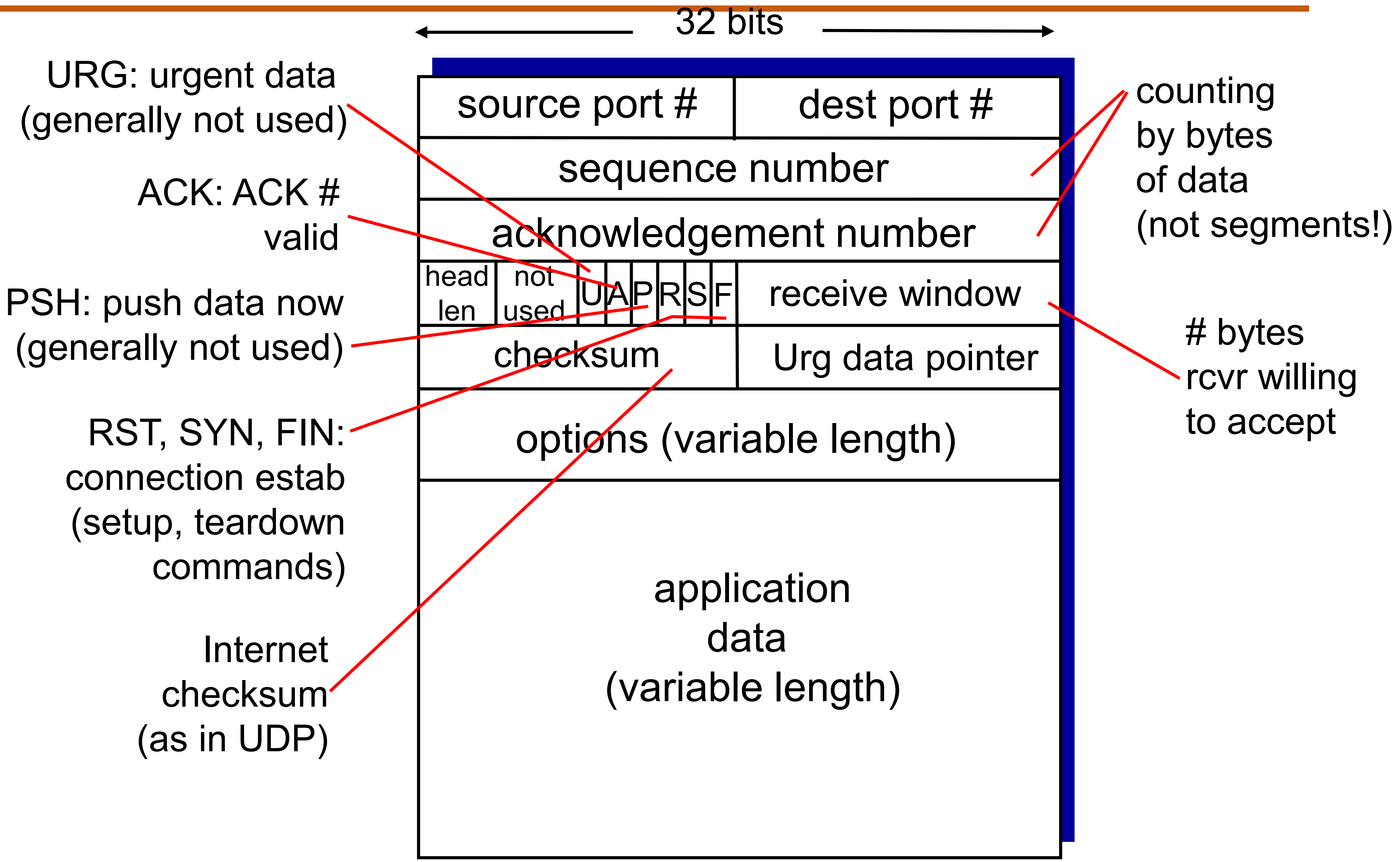
- TCP: Overview RFCs: 793,1122,1323, 2018, 2581
- TCP segment structure
- TCP seq. numbers, ACKs
- TCP round trip time, timeout

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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

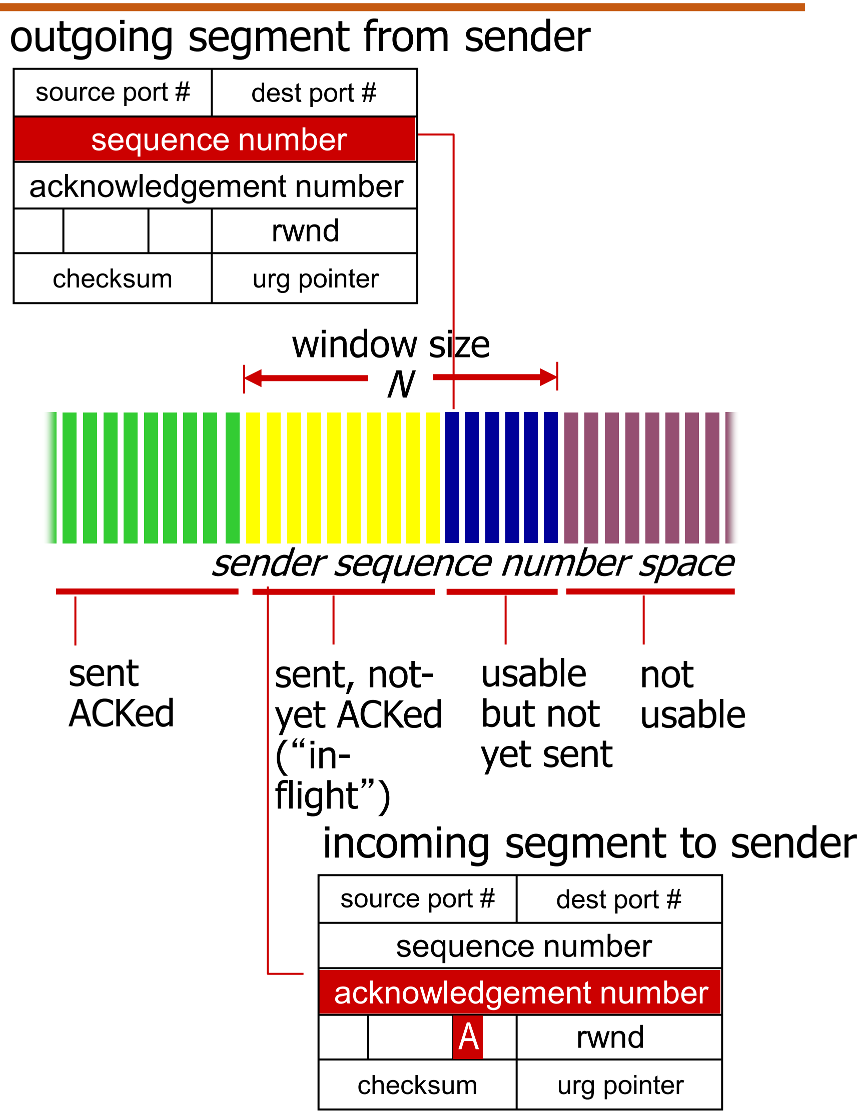
- byte stream "number" of first byte in segment's data

acknowledgements:

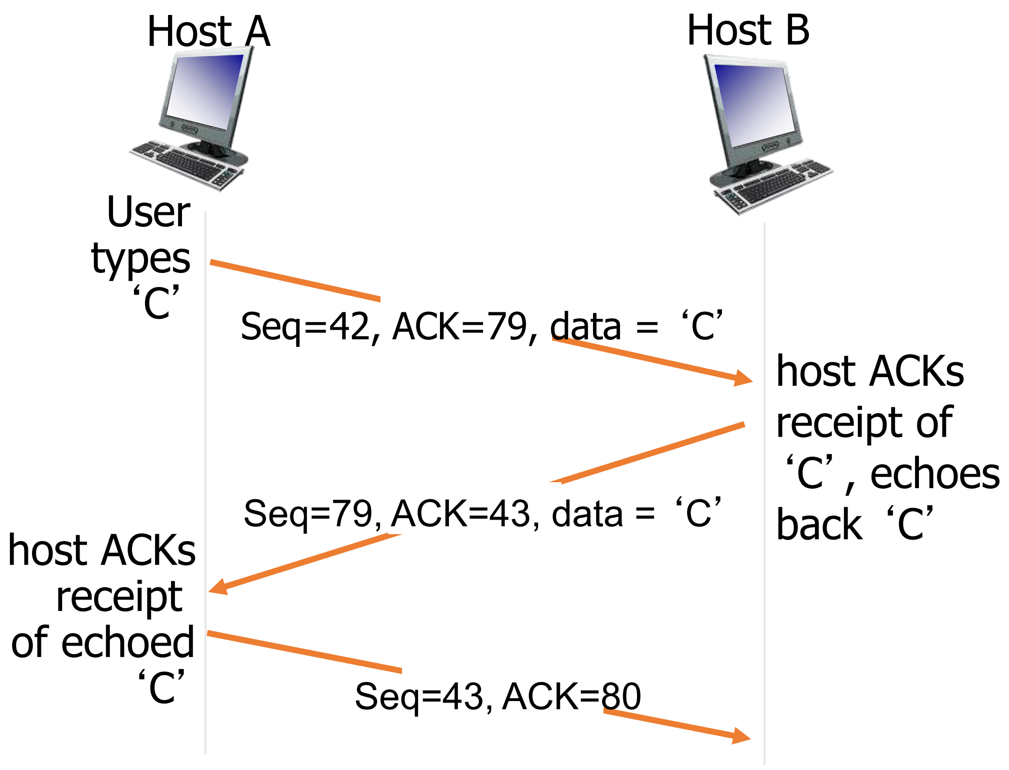
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



TCP seq. numbers, ACKs



simple telnet scenario

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout, unnecessary retransmissions
- too long: slow reaction to segment loss

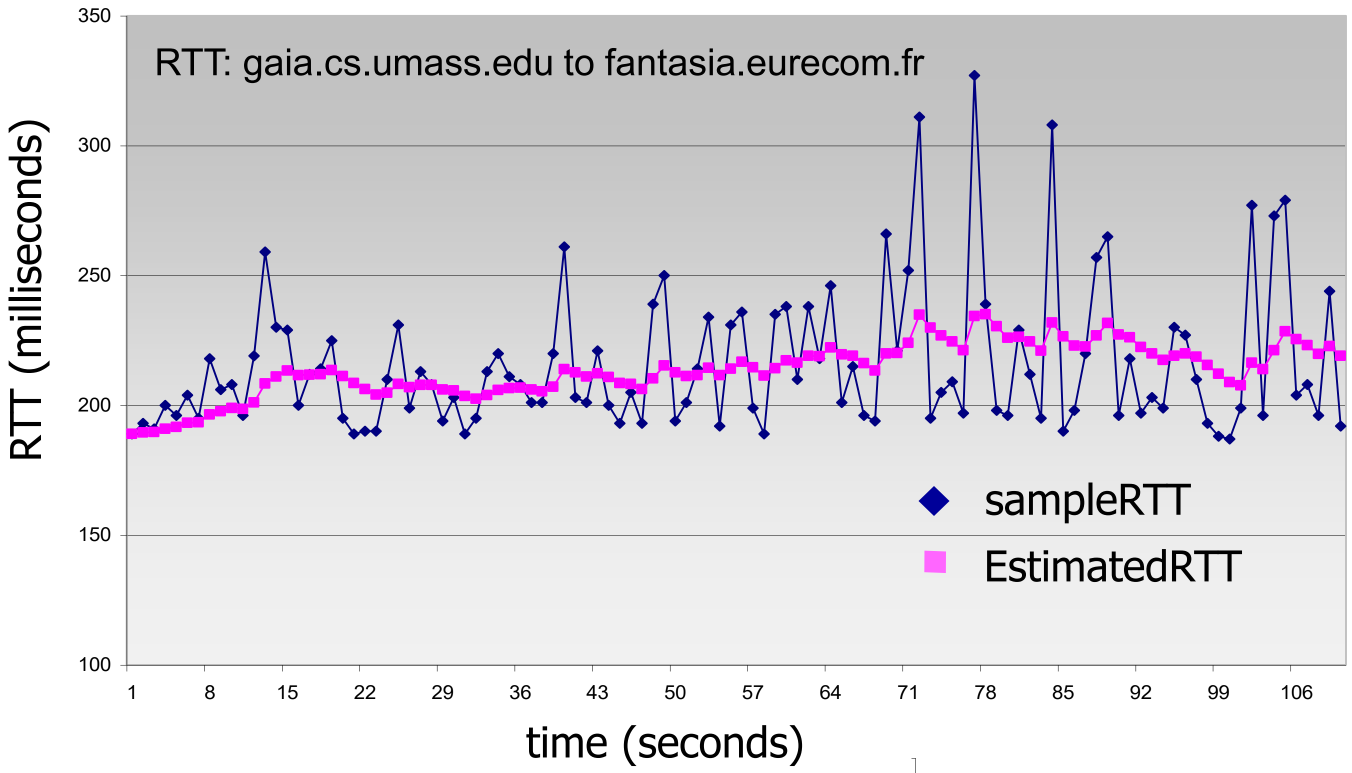
Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT **"smoother"**
  - average several recent measurements, not just current `SampleRTT`

TCP round trip time, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: EstimatedRTT plus "safety margin"
  - large variation in EstimatedRTT -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|
              (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603

COMPUTER NETWORKS

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Connection-oriented transport: TCP
Reliable data transfer

Animesh Giri
Department of Computer Science & Engineering

In this segment

- TCP reliable data transfer
- TCP sender events
- TCP sender (simplified)
- TCP: retransmission scenarios
- TCP ACK generation [RFC 1122, RFC 2581]
- TCP fast retransmit
- Summary

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 initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events

data rcvd from app:

- create 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 still unacked segments

TCP sender (simplified)

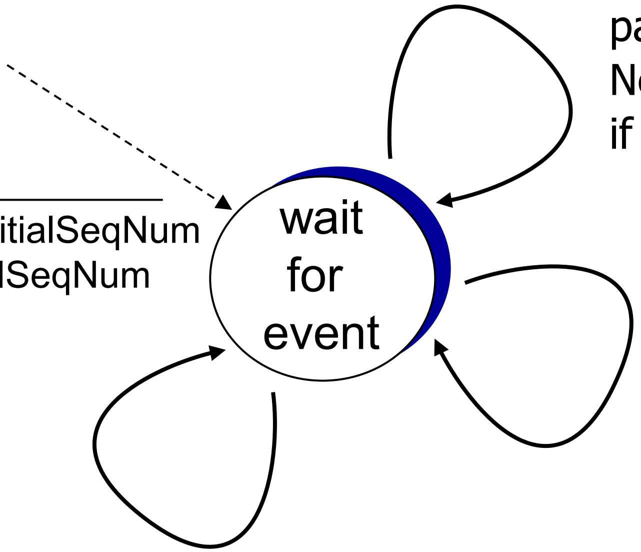
data received from application above

---

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

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



timeout

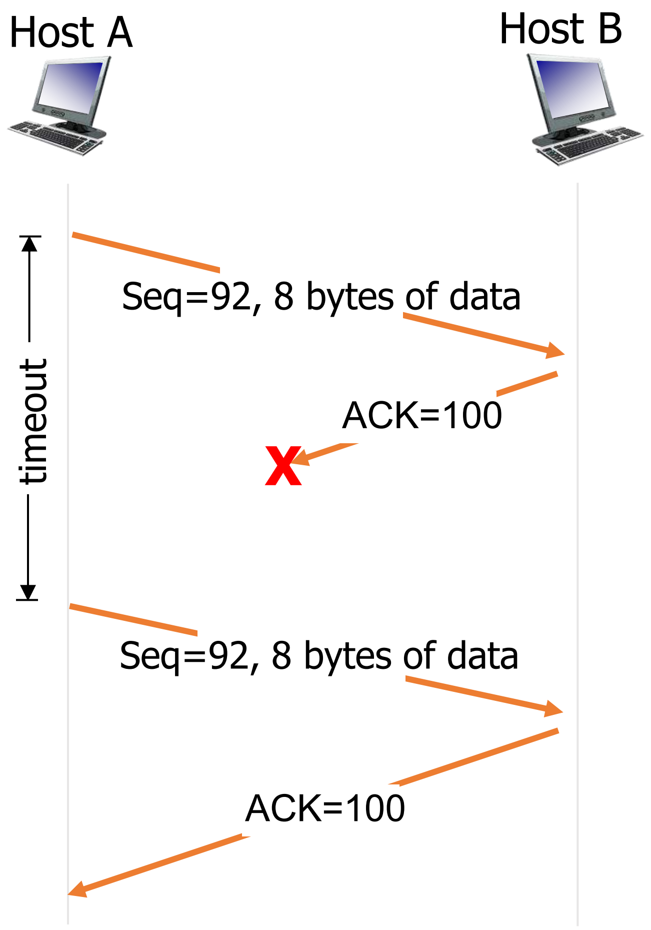
---

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

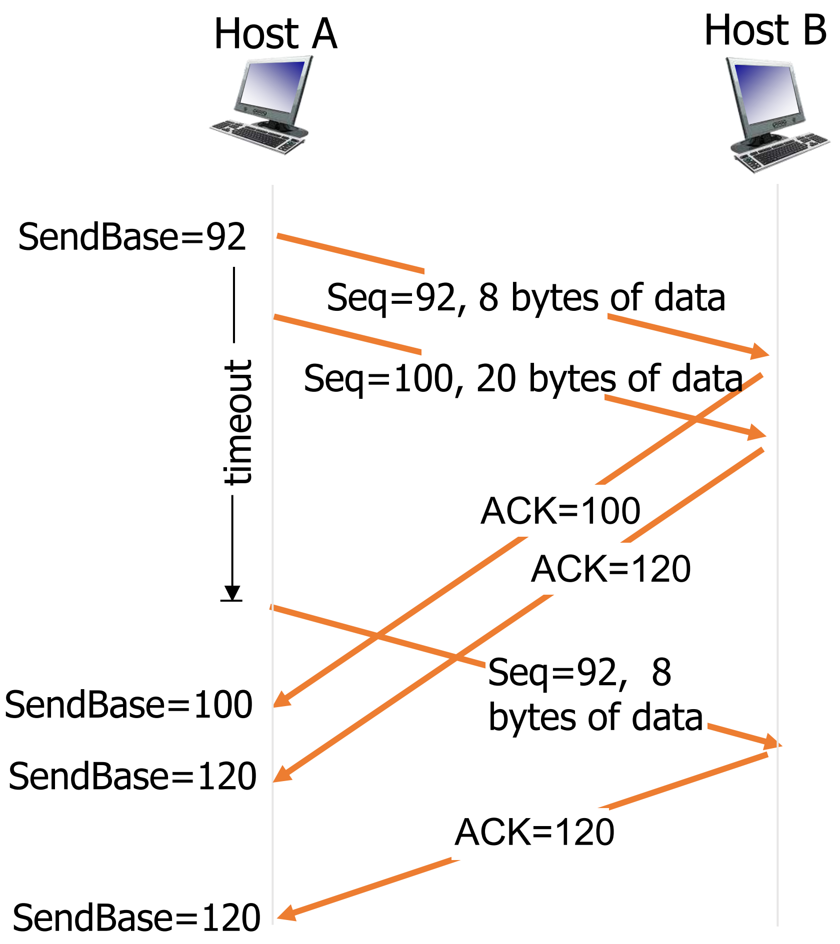
ACK received, with ACK field value y

```
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

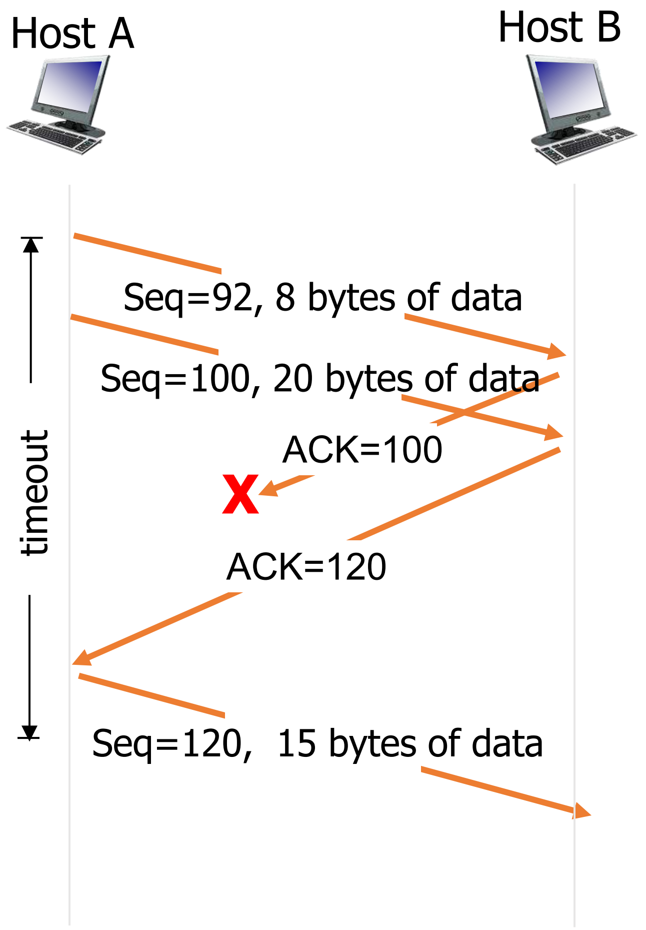


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP ACK generation [RFC 1122, RFC 2581]

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 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *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

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.

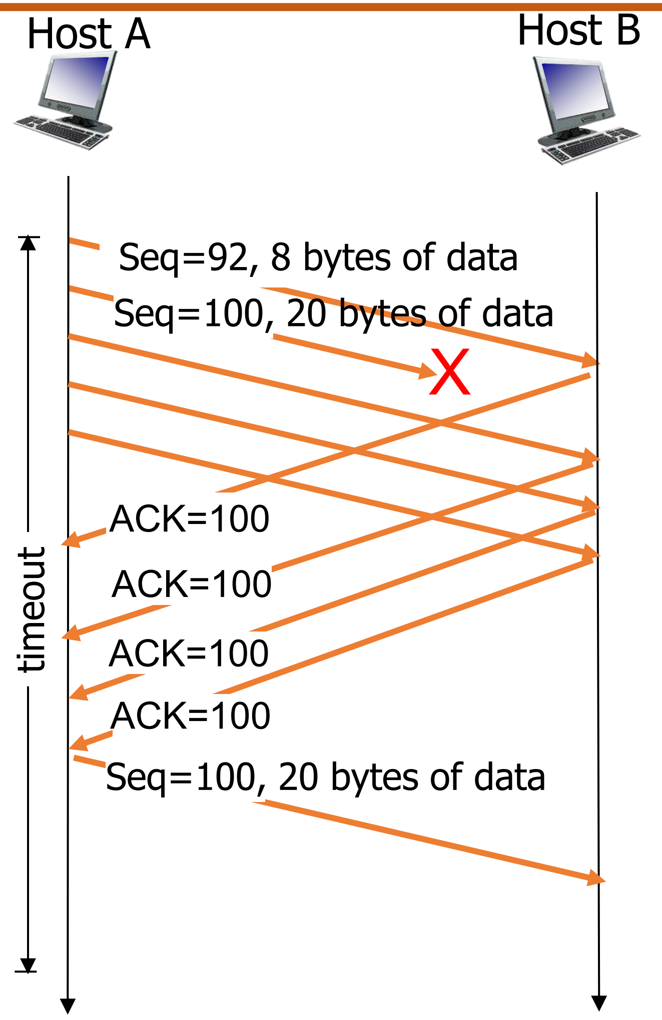
TCP fast retransmit

if sender receives 3 ACKs for same data

("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Summary

THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603

COMPUTER NETWORKS

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Connection-oriented transport: TCP
Flow control

Animesh Giri
Department of Computer Science & Engineering

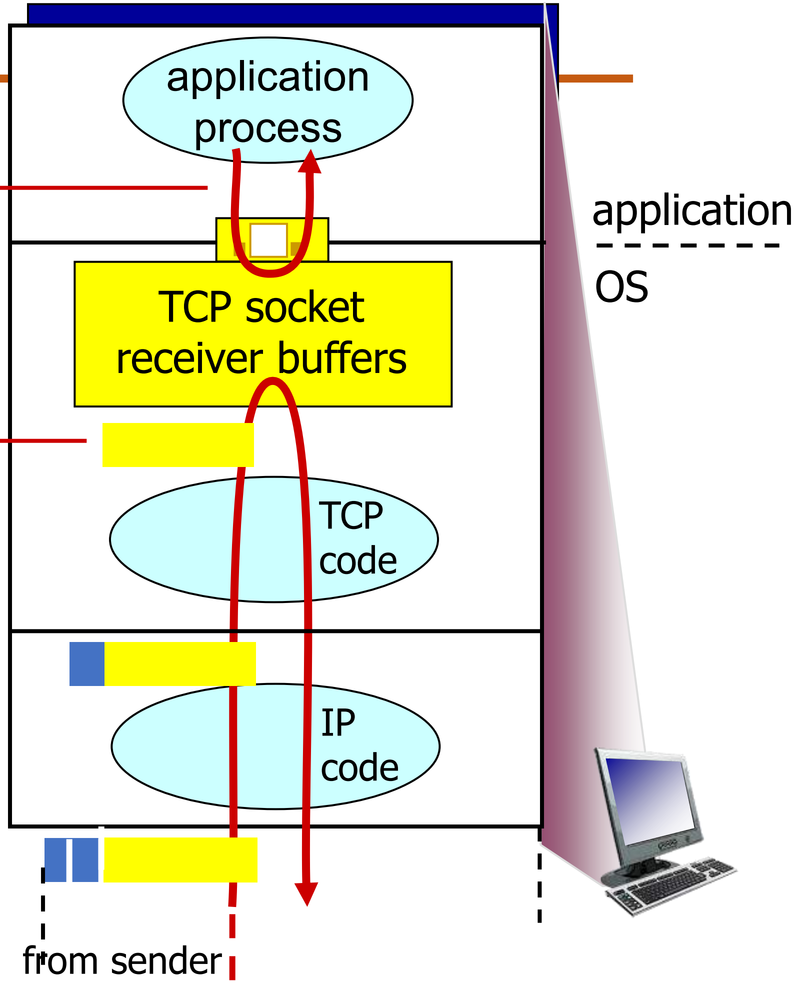
In this segment

- TCP flow control

TCP flow control

application may remove data from TCP socket buffers ….

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



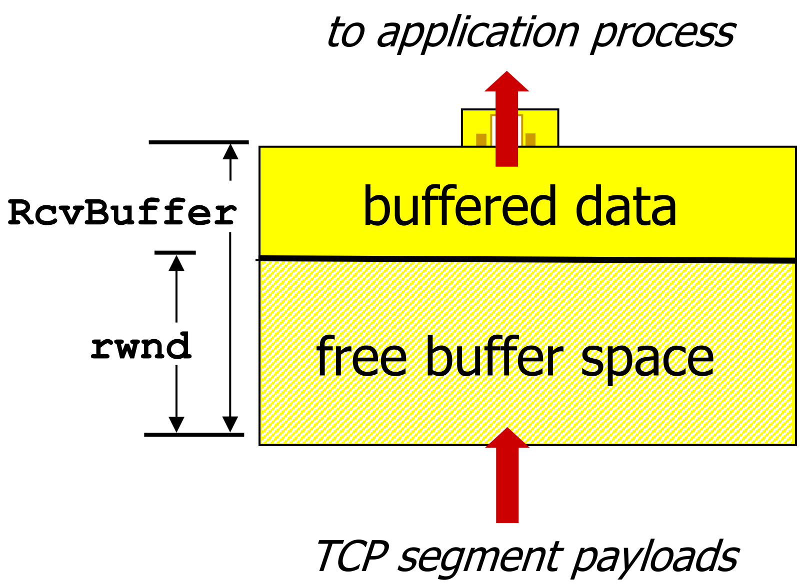
receiver protocol stack

flow control

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

TCP flow control

- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



receiver-side buffering

COMPUTER NETWORKS

Connection-oriented transport: TCP Connection Management

Animesh Giri

Assistant Professor, Department of Computer Science & Engineering

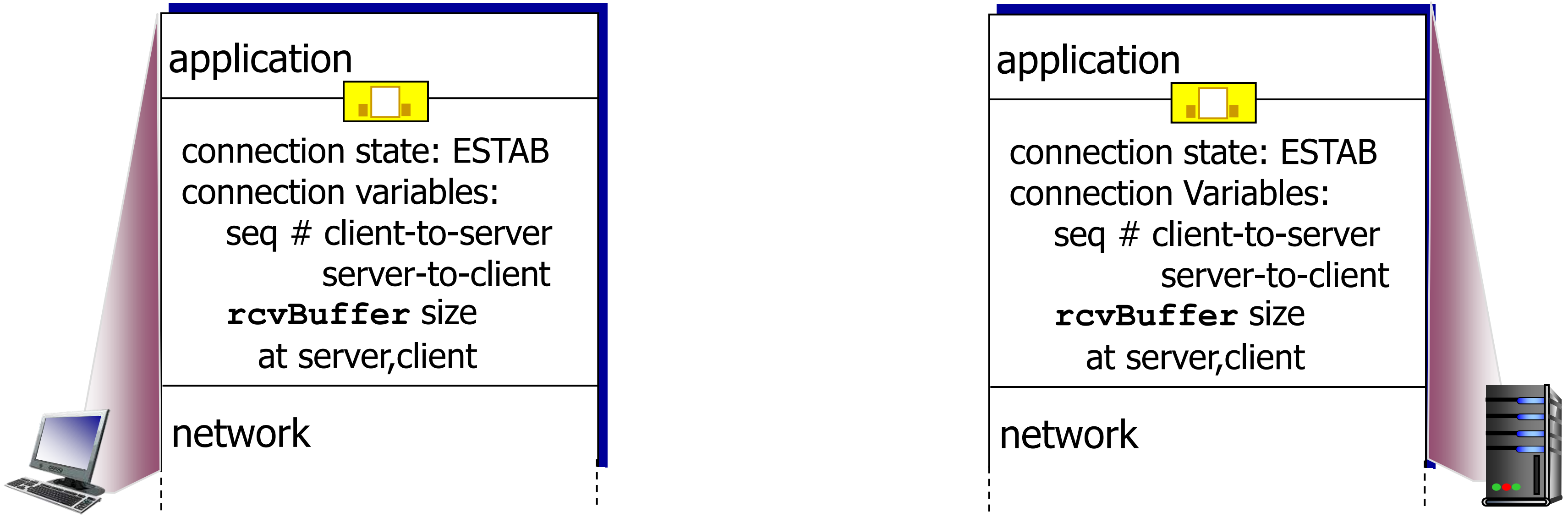
In this segment

- Connection Management
- Agreeing to establish a connection
- TCP 3-way handshake
- TCP 3-way handshake: FSM
- TCP: closing a connection

Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

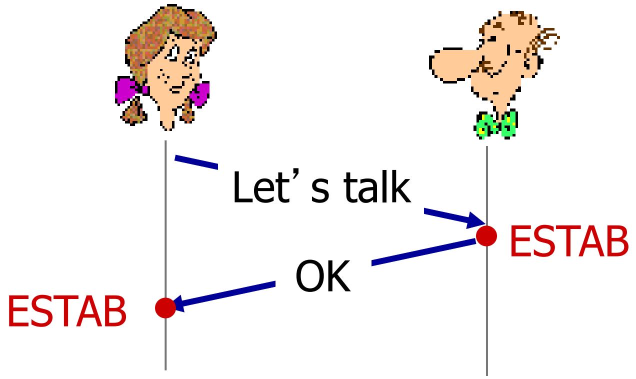


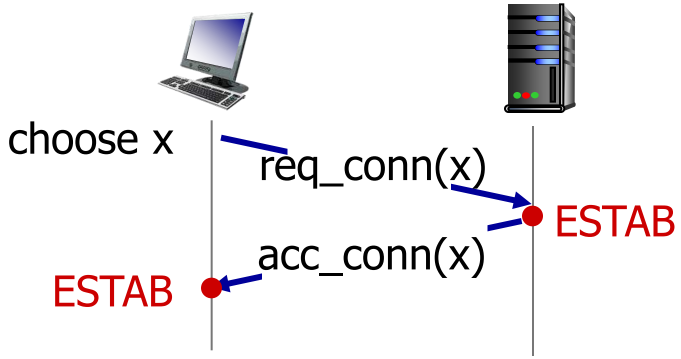

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Agreeing to establish a connection

2-way handshake:





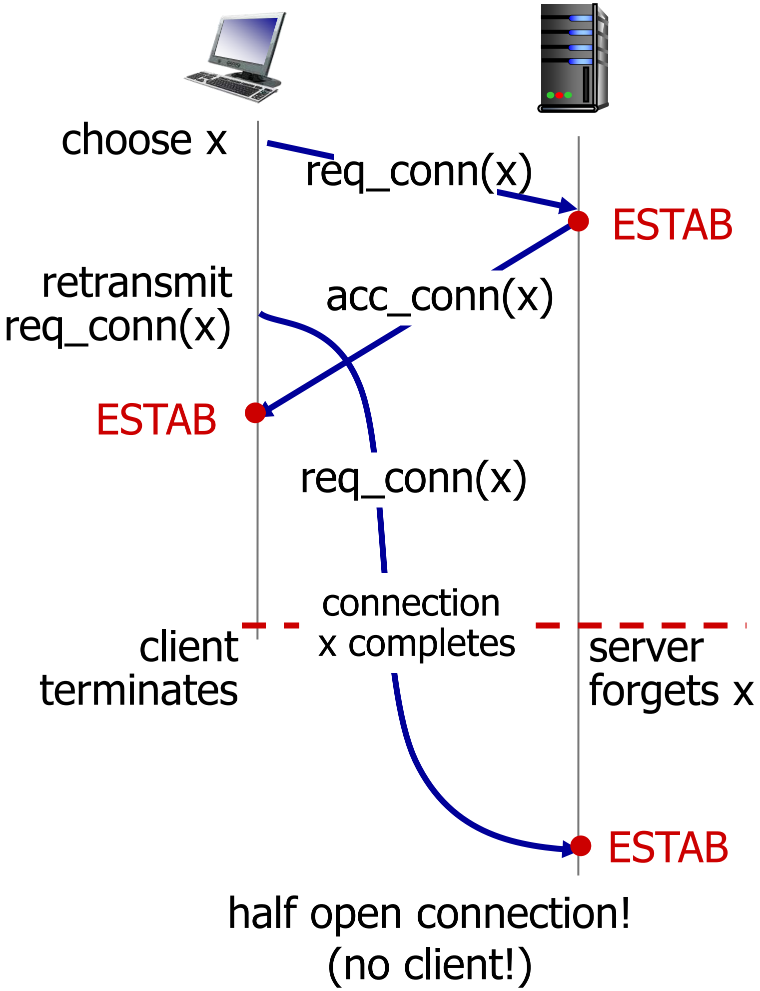
Q: will 2-way handshake always work in network?

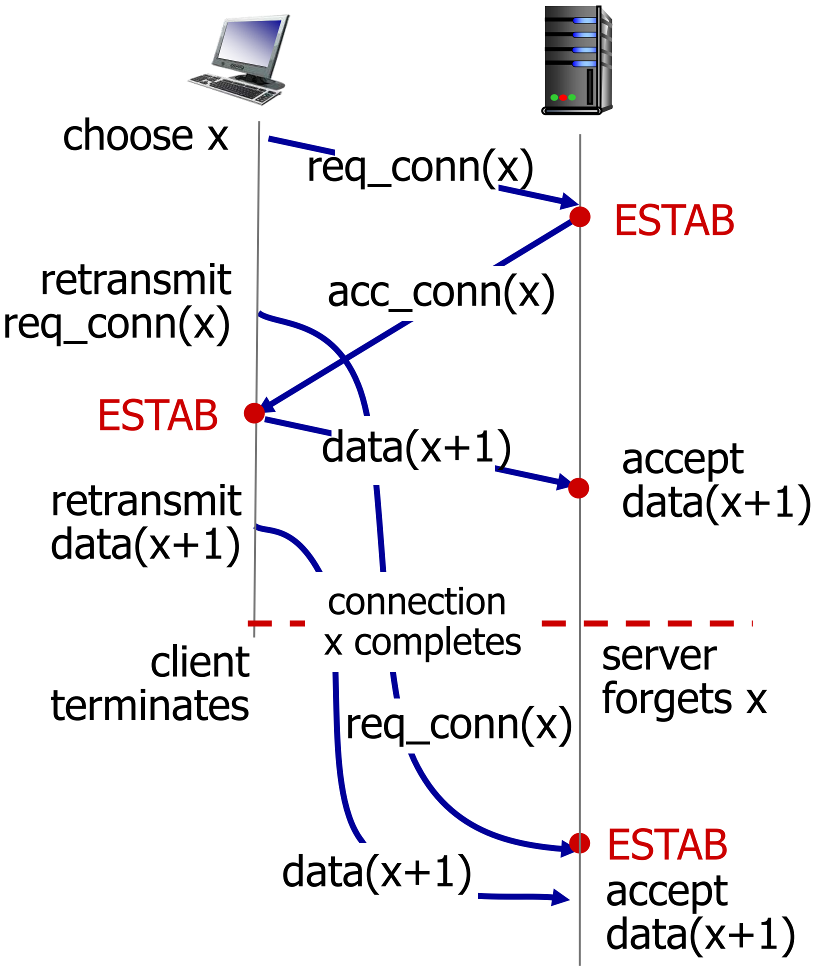
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

Agreeing to establish a connection

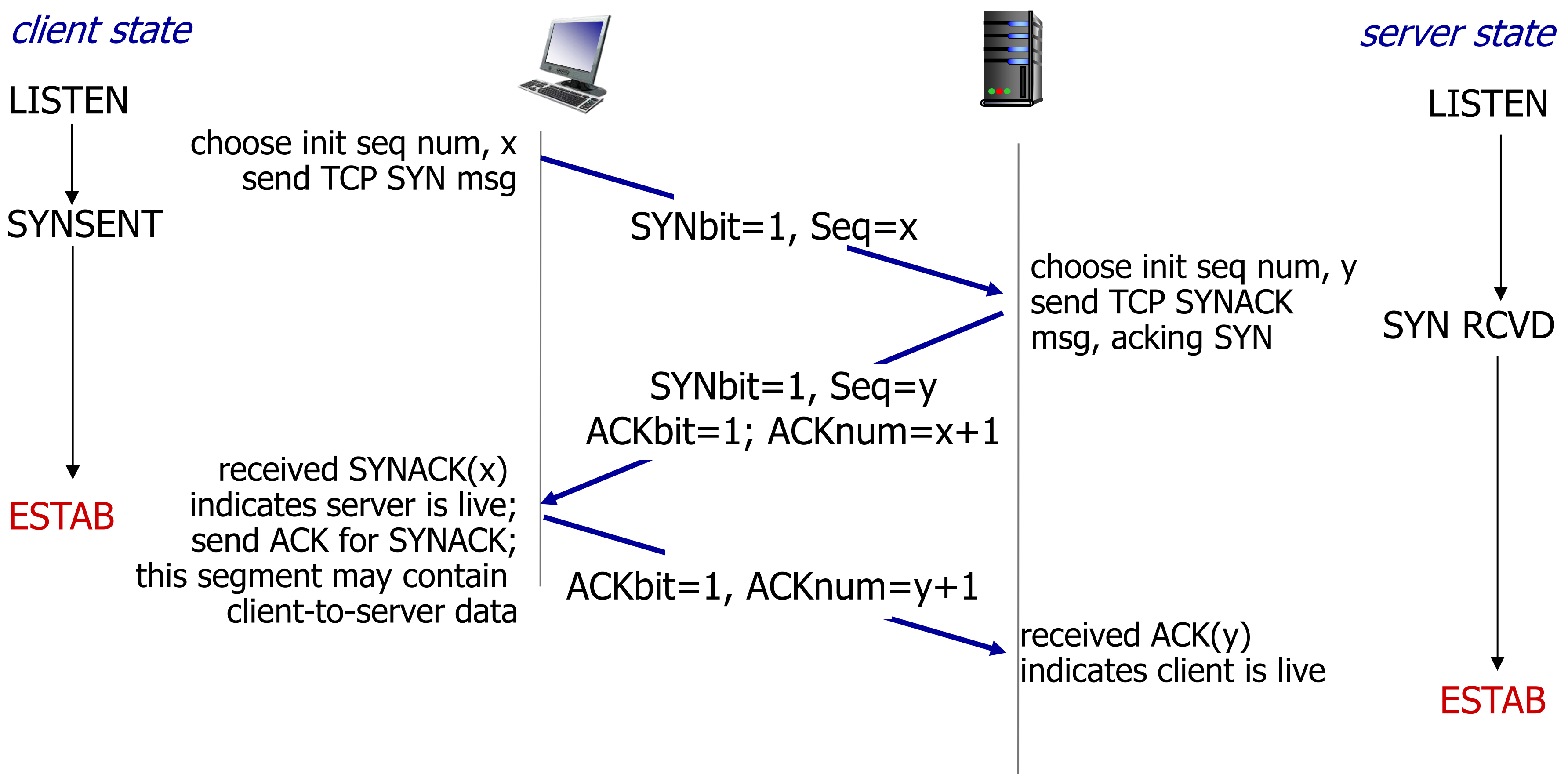
PES UNIVERSITY ONLINE

2-way handshake failure scenarios:

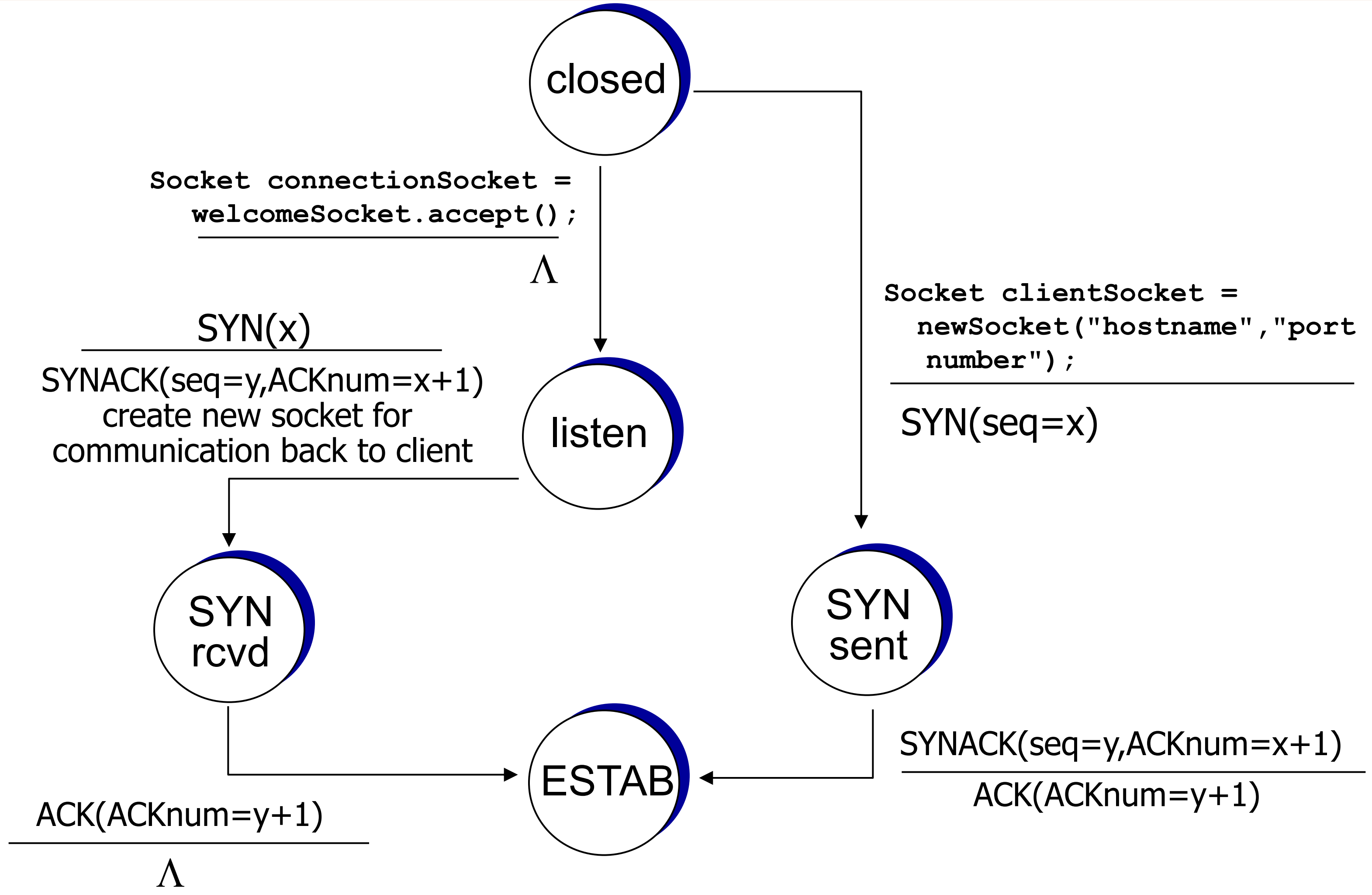




TCP 3-way handshake



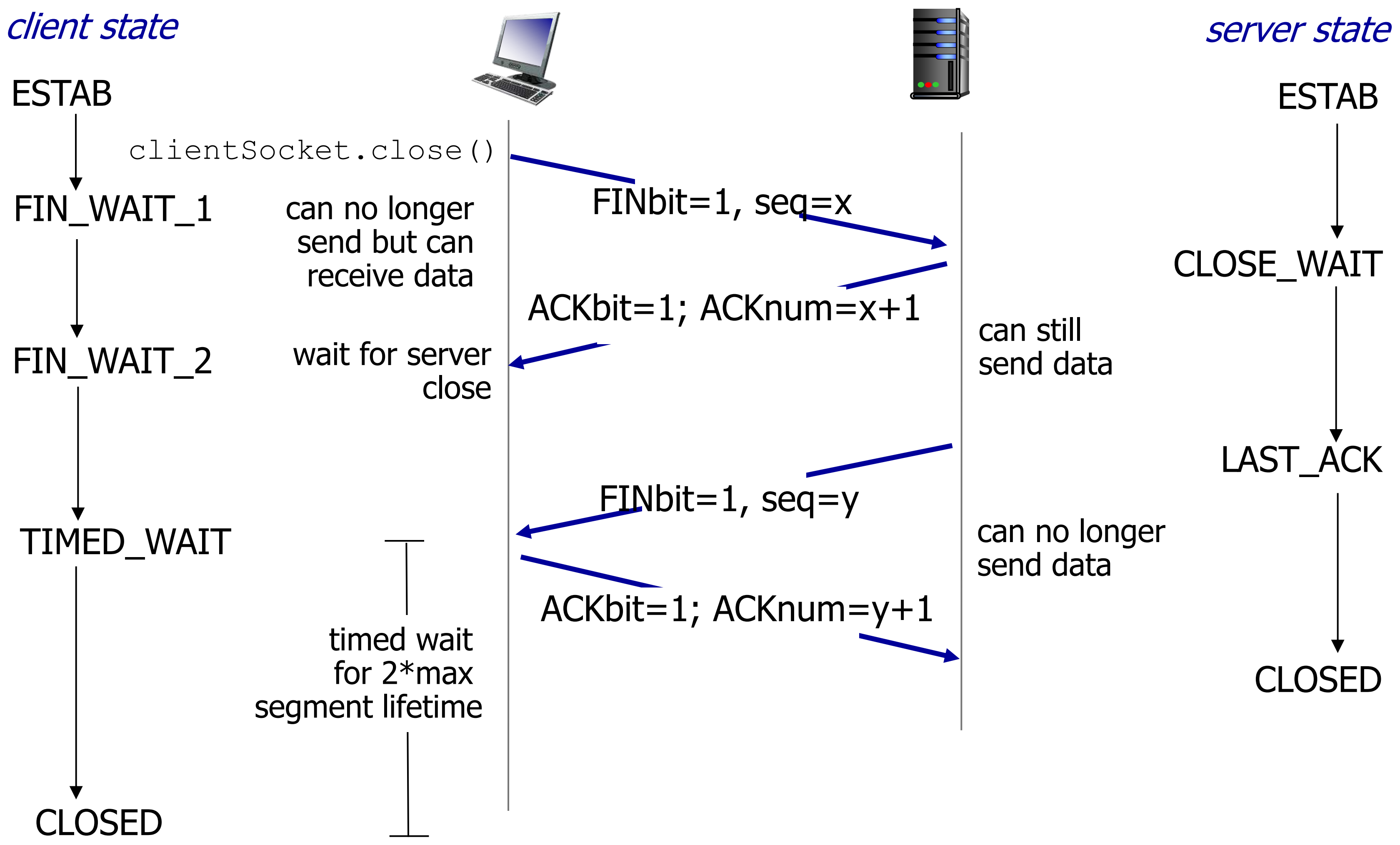
TCP 3-way handshake: FSM



TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection



THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 66186603

COMPUTER NETWORKS

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri

Department of Computer Science & Engineering

COMPUTER NETWORKS

Principles of Congestion Control

Animesh Giri
Department of Computer Science & Engineering

In this segment

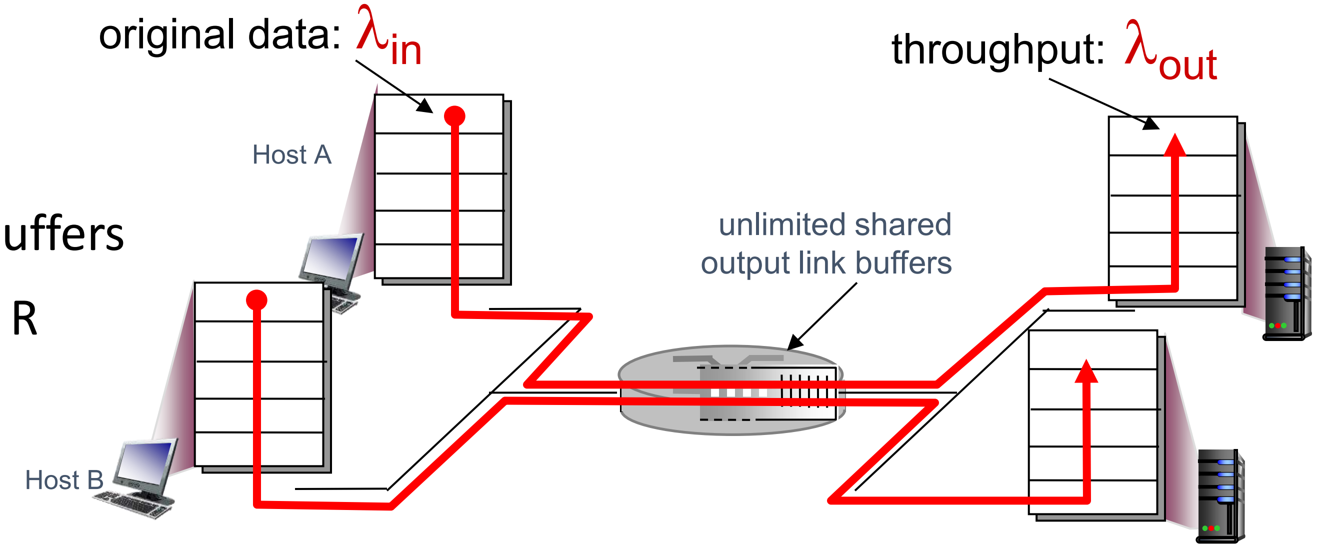
- Principles of congestion control
- Causes/costs of congestion: scenario 1
- Causes/costs of congestion: scenario 2
- Causes/costs of congestion: scenario 3

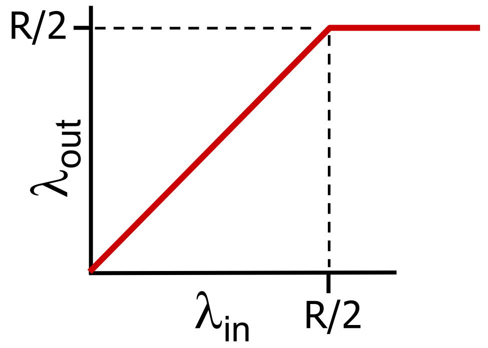
congestion:

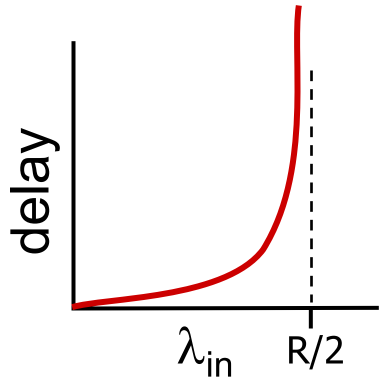
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission





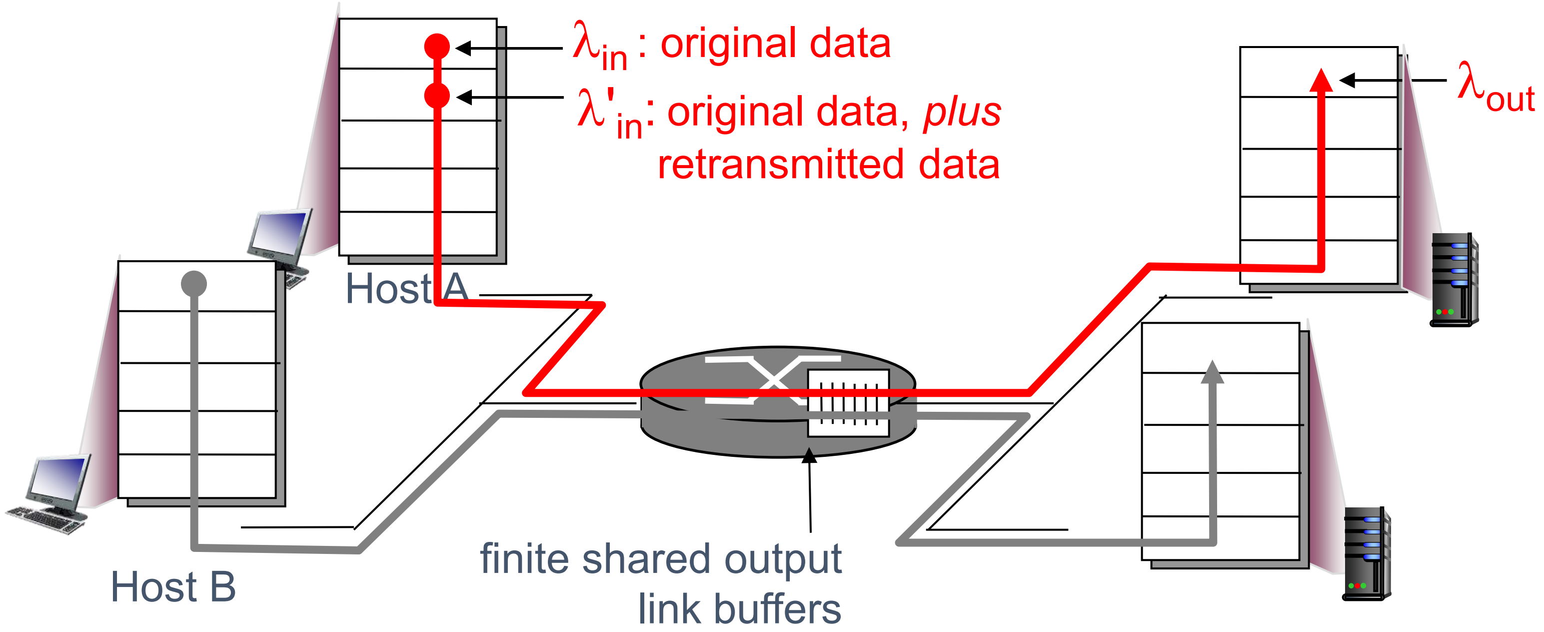


- maximum per-connection throughput: R/2

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

Causes/costs of congestion: scenario 2

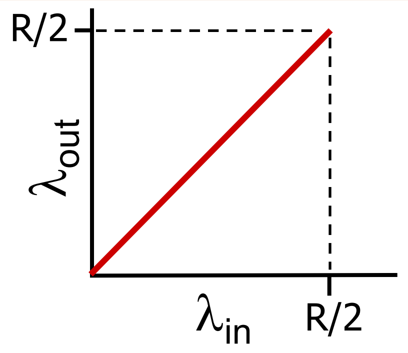
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda_{in} \geq \lambda'_{in}$

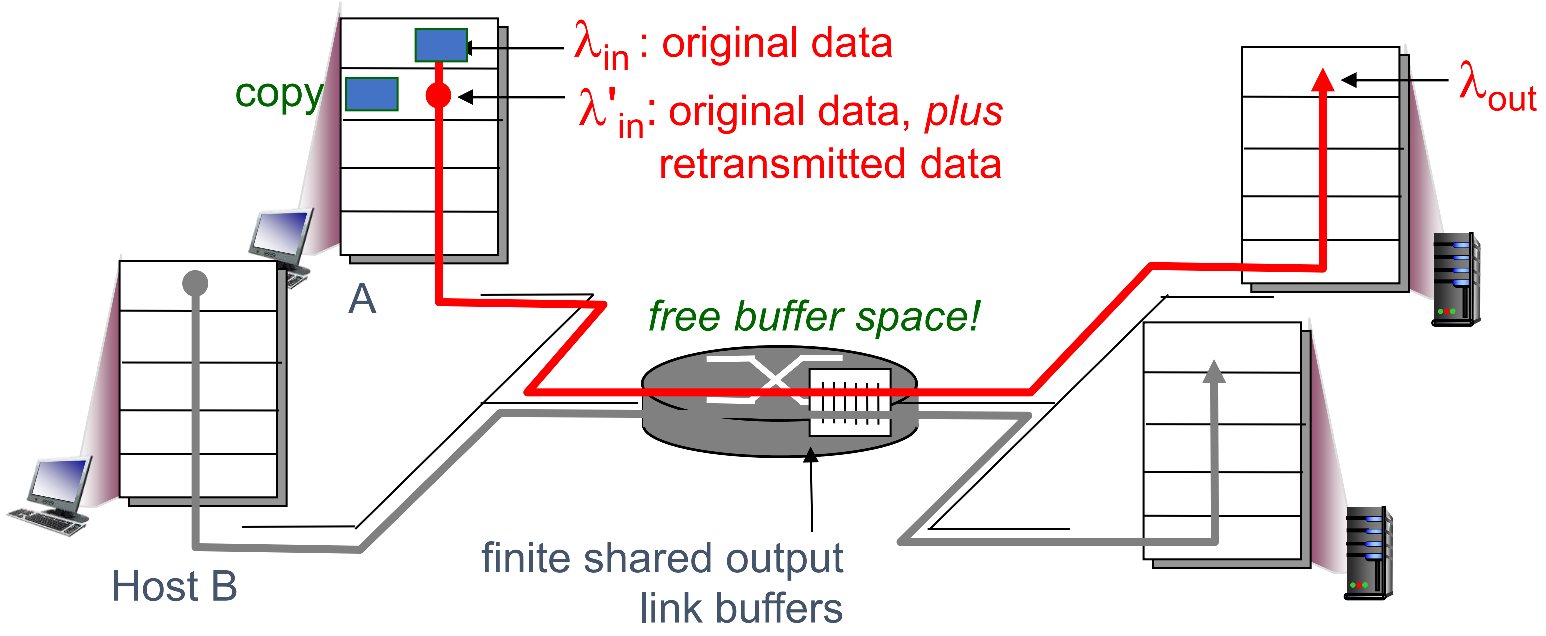


Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

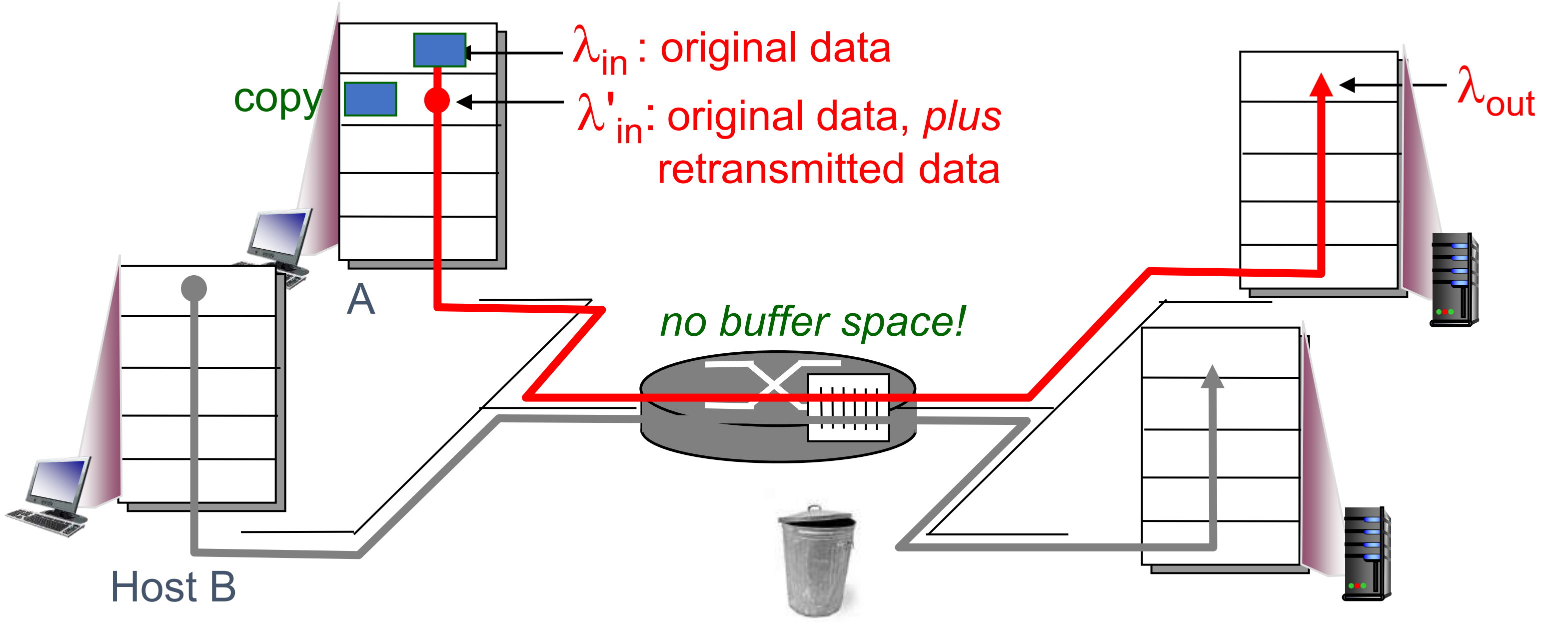




Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost, dropped at router due to full buffers

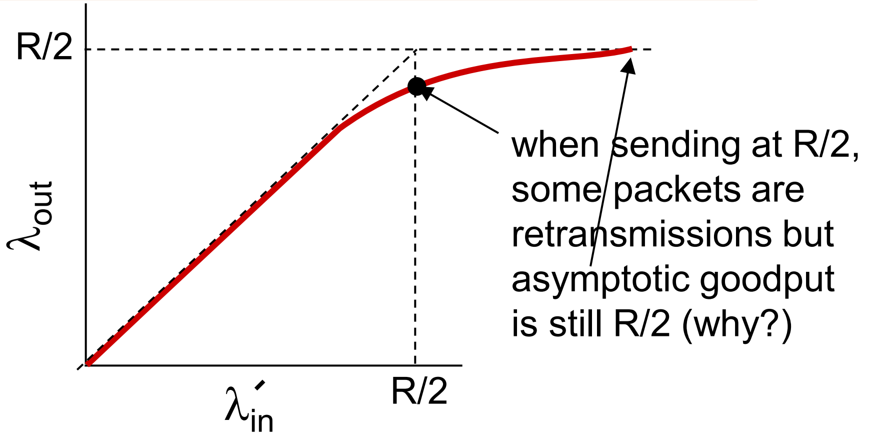
- sender only resends if packet *known* to be lost

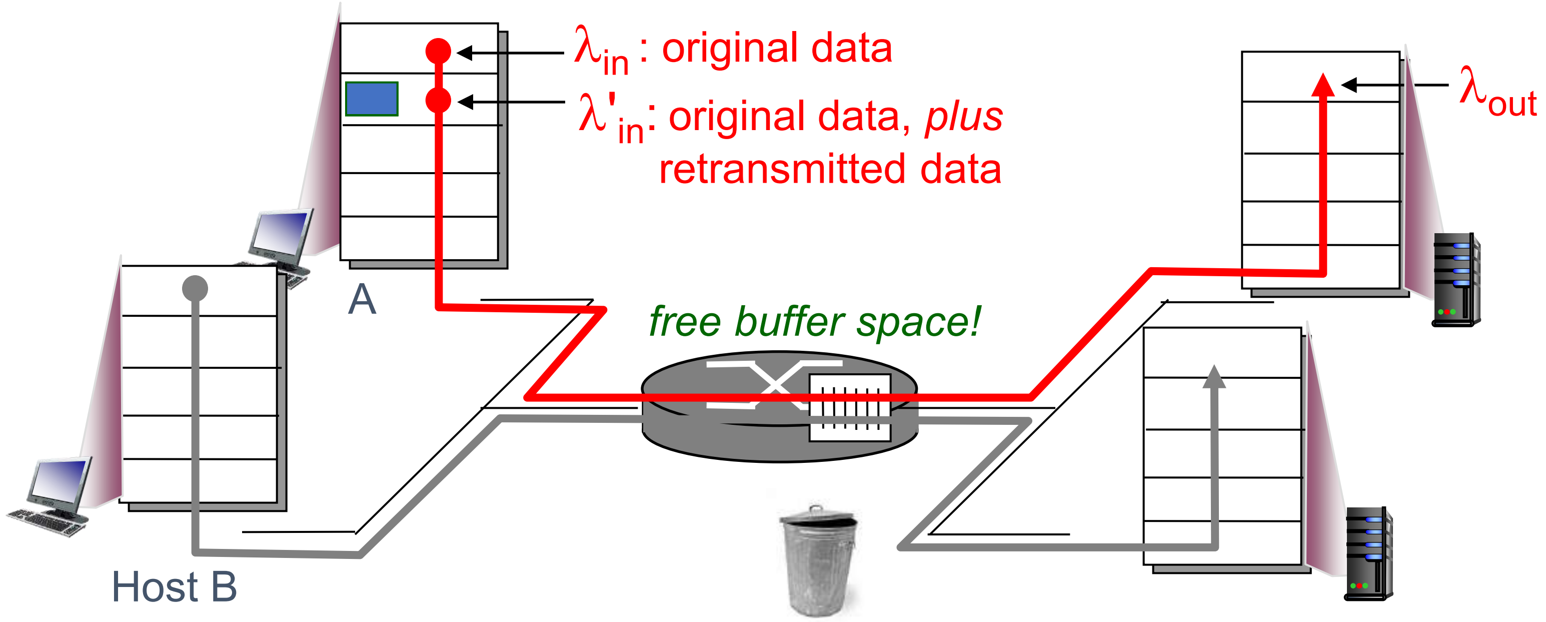


Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

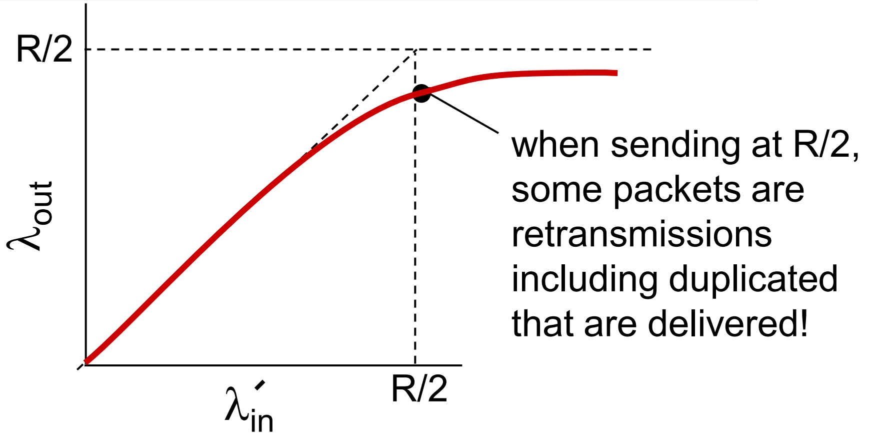


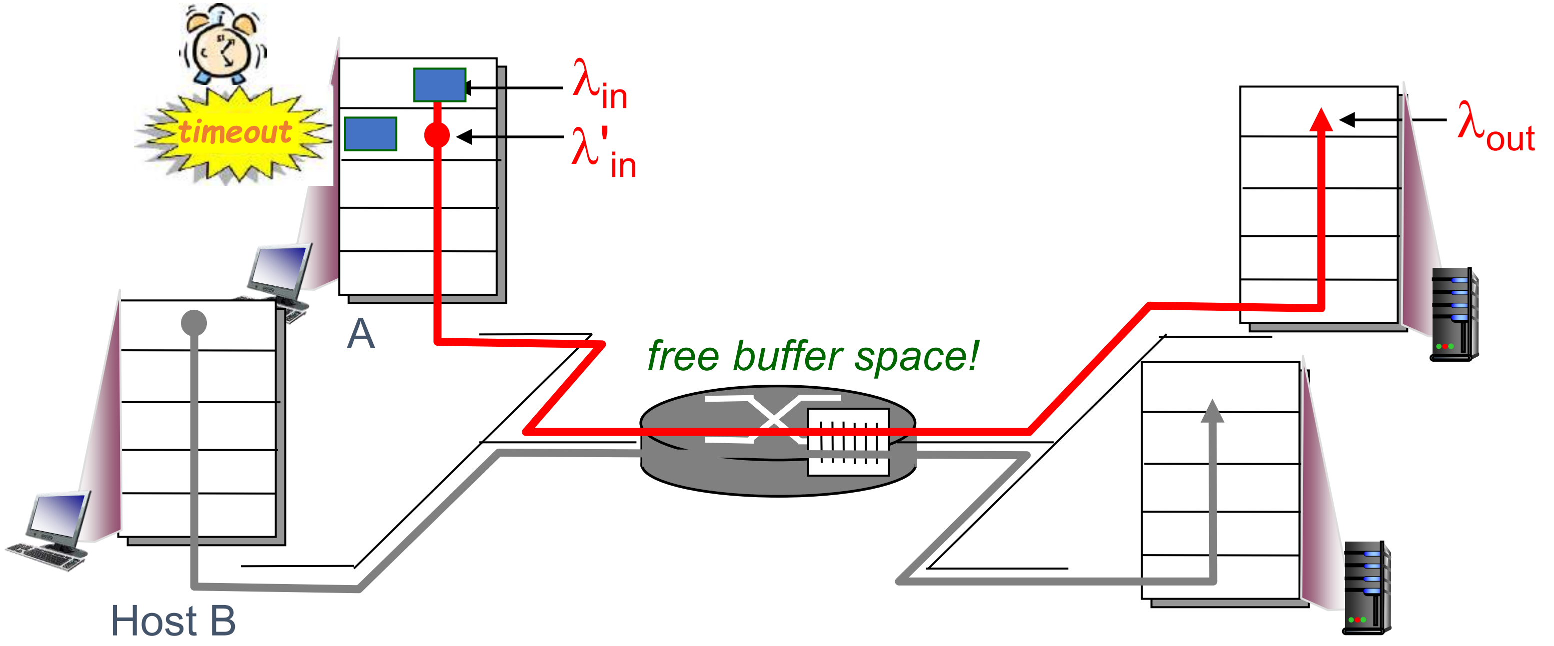


Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

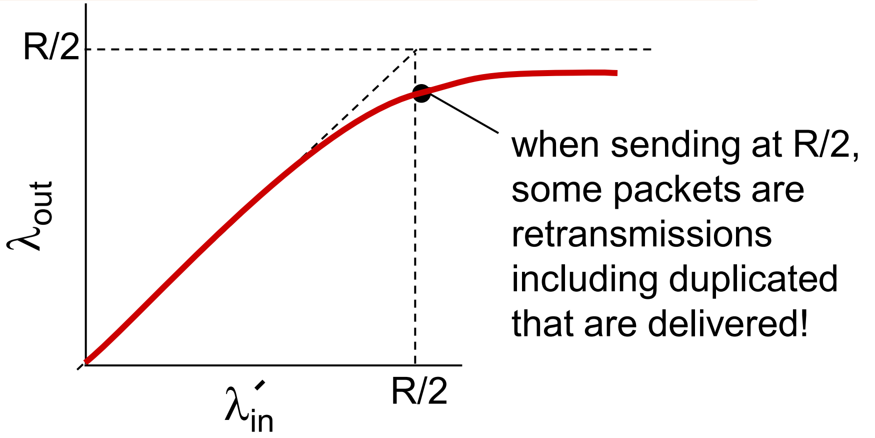




Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



"costs" of congestion:

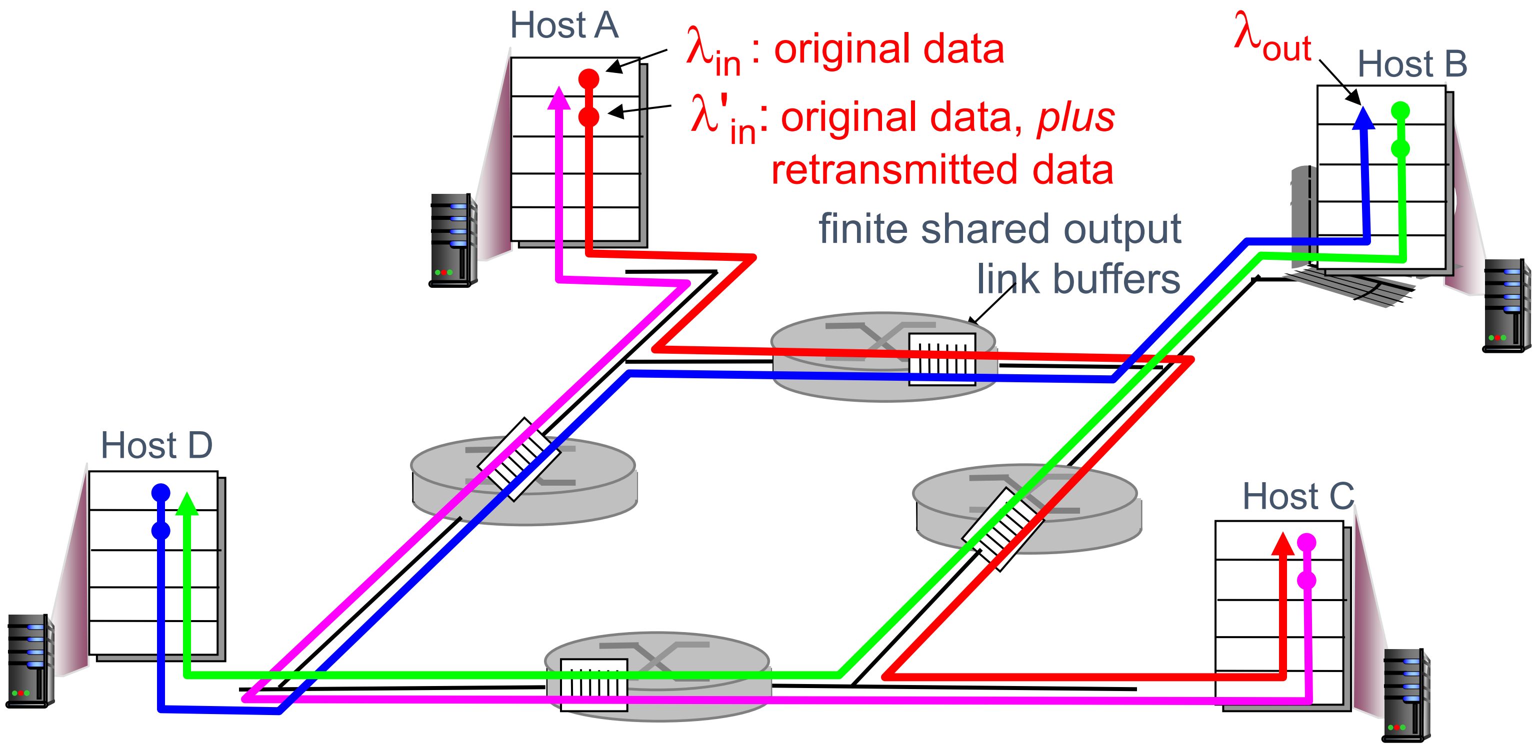
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

Causes/costs of congestion: scenario 3

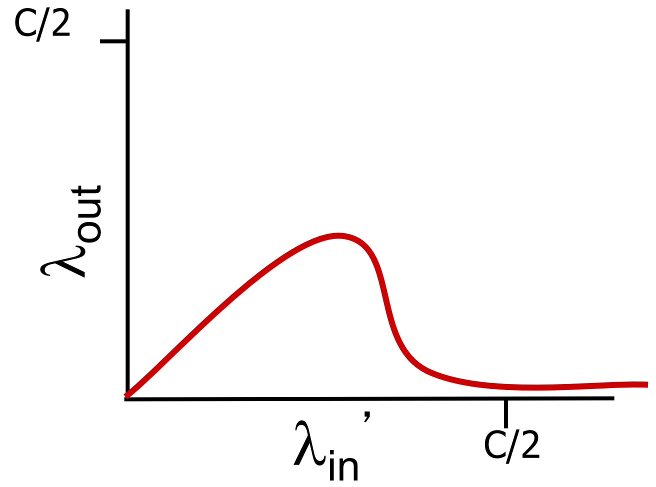
- four senders
- multihop paths
- timeout/retransmit

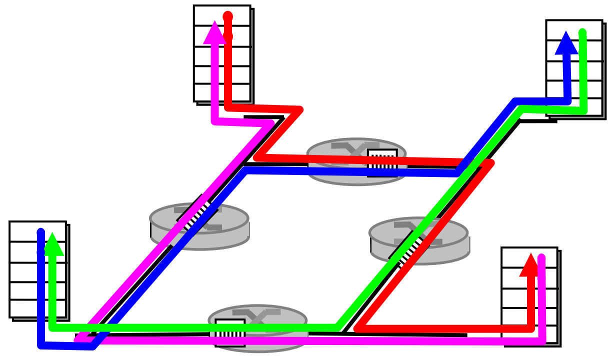
Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



Causes/costs of congestion: scenario 3





another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 66186603

COMPUTER NETWORKS

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

Transport Layer

Animesh Giri
Department of Computer Science & Engineering

COMPUTER NETWORKS

TCP Congestion Control

Animesh Giri
Department of Computer Science & Engineering

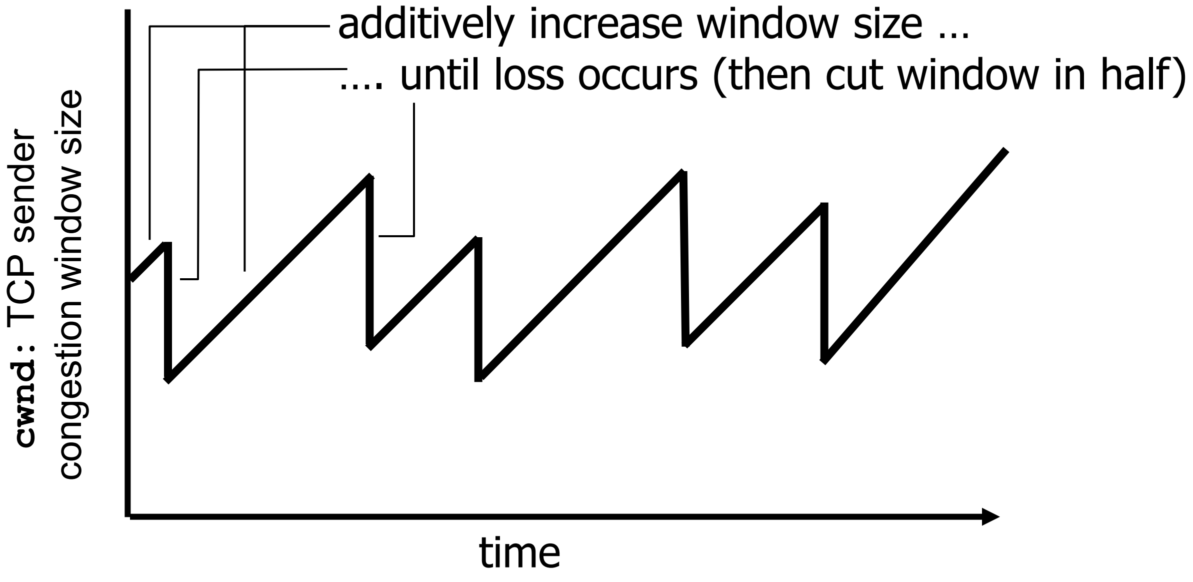
In this segment

- TCP congestion control: additive increase multiplicative decrease
- TCP Congestion Control: details
- TCP Slow Start
- TCP: detecting, reacting to loss
- TCP: switching from slow start to CA
- Summary: TCP Congestion Control
- TCP throughput
- TCP Futures: TCP over “long, fat pipes”
- TCP Fairness
- Why is TCP fair?
- Explicit Congestion Notification (ECN)

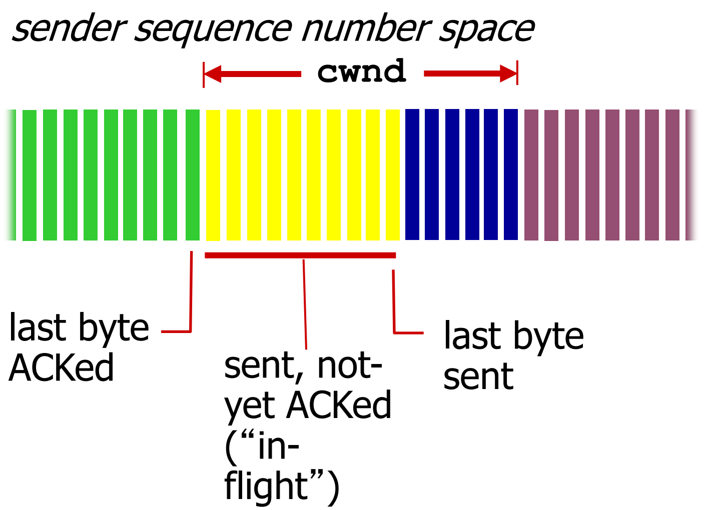
TCP congestion control: additive increase multiplicative decrease

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

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control: details



- sender limits transmission:

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

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

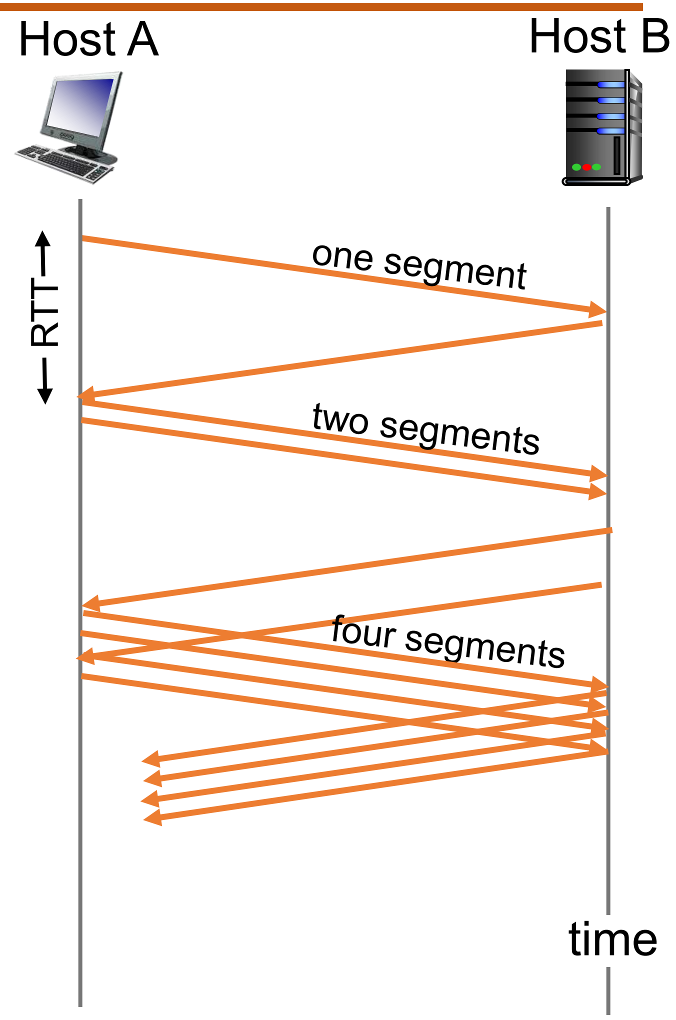
TCP sending rate:

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

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

TCP Slow Start

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



TCP: detecting, reacting to loss

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

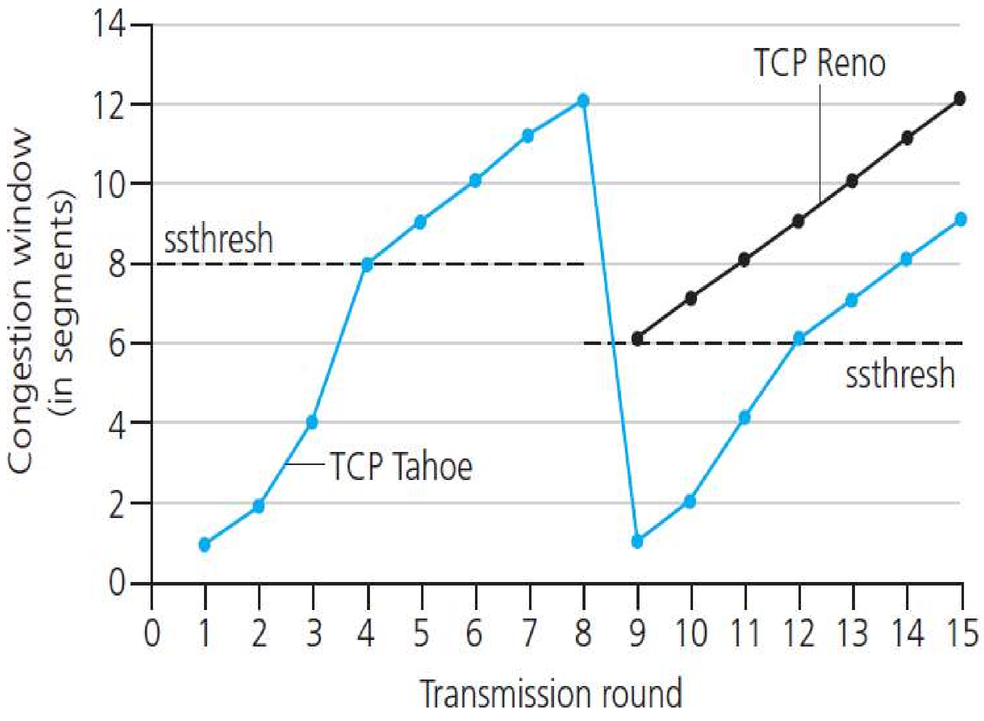
TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

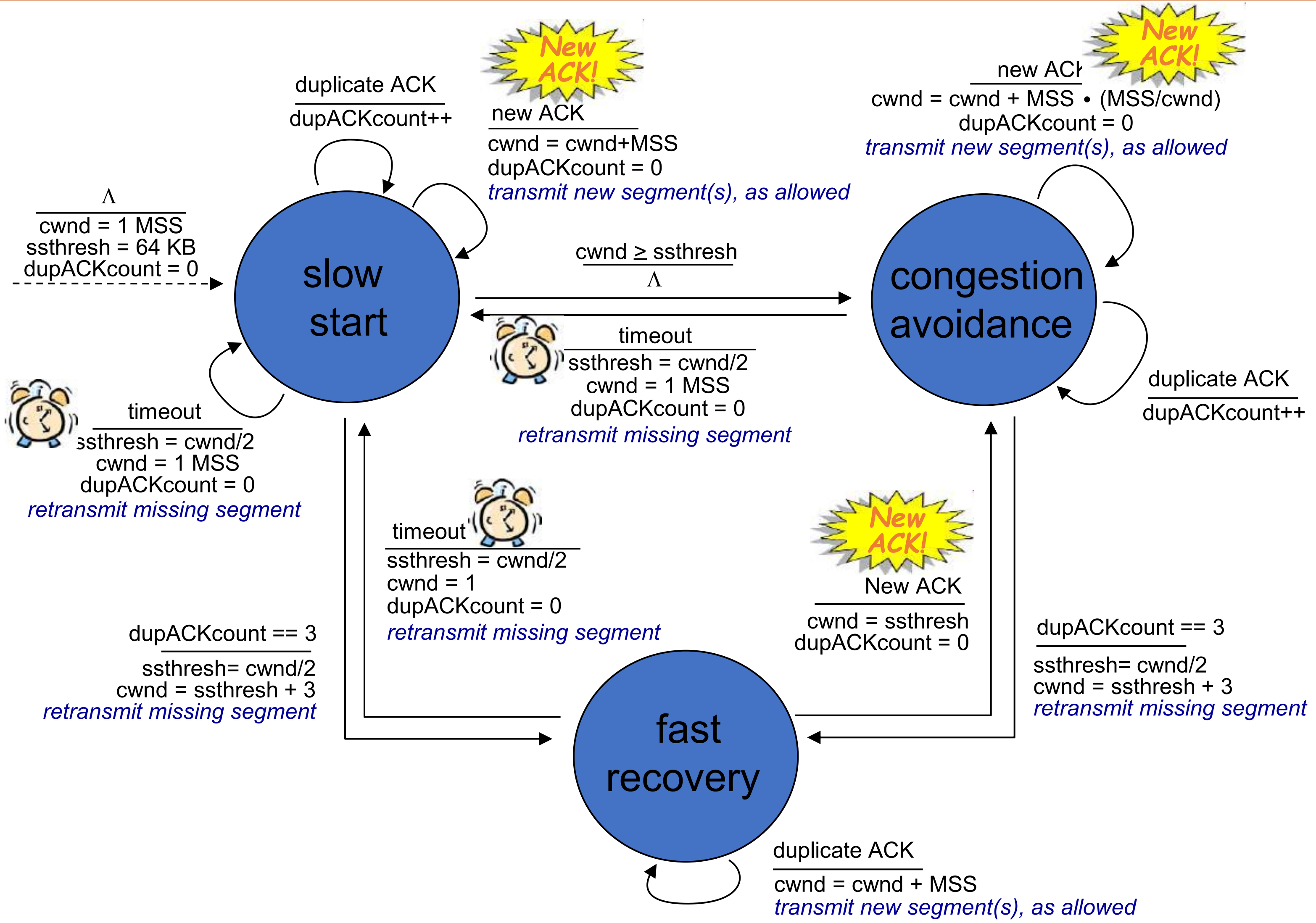
A: when **cwnd** gets to 1/2 of its value before timeout.

Implementation:

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



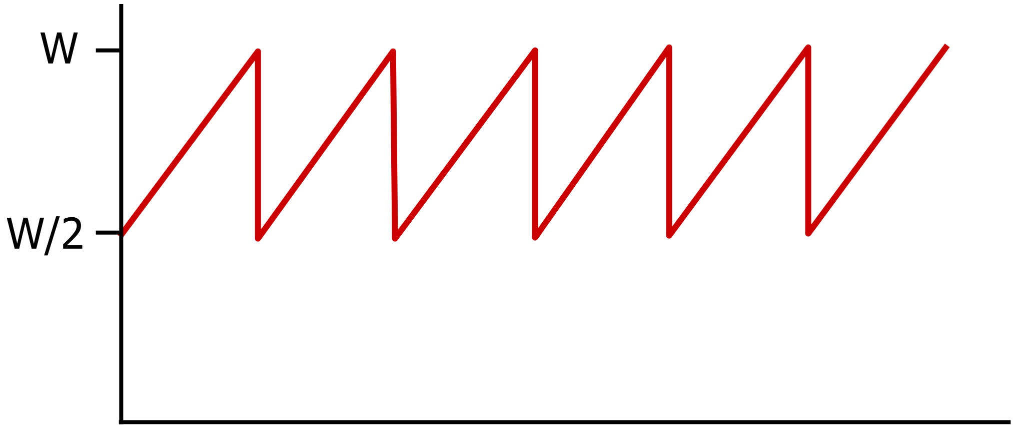
Summary: TCP Congestion Control



TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

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



TCP Futures: TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

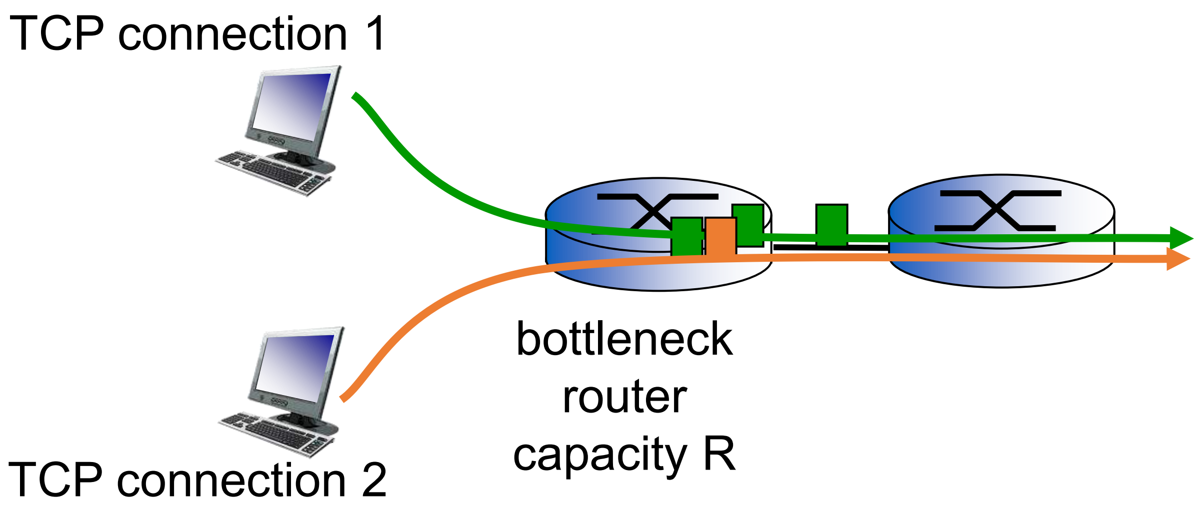
$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ *– a very small loss rate!*

- new versions of TCP for high-speed

TCP Fairness

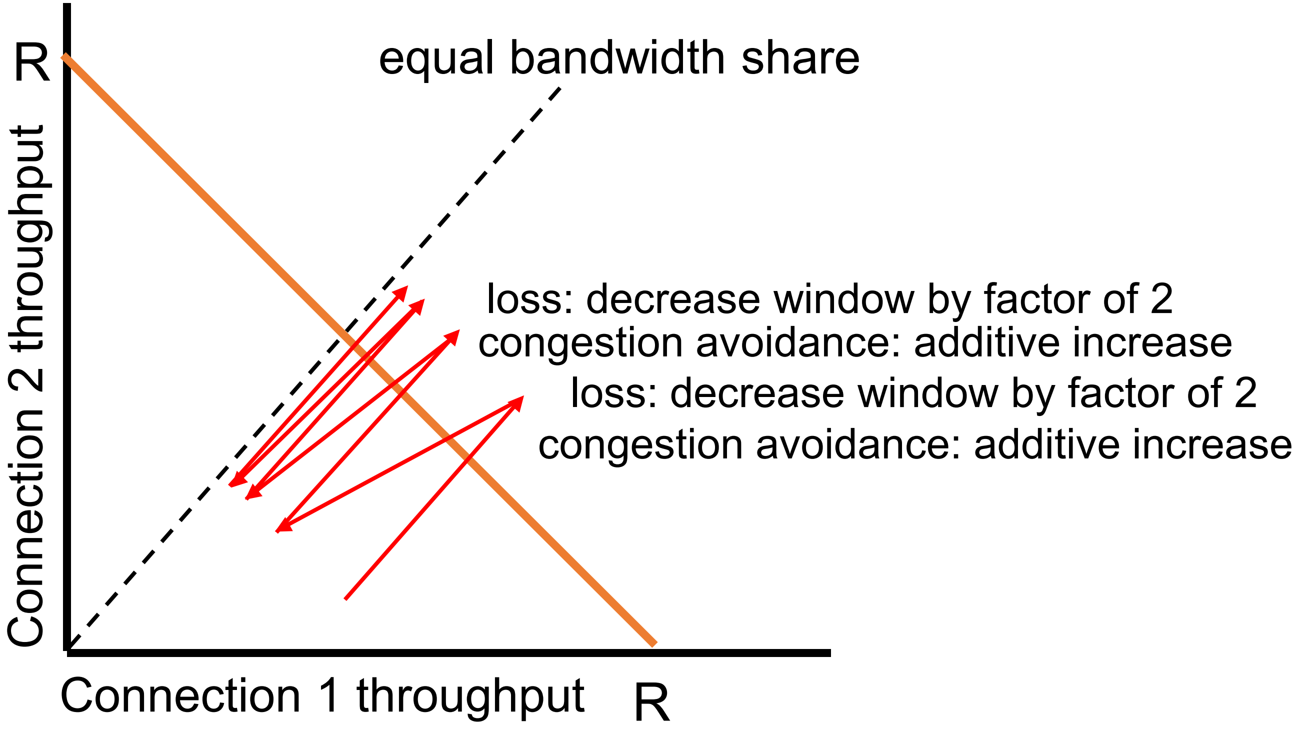
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

two competing sessions:

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



Fairness (more)

Fairness and UDP

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

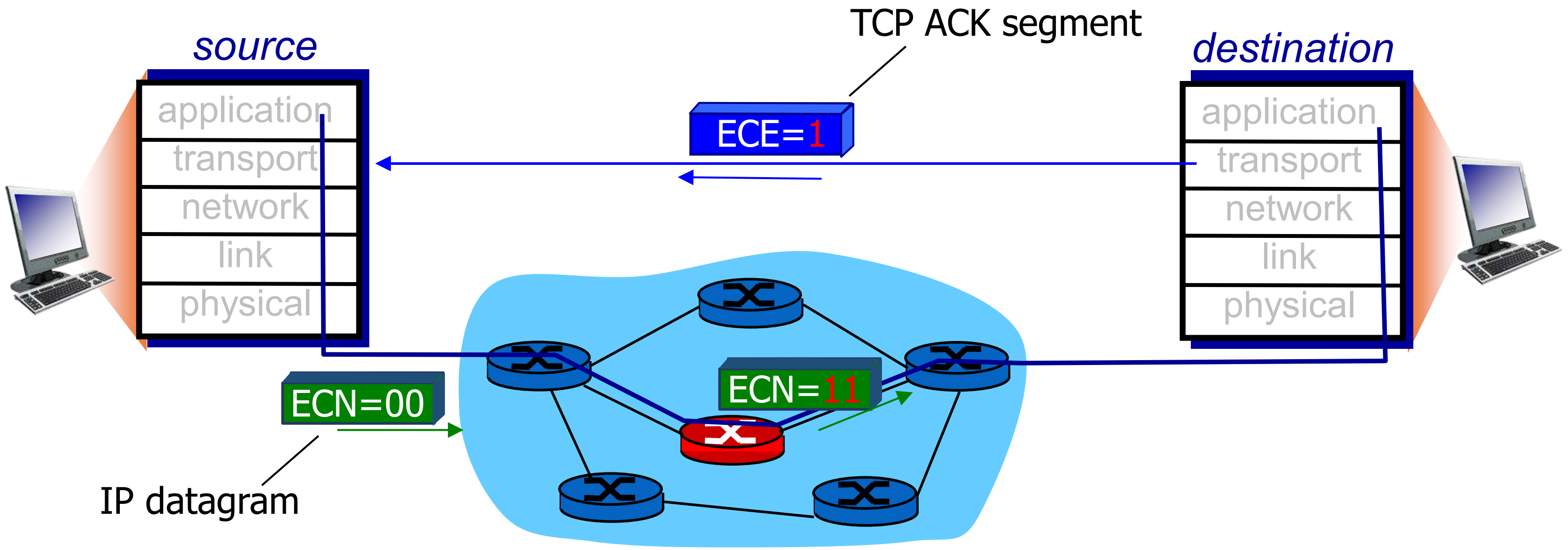
Fairness, parallel TCP connections

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

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



THANK YOU

Animesh Giri
Department of Computer Science & Engineering

animeshgiri@pes.edu

+91 80 6618 6603