# Transport Layer

our 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

# Transport Layer

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

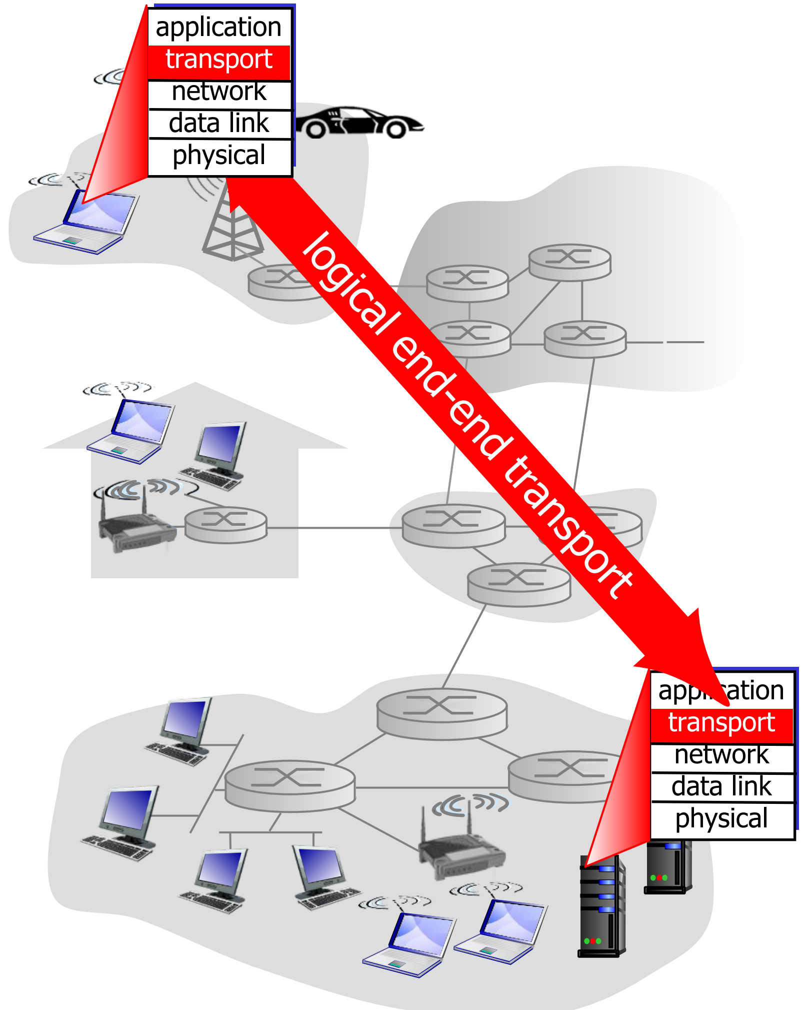
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

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
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



# Transport vs. network layer

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

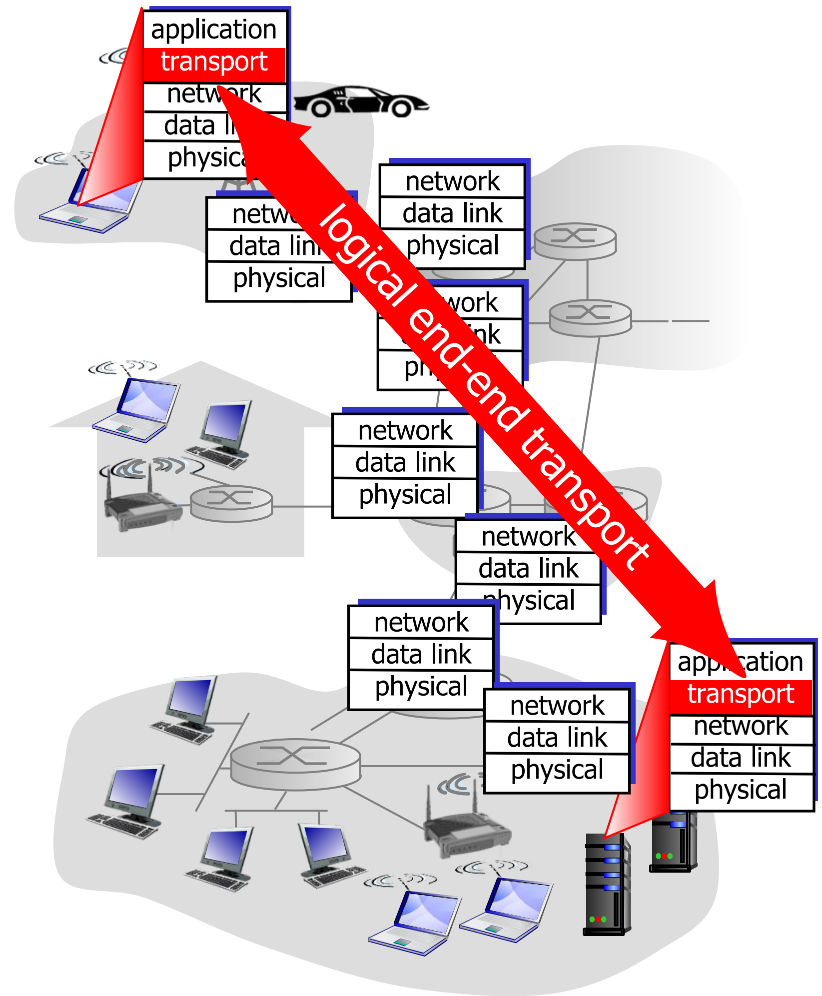
*household analogy:*

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

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

# Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



# Transport Layer

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

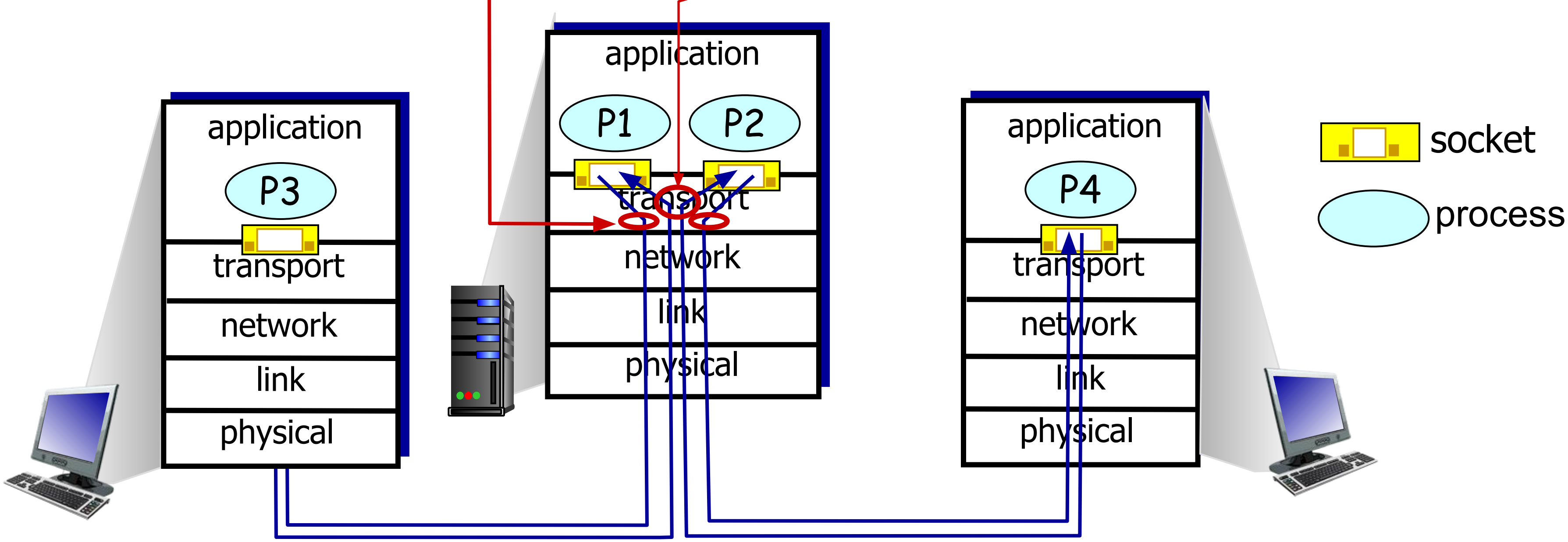
# Multiplexing/demultiplexing

*multiplexing at sender:*

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

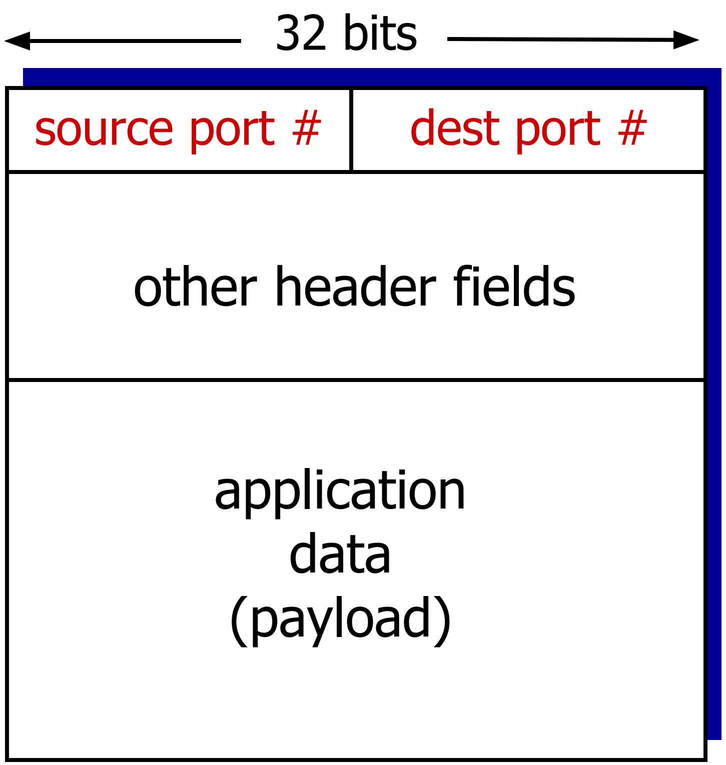
*demultiplexing at receiver:*

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 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 #

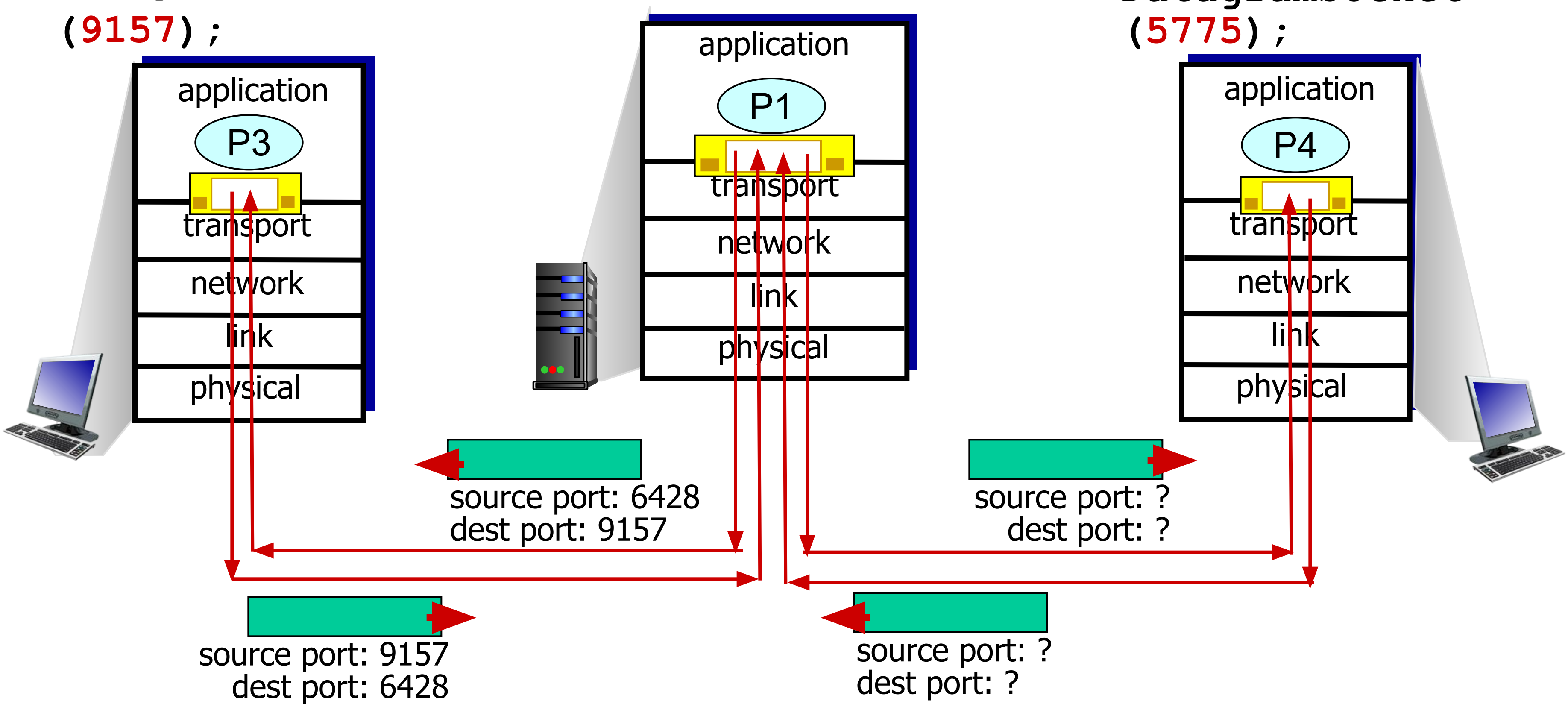
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

# Connectionless demux: example

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

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

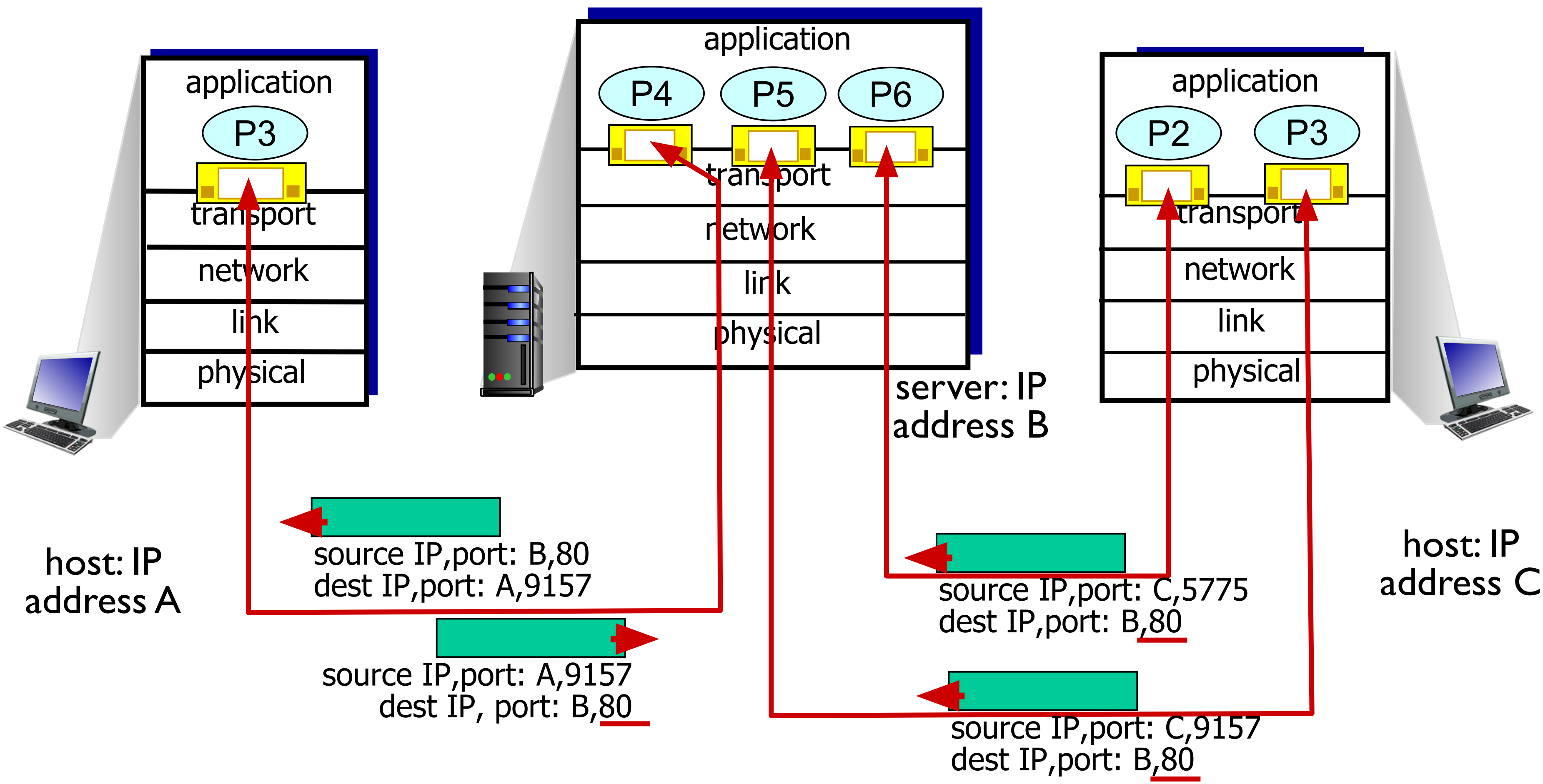
```
DatagramSocket
 mySocket1 = new
 DatagramSocket
  (5775);
```



# Connection-oriented demux

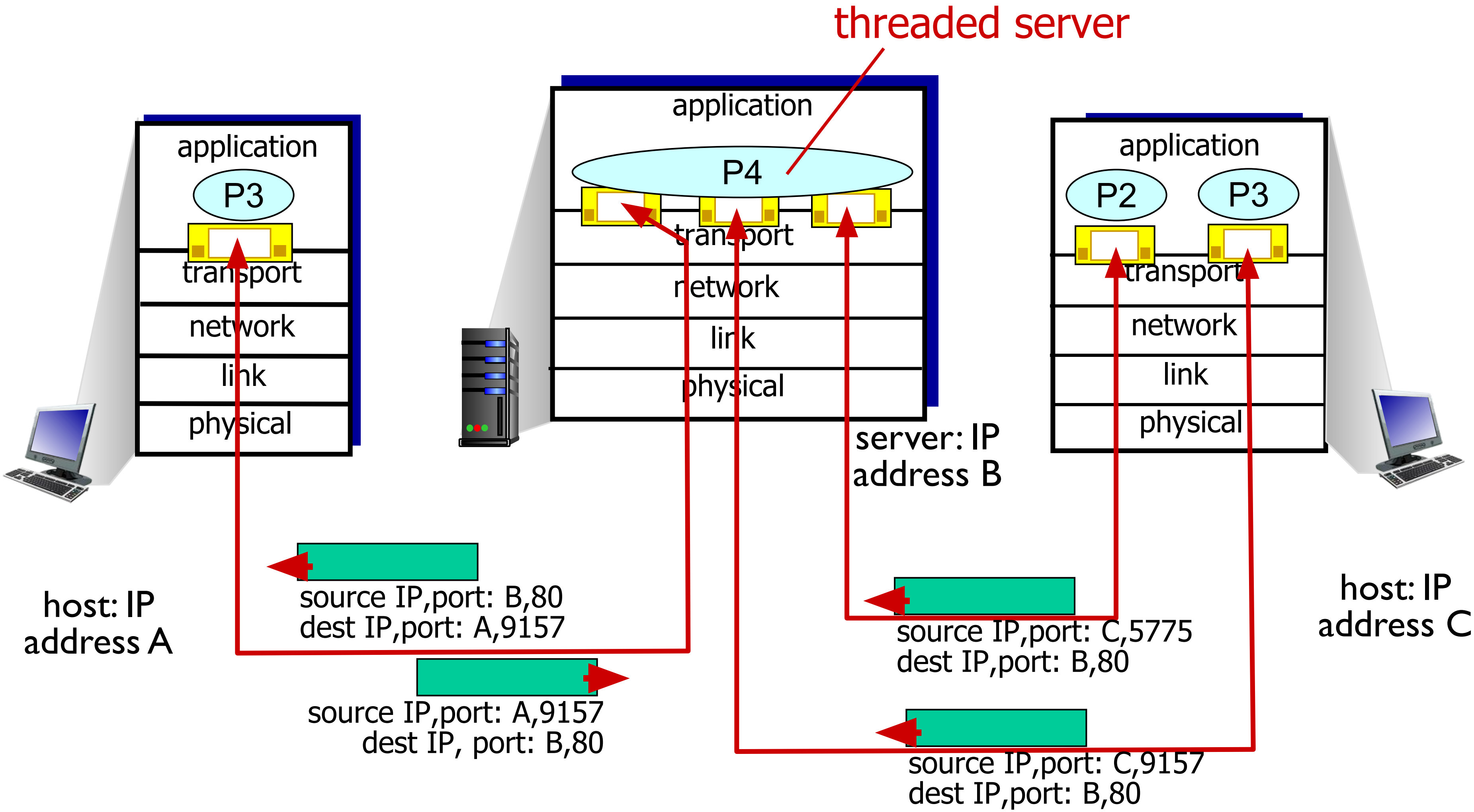
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- 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

# Connection-oriented demux: example



# Transport Layer

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

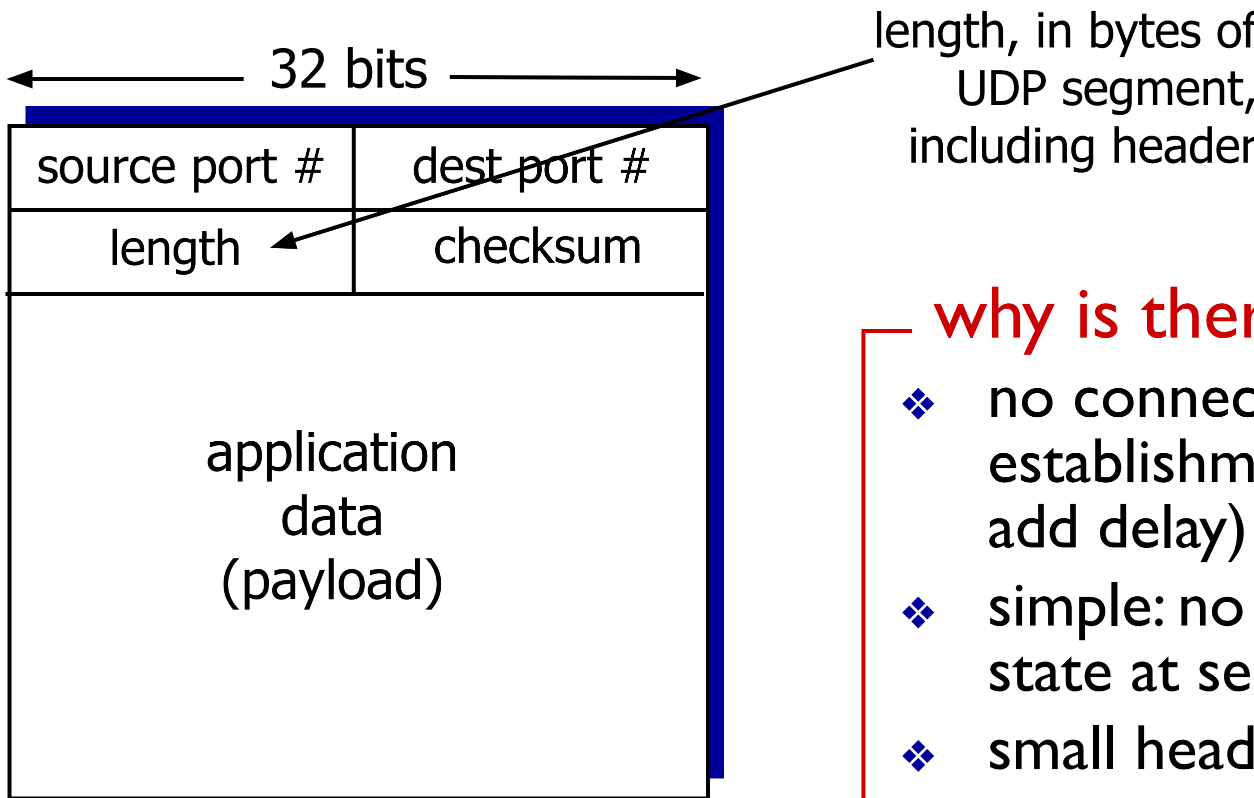
3.7 TCP congestion control

# 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 independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

# UDP: segment header




UDP segment format

## why is there a UDP?

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

# UDP 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

# Transport Layer

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

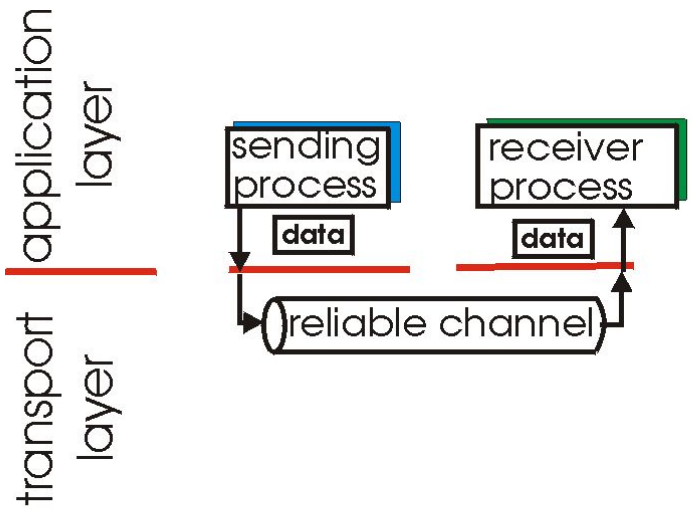
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

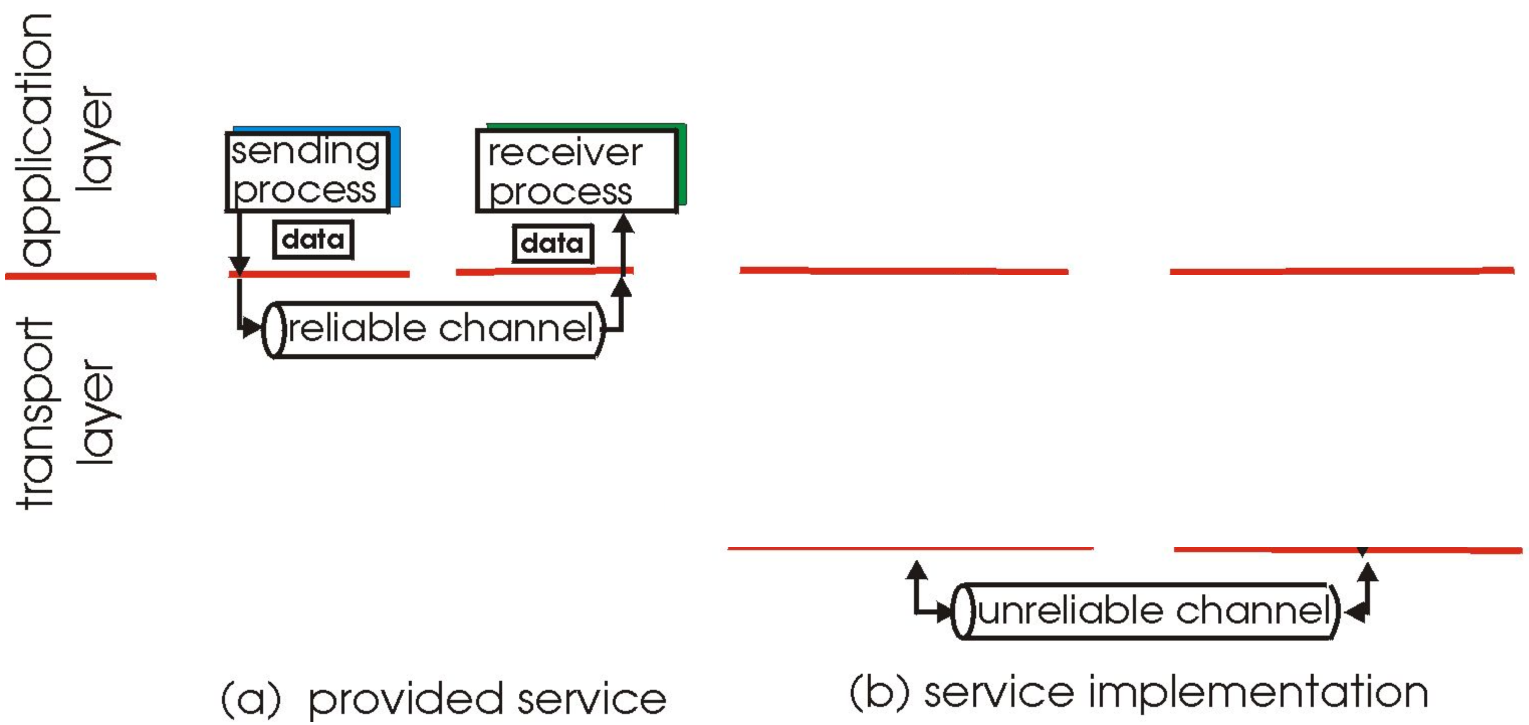
3.7 TCP congestion control

# Principles of reliable data transfer

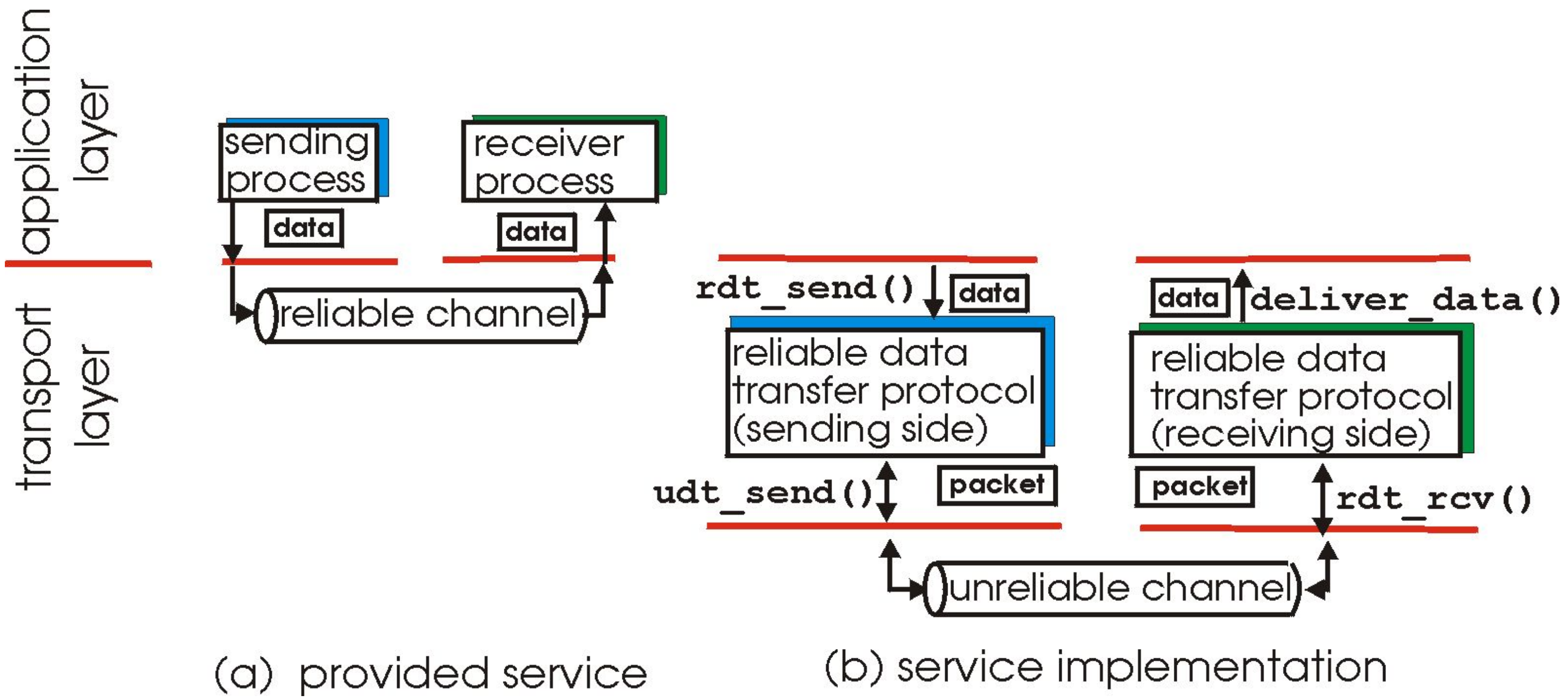



(a) provided service

# Principles of reliable data transfer



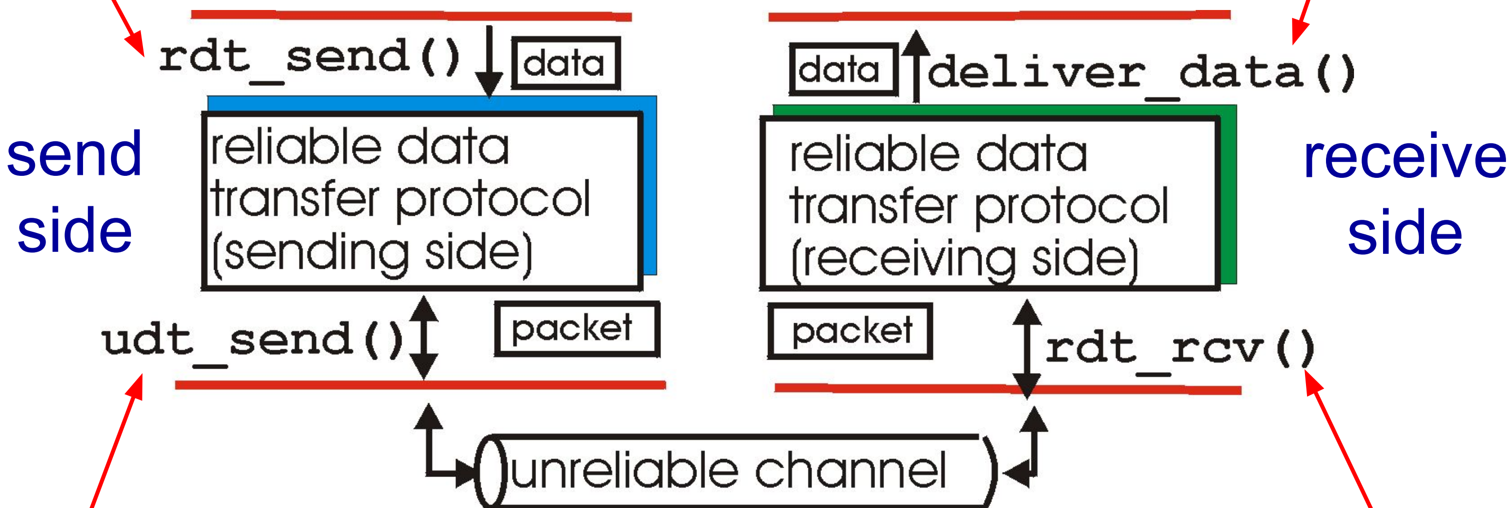
# Principles of reliable data transfer



# 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

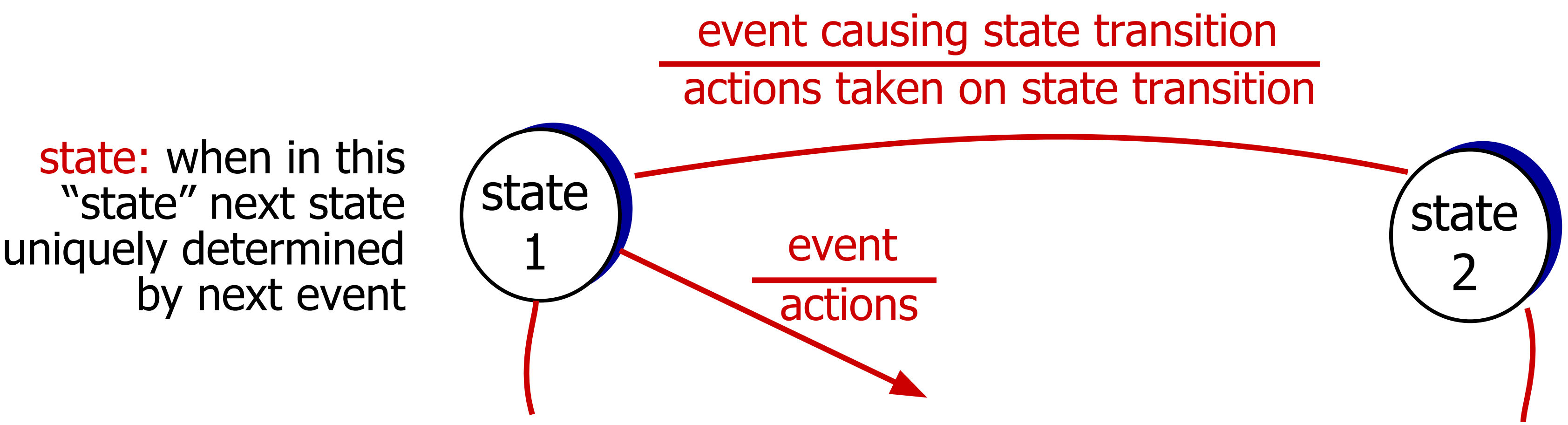



**udt_send():** called by rdt, to transfer packet over unreliable channel to receiver

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

# Reliable data transfer: getting started

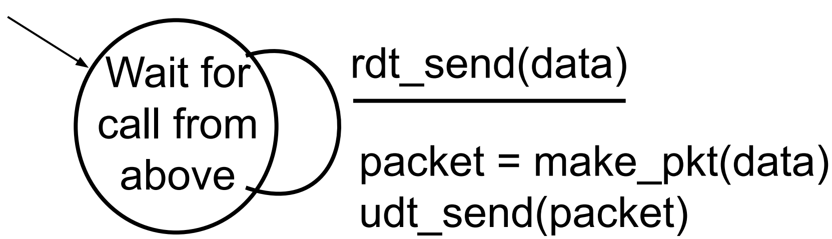
- Incremental Development
- Finite State Machines (FSM)



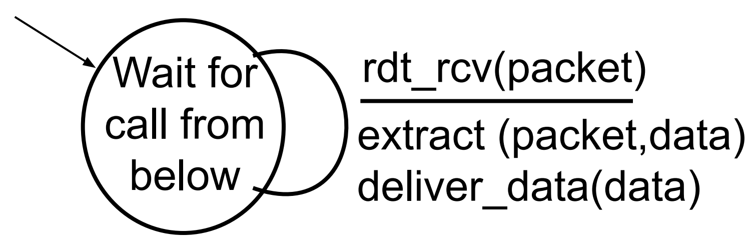
# rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - No Bit Errors
  - No Packets Loss

Sender




Receiver

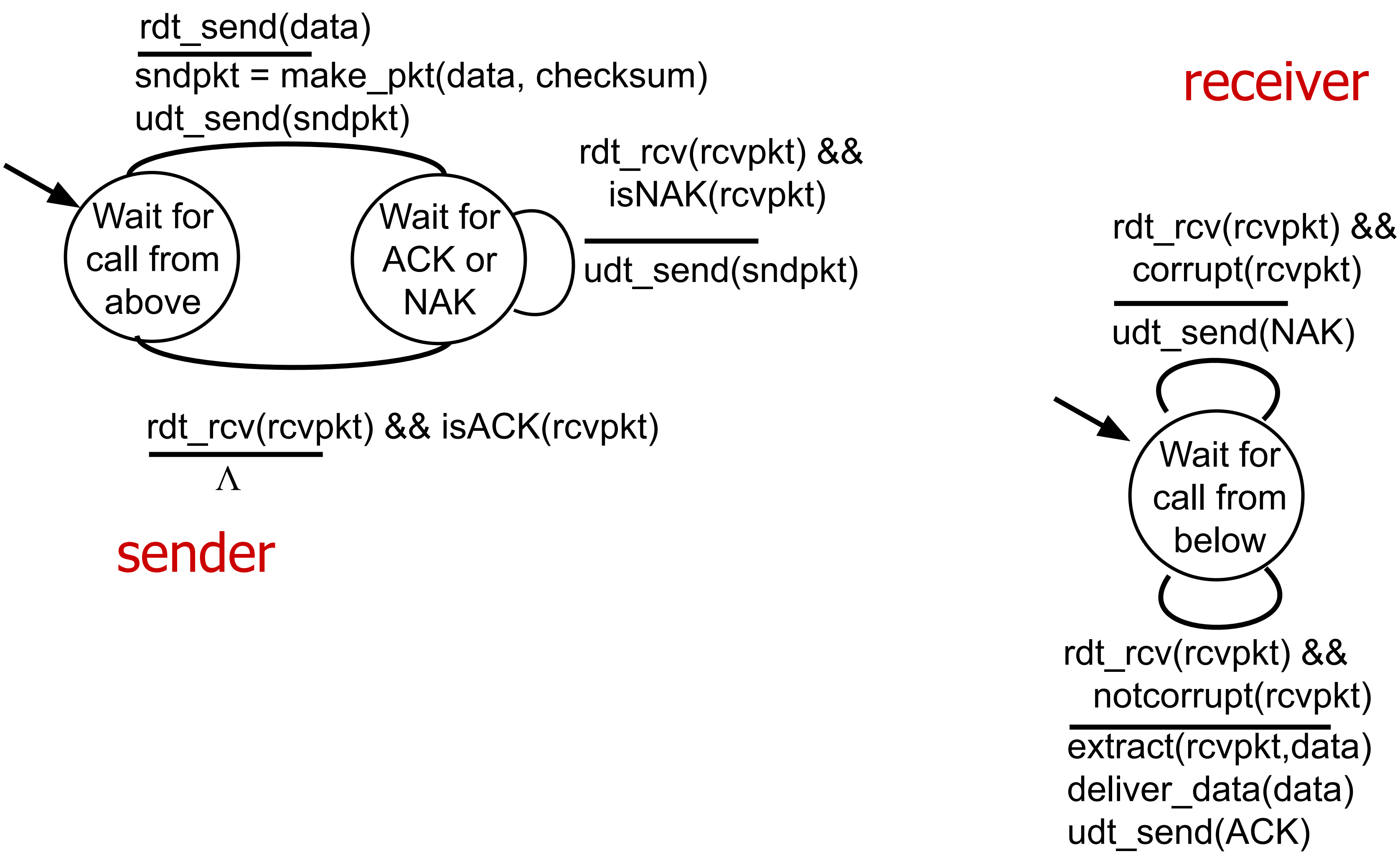


# rdt2.0: channel with bit errors

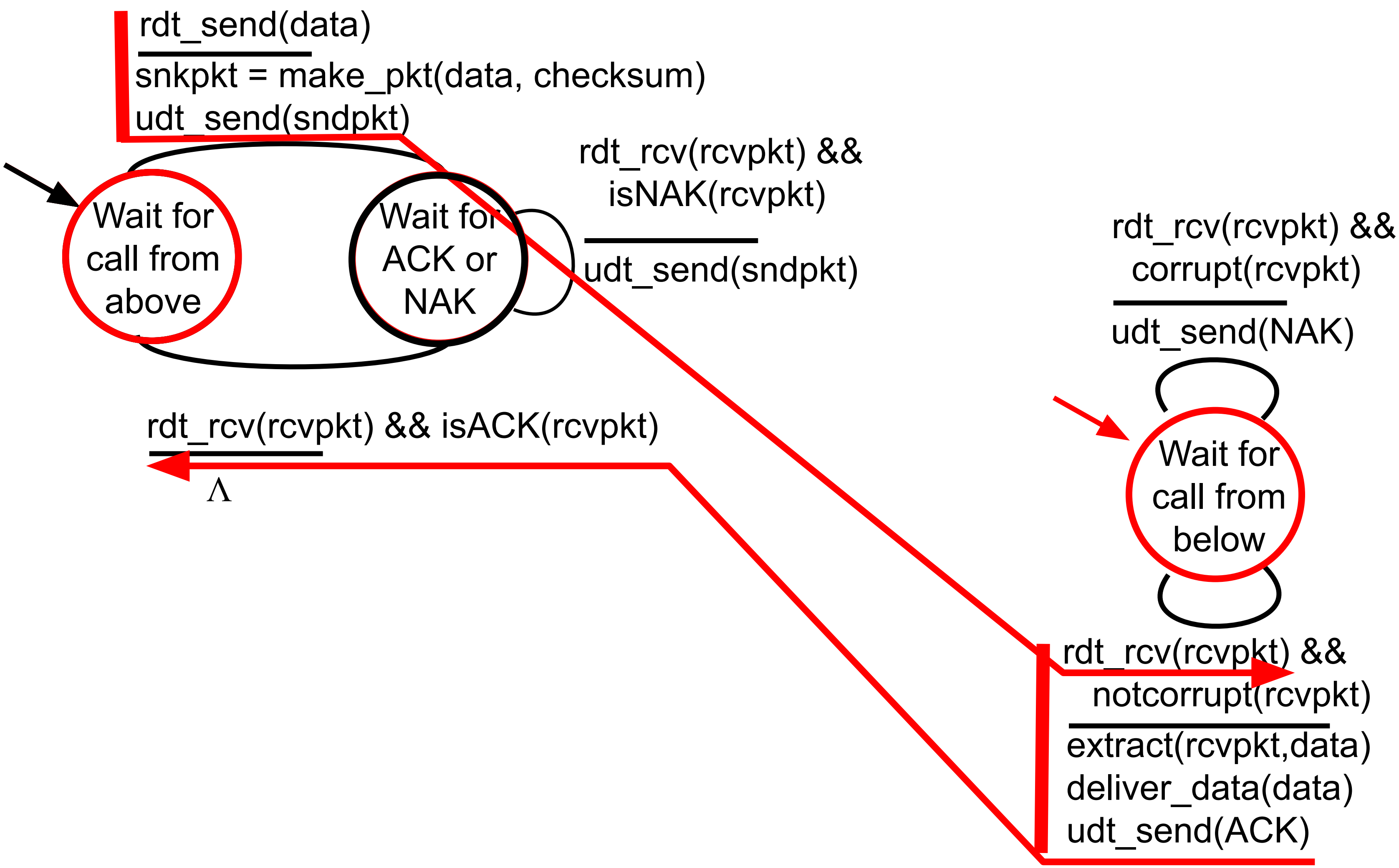
- Underlying channel
  - Bit Errors
- How to detect errors?
  - Checksum
- How to recover from errors?
  - Receiver Feedback
    - Acknowledgements (ACKs)
    - Negative acknowledgements (NAKs)
  - Retransmission

This is called stop-and-wait protocol

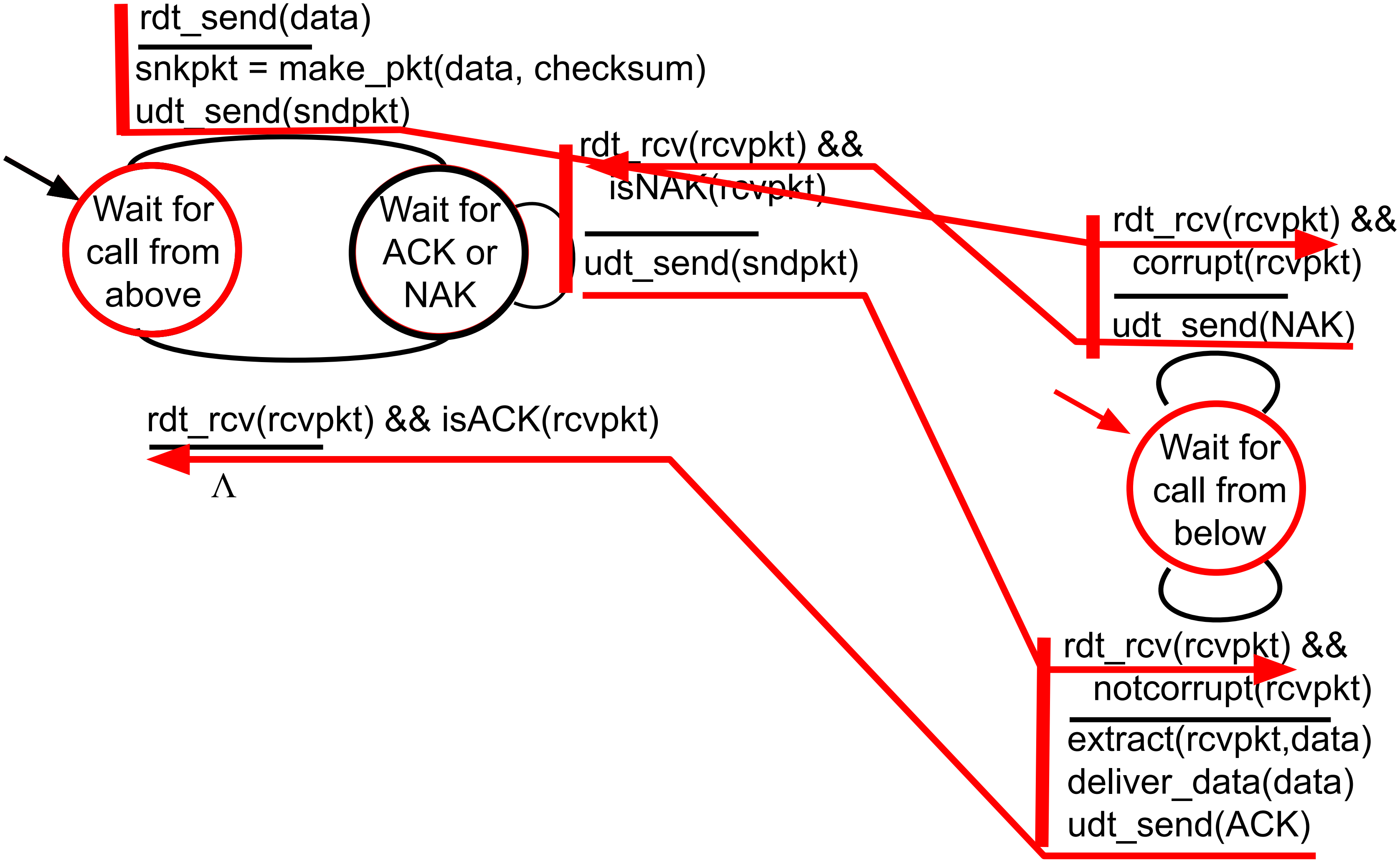
# rdt2.0: FSM specification



# rdt2.0: operation with no errors



# rdt2.0: error scenario



# rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- Retransmit?

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

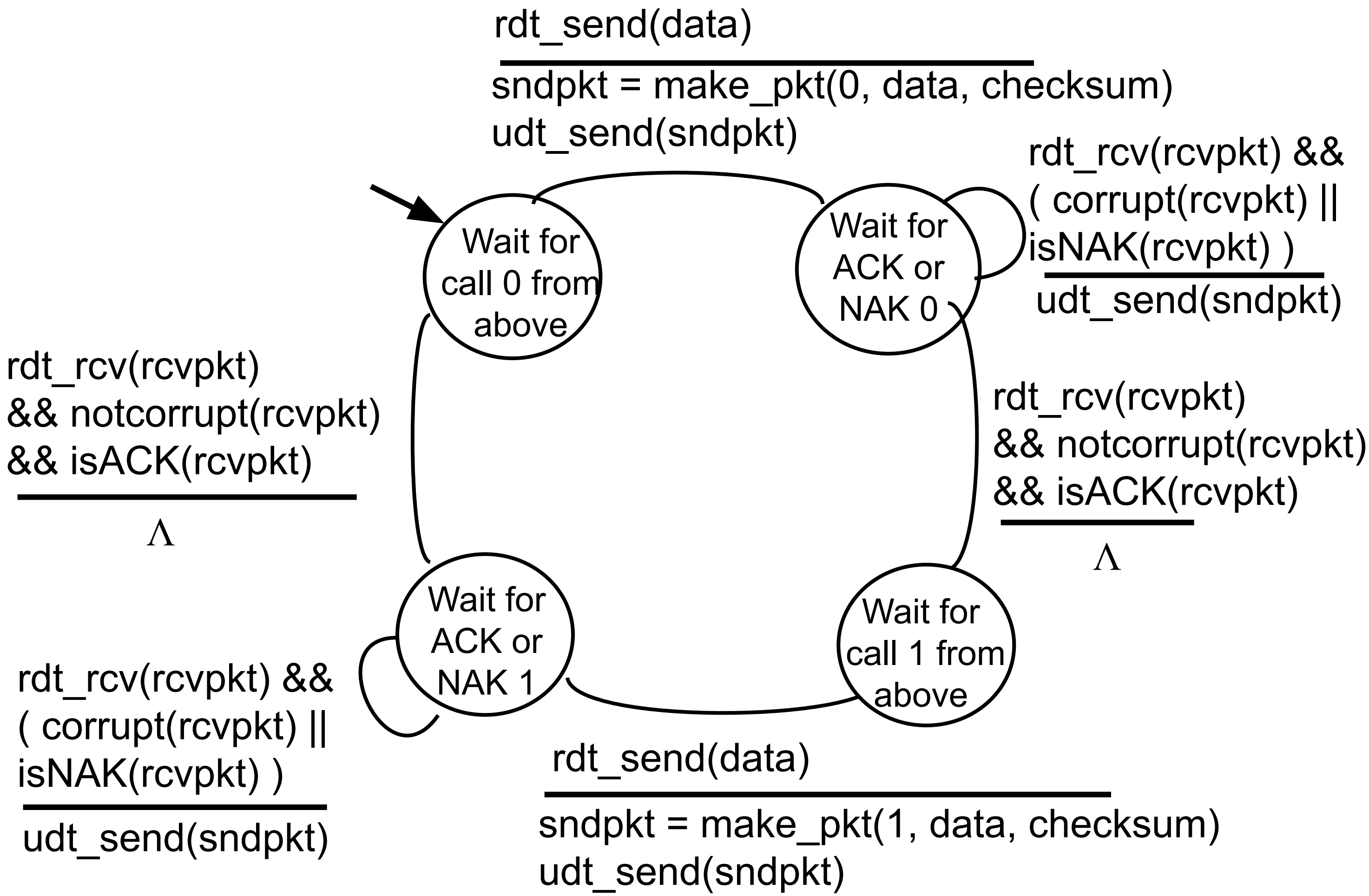
**stop and wait**

sender sends one packet, then waits for receiver response

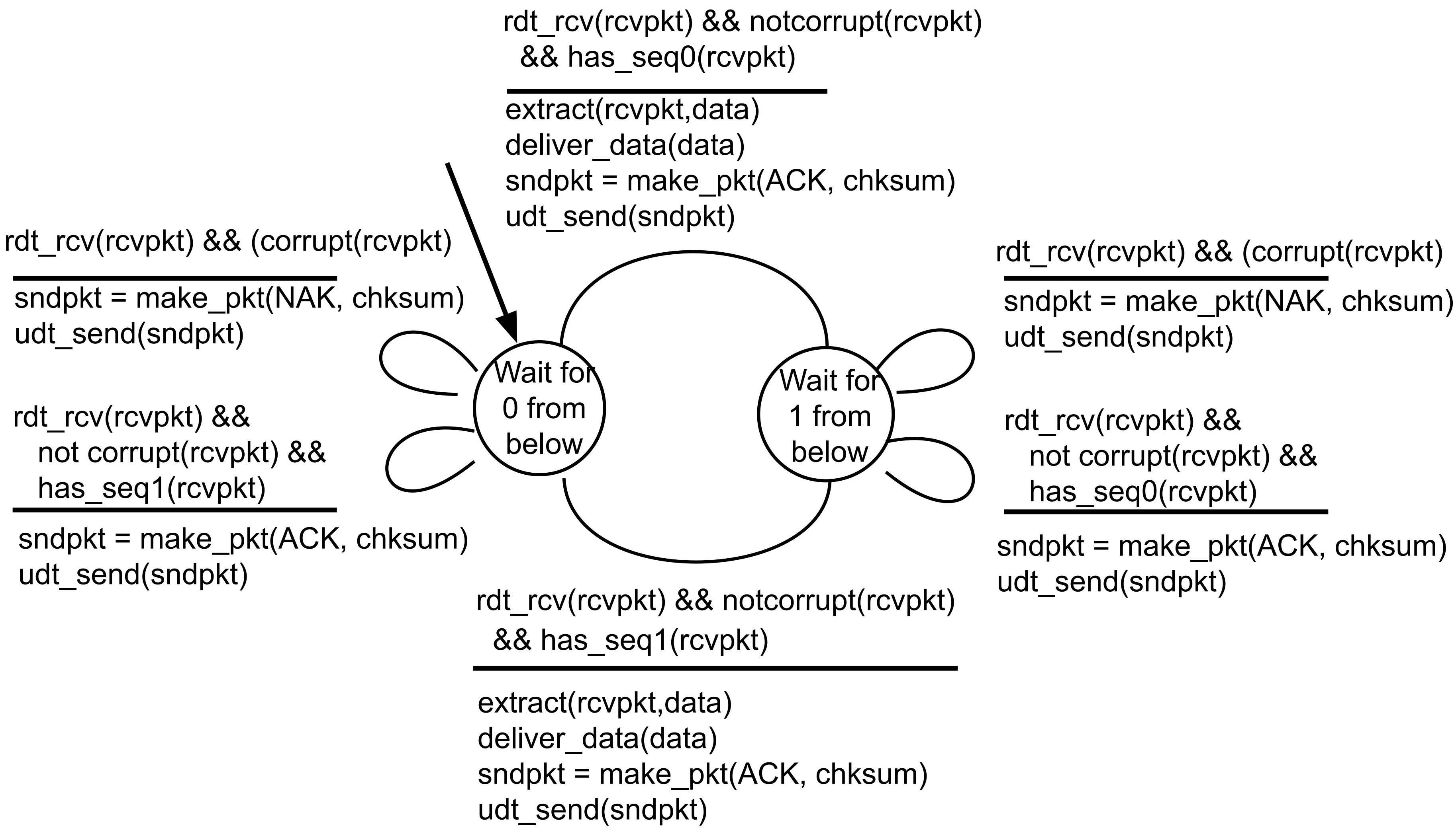
# rdt2.1: sender, handles garbled ACK/NAKs

- Resend packet when garbled ACK/NAK received
- Problem
  - Duplicates
- Solution
  - Sequence Number

# 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

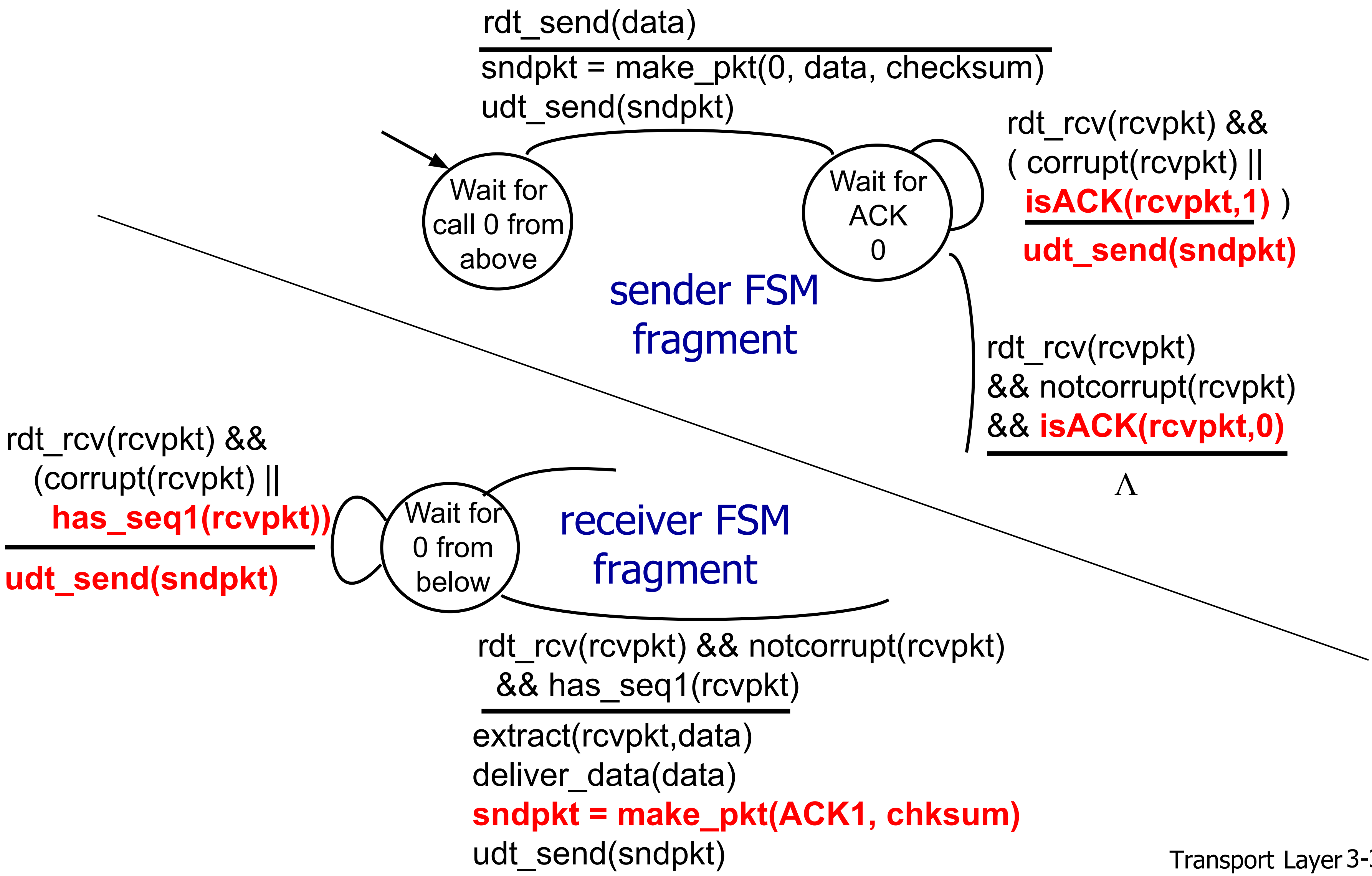
receiver:

- 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



# rdt3.0: channels with errors *and* loss

- Underlying Channel
  - Bit Errors
  - Packet loss
- Error Detection
  - checksum, seq. #, ACKs, retransmission
- Loss Detection
  - ?

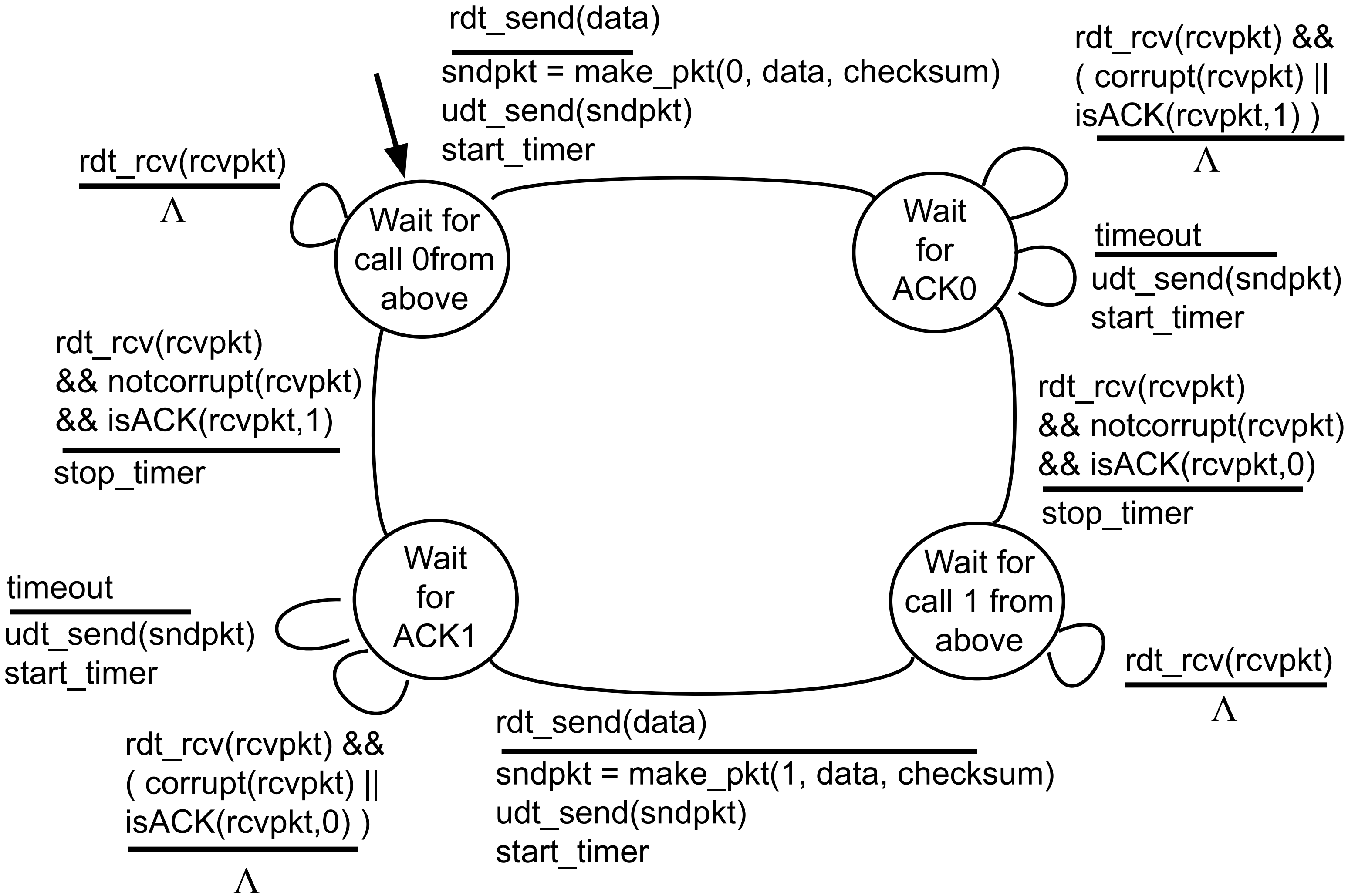
# rdt3.0: how to detect packet loss?

- Sender waits "reasonable" amount of time for ACK
- Retransmits if no ACK received in this time

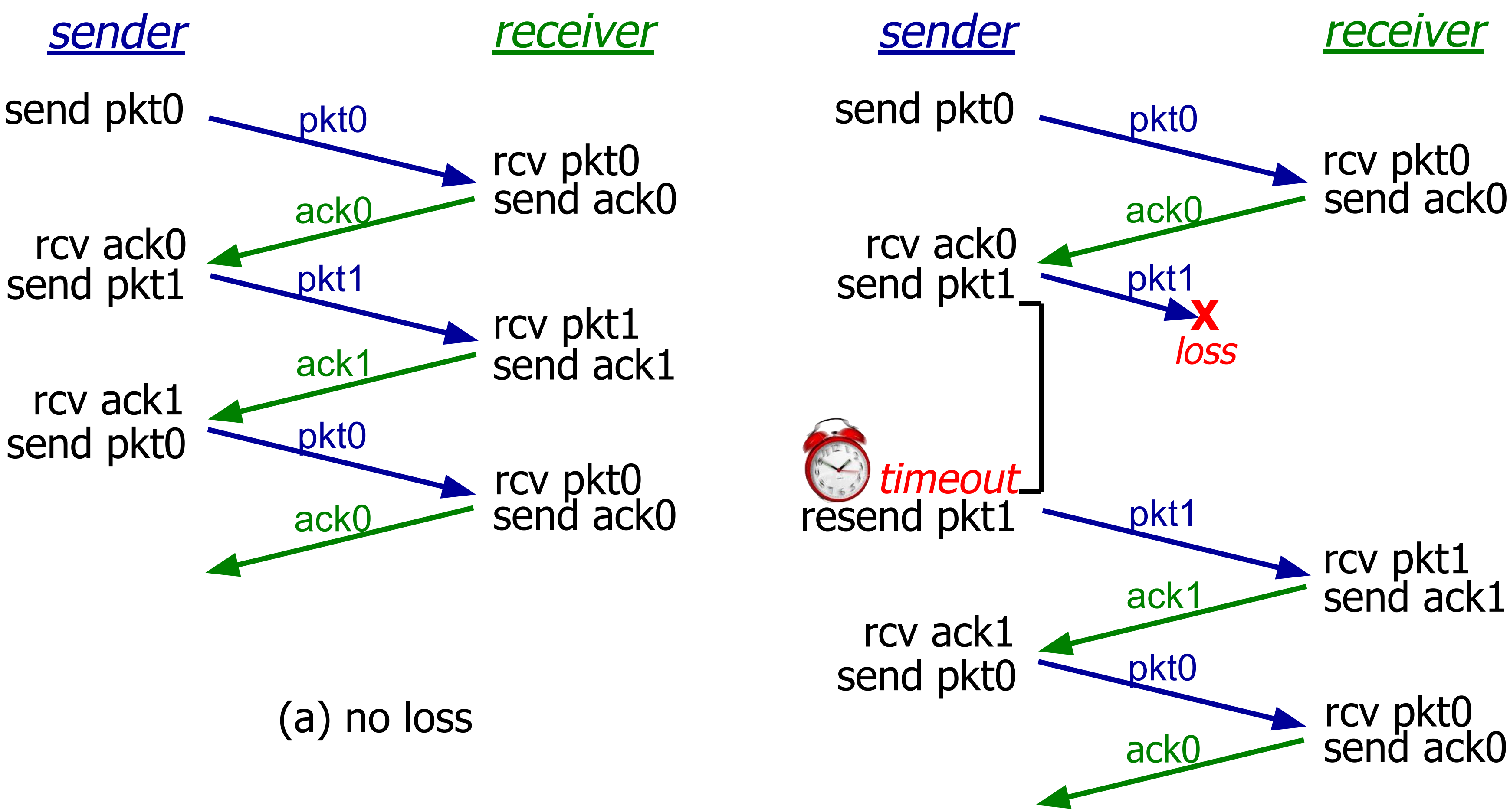
- Countdown Timer

- What if a packet is just delayed?
  - Duplicates possible

# rdt3.0 sender



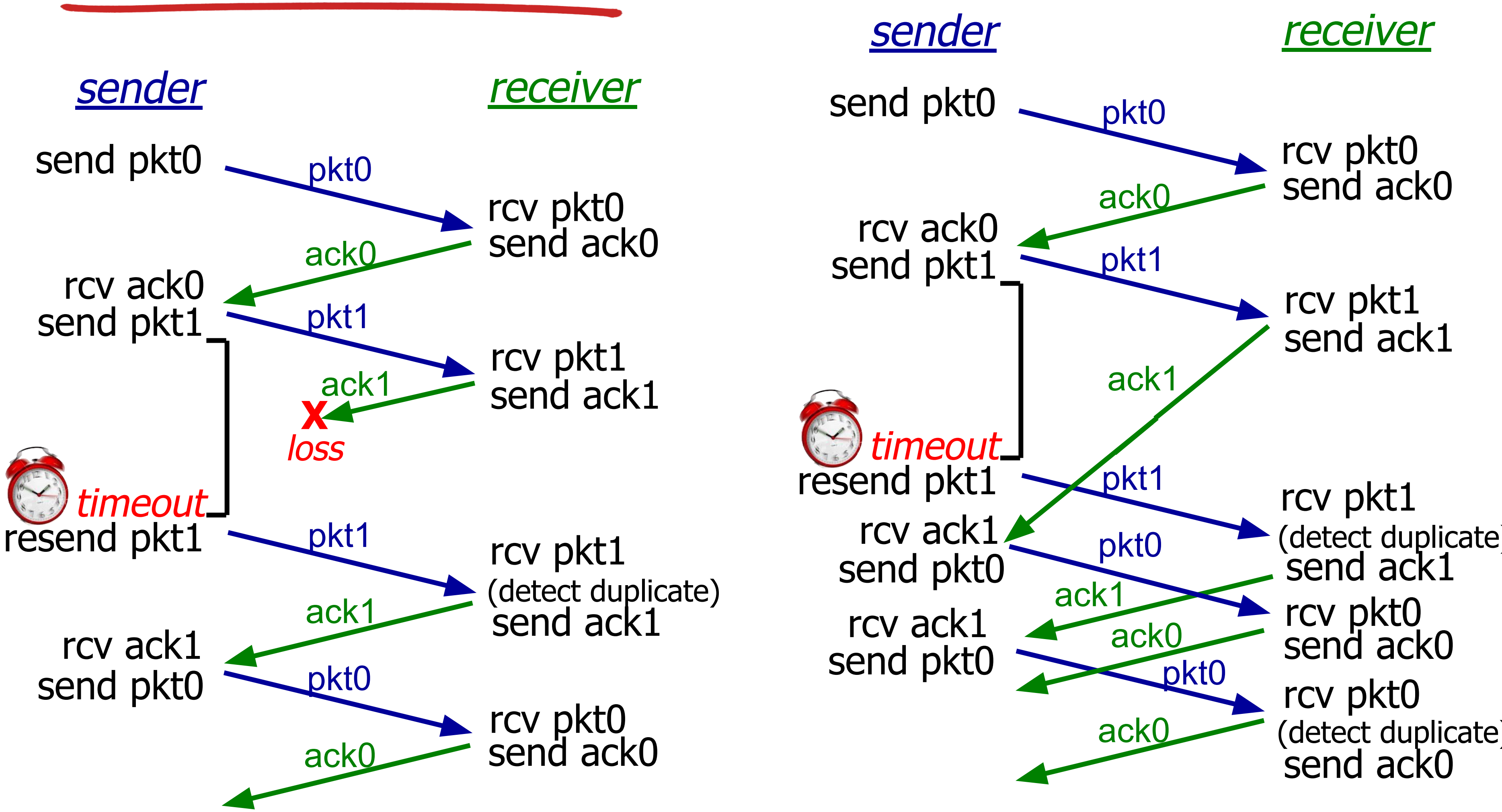
# rdt3.0 in action




(a) no loss

(b) packet loss

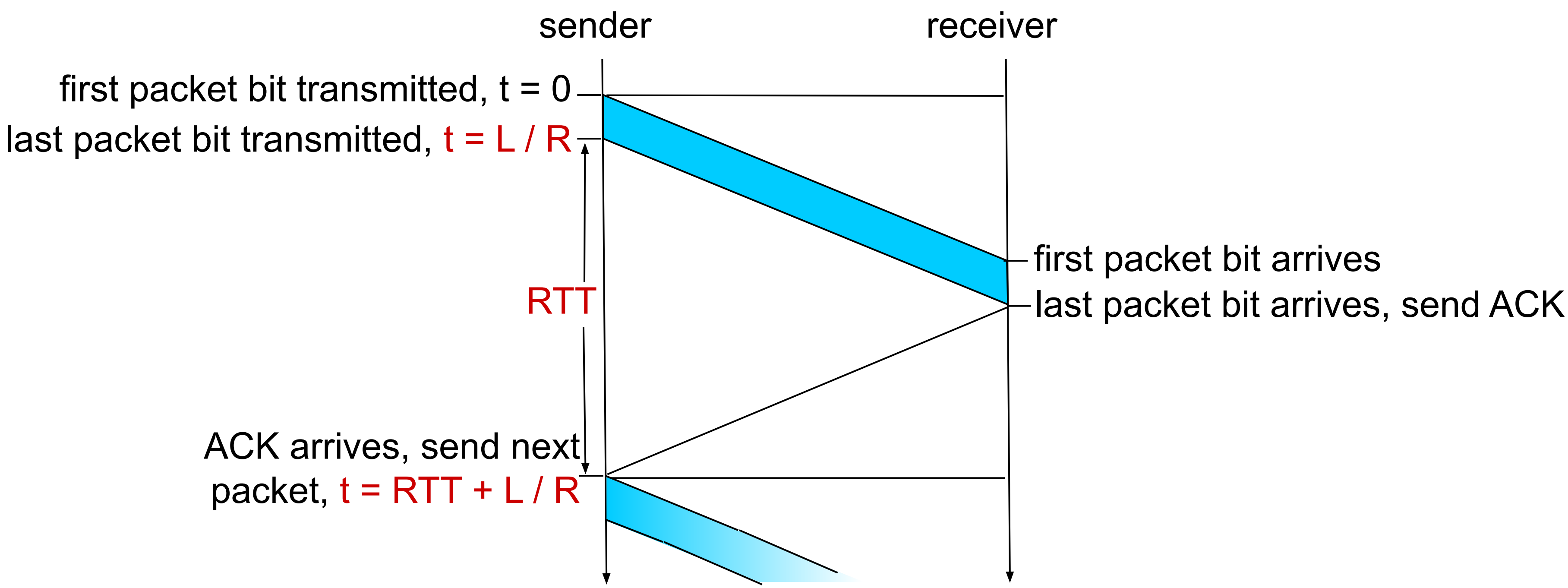
# rdt3.0 in action




(c) ACK loss

(d) premature timeout/ delayed ACK

# rdt3.0: stop-and-wait operation



# Performance of rdt3.0 (example)

- 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

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

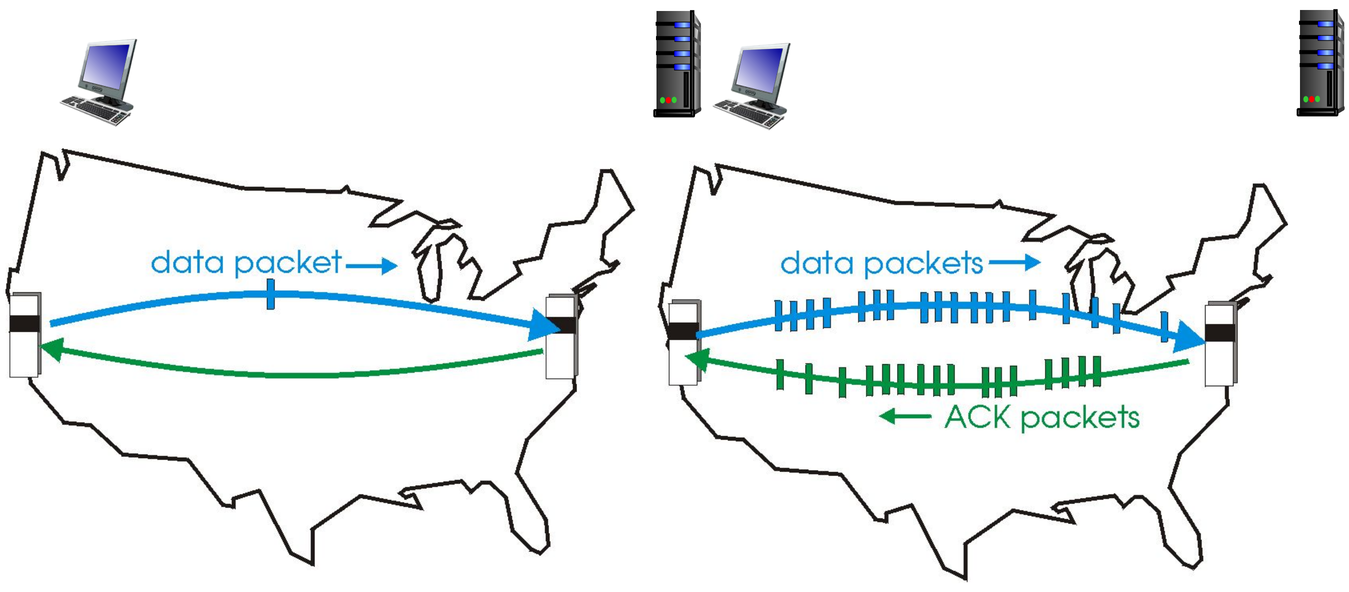
- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link

- network protocol limits use of physical resources!

# Pipelined protocols

- Multiple, "in-flight", yet-to-be-acknowledged pkts
  - range of Seq.#
  - buffering at sender and/or receiver

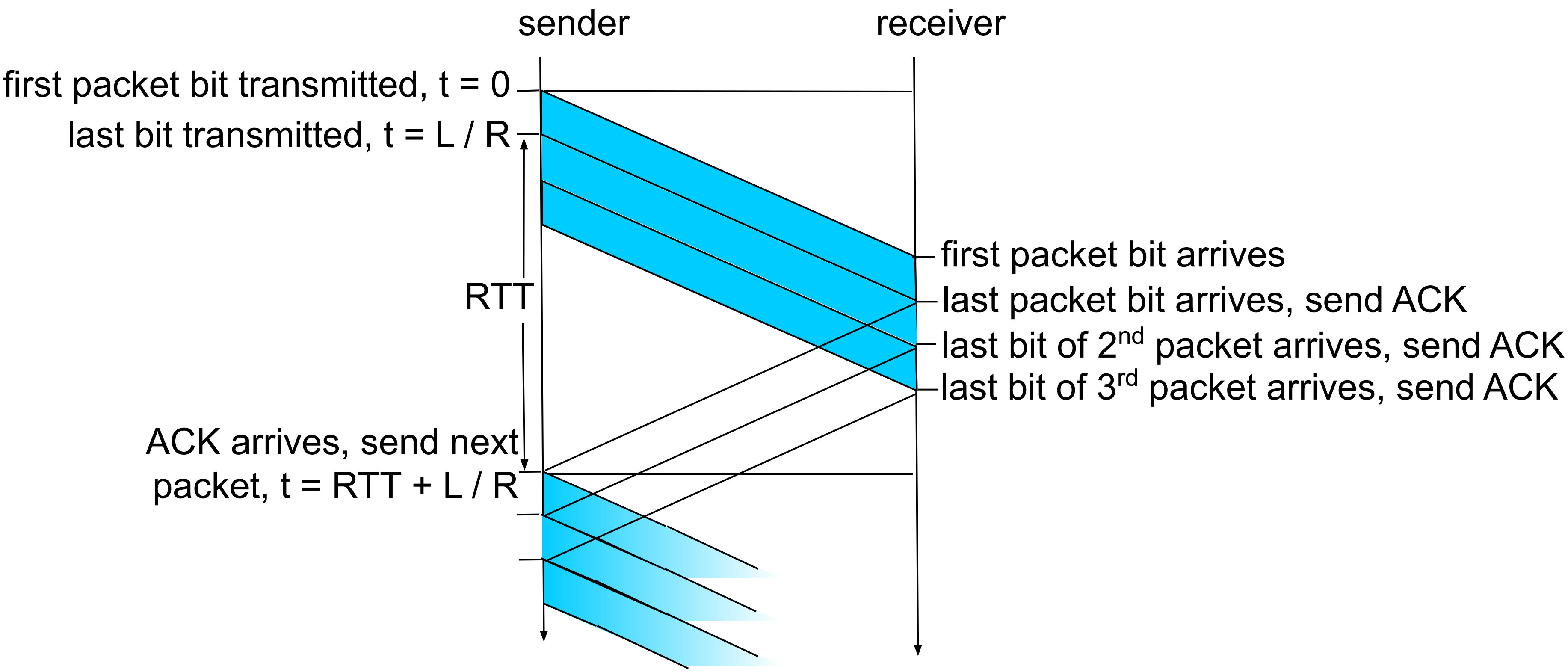
- 2 generic forms of pipelined protocols:
  - *go-Back-N,*
  - *selective repeat*




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

(b) a pipelined protocol in operation

# Pipelining: increased utilization



# Pipelined protocols: overview

## Go-back-N:

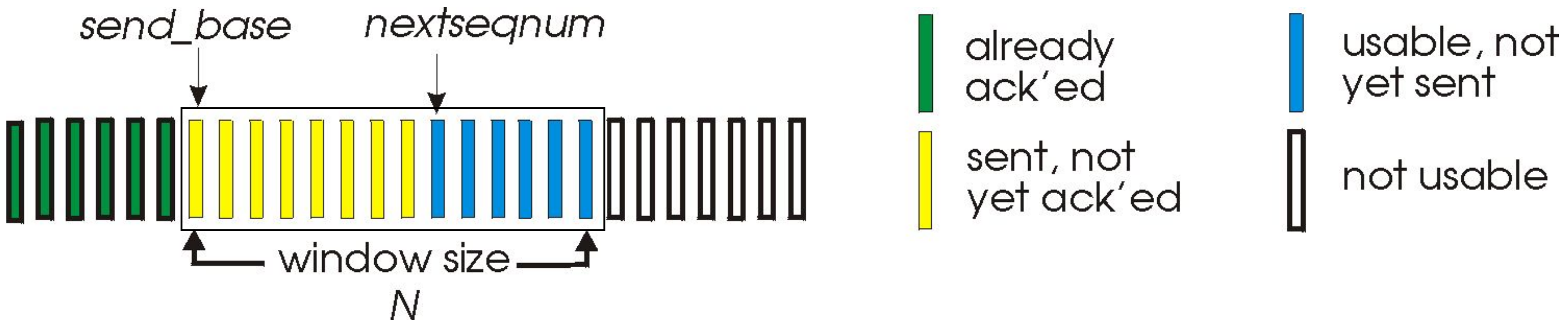
- sender
  - can have up to N unACKed packets in pipeline
- receiver
  - only sends *cumulative ACK*
  - doesn't ACK packet if there's a gap
- sender
  - has timer for oldest unACKed packet
  - when timer expires, retransmit *all* unACKed packets

## Selective Repeat:

- sender
  - can have up to N unACKed packets in pipeline
- receiver
  - sends *individual ACK* for each packet
- sender
  - maintains timer for each unACKed packet
  - when timer expires, retransmit only that unACKed packet

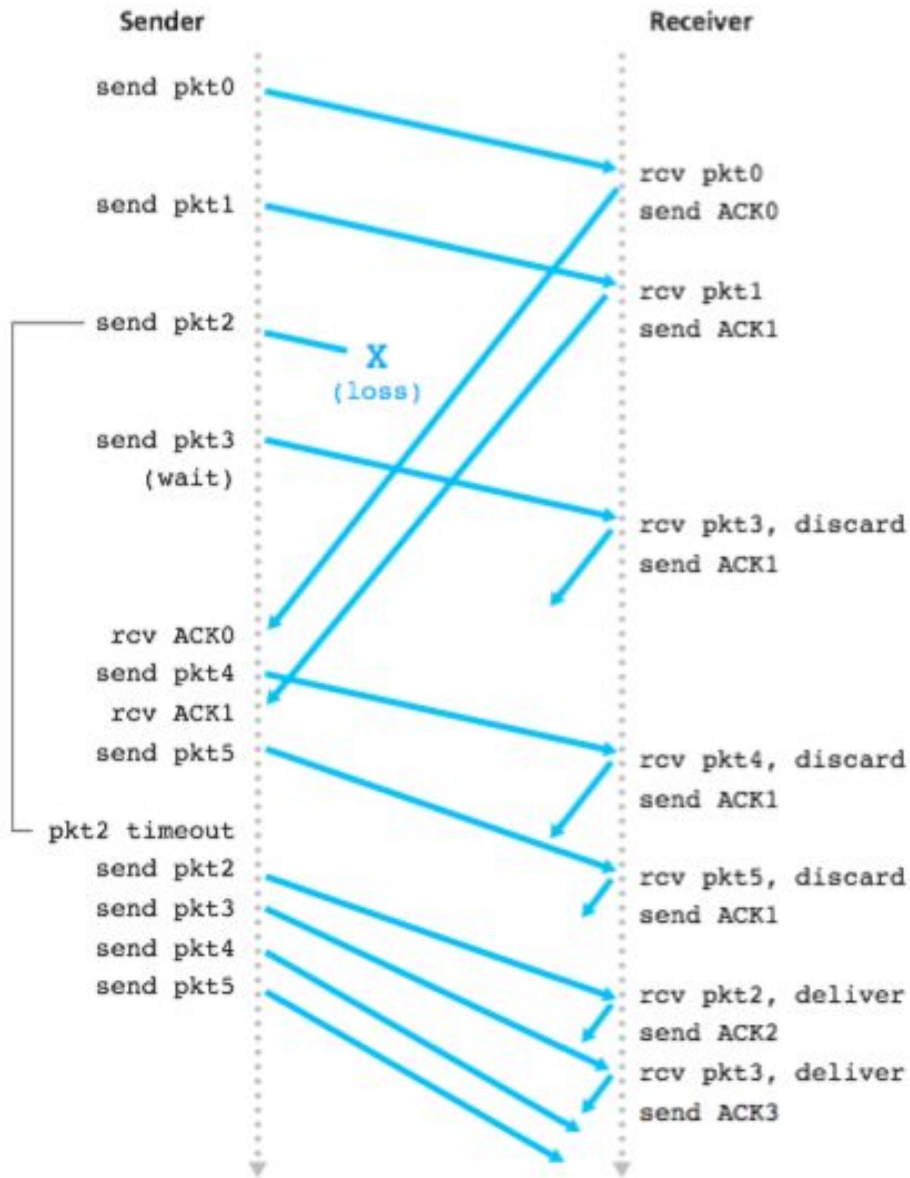
# Go-Back-N: sender

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

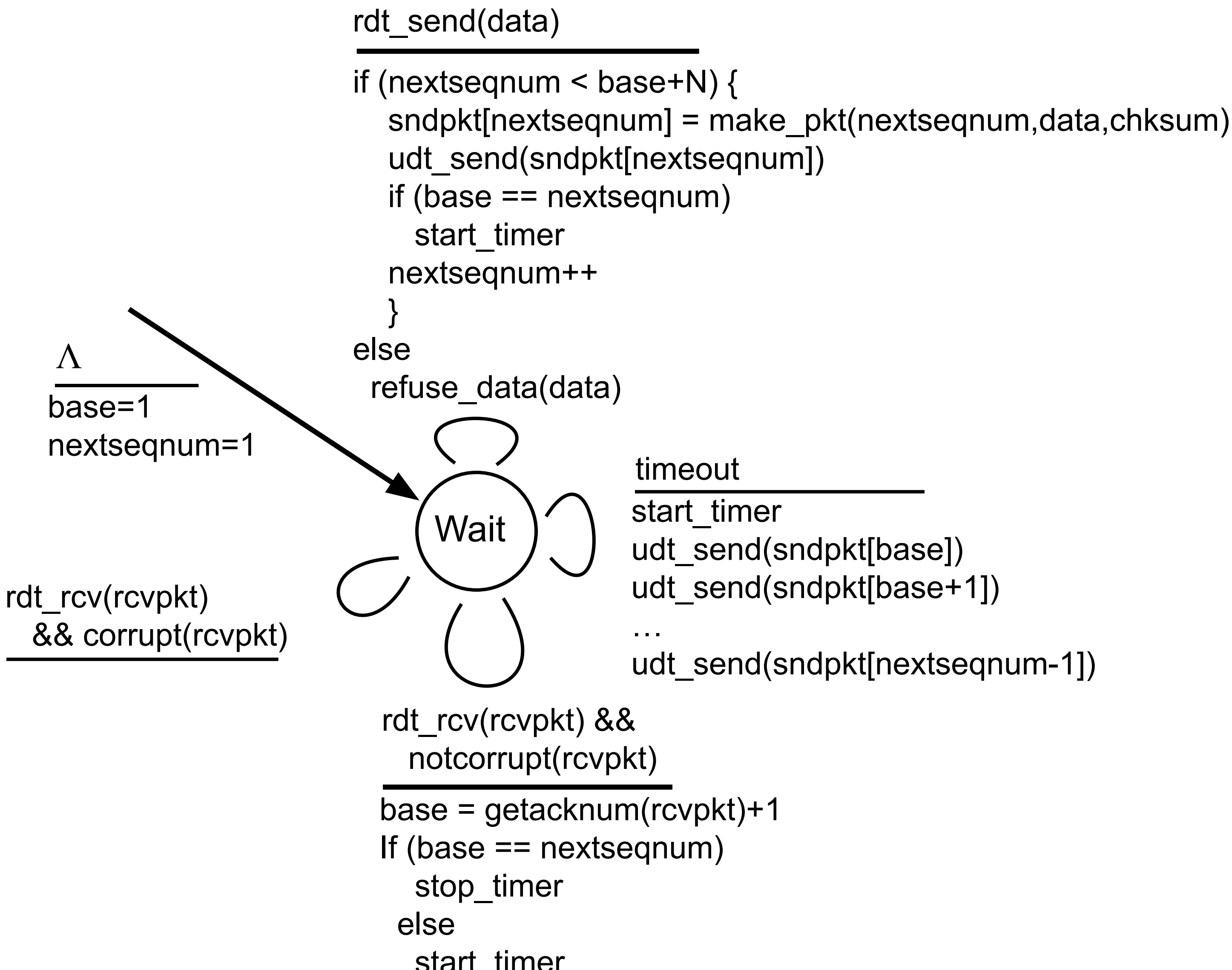



- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

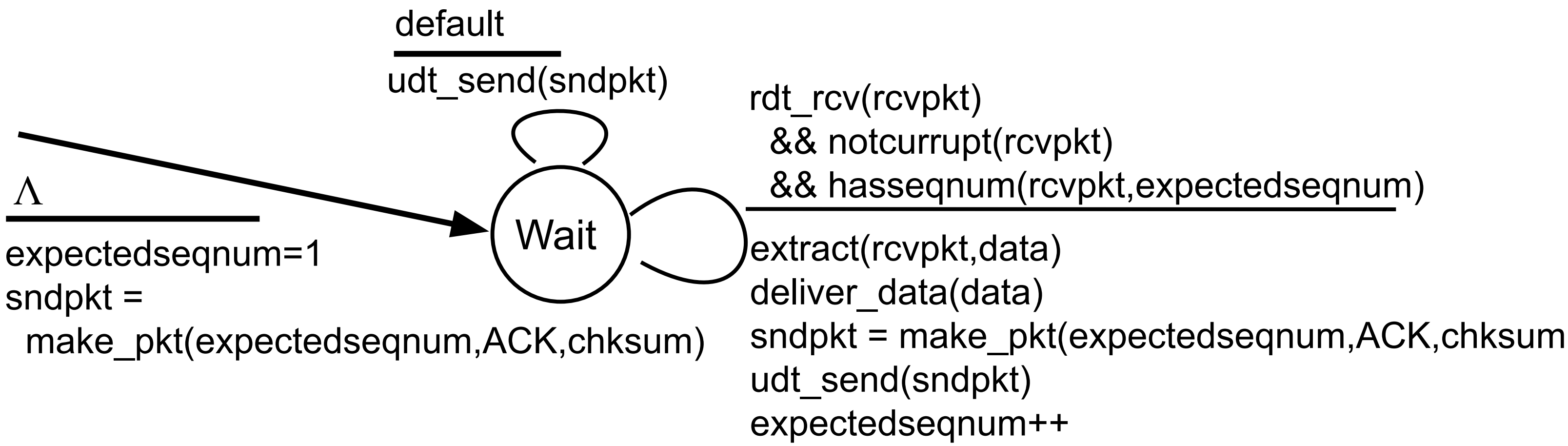
# Go-Back-N


Sender
Receiver
send pkt0
send pkt1
send pkt2
X
(loss)
send pkt3
(wait)
rcv ACK0
send pkt4
rcv ACK1
send pkt5
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

# GBN: sender extended FSM



# GBN: receiver extended FSM

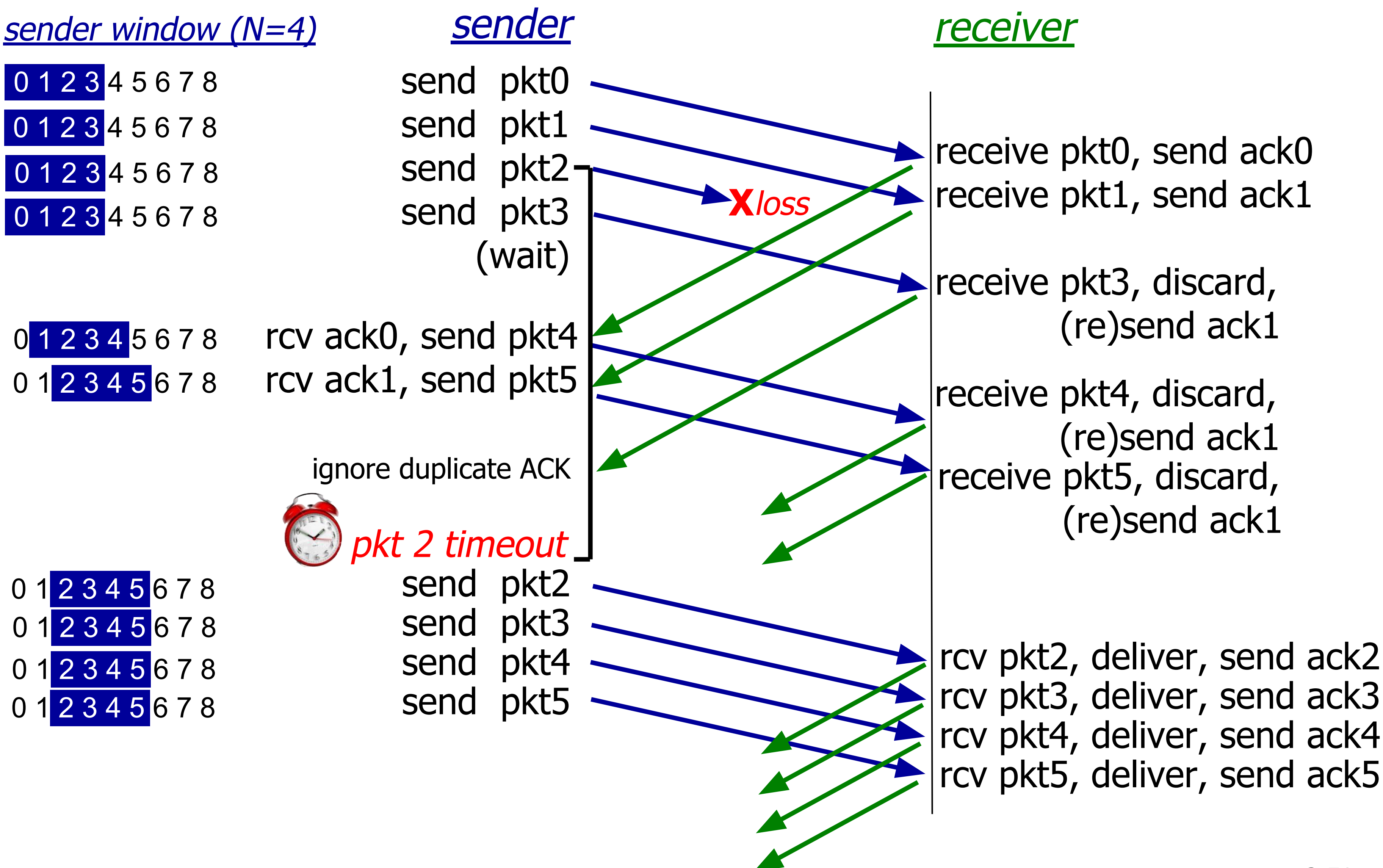



ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

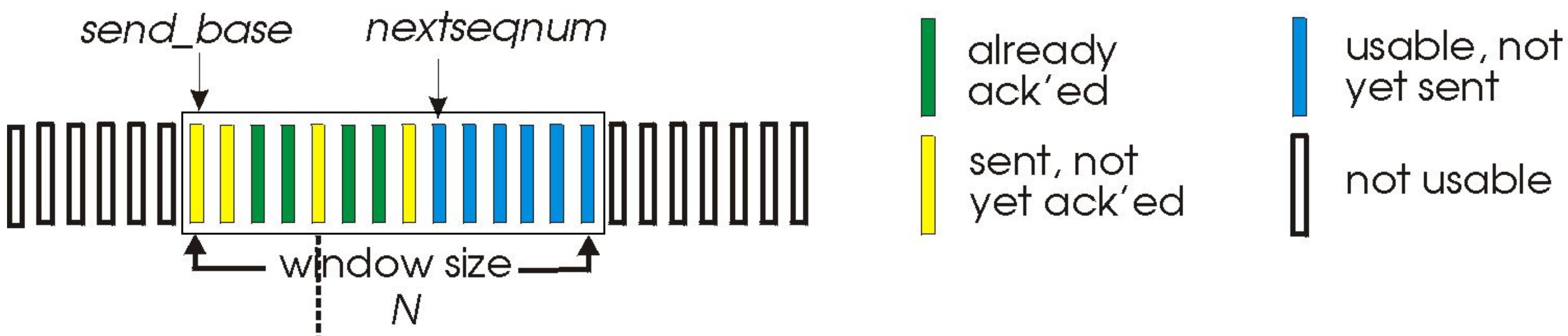
- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

- out-of-order pkt:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

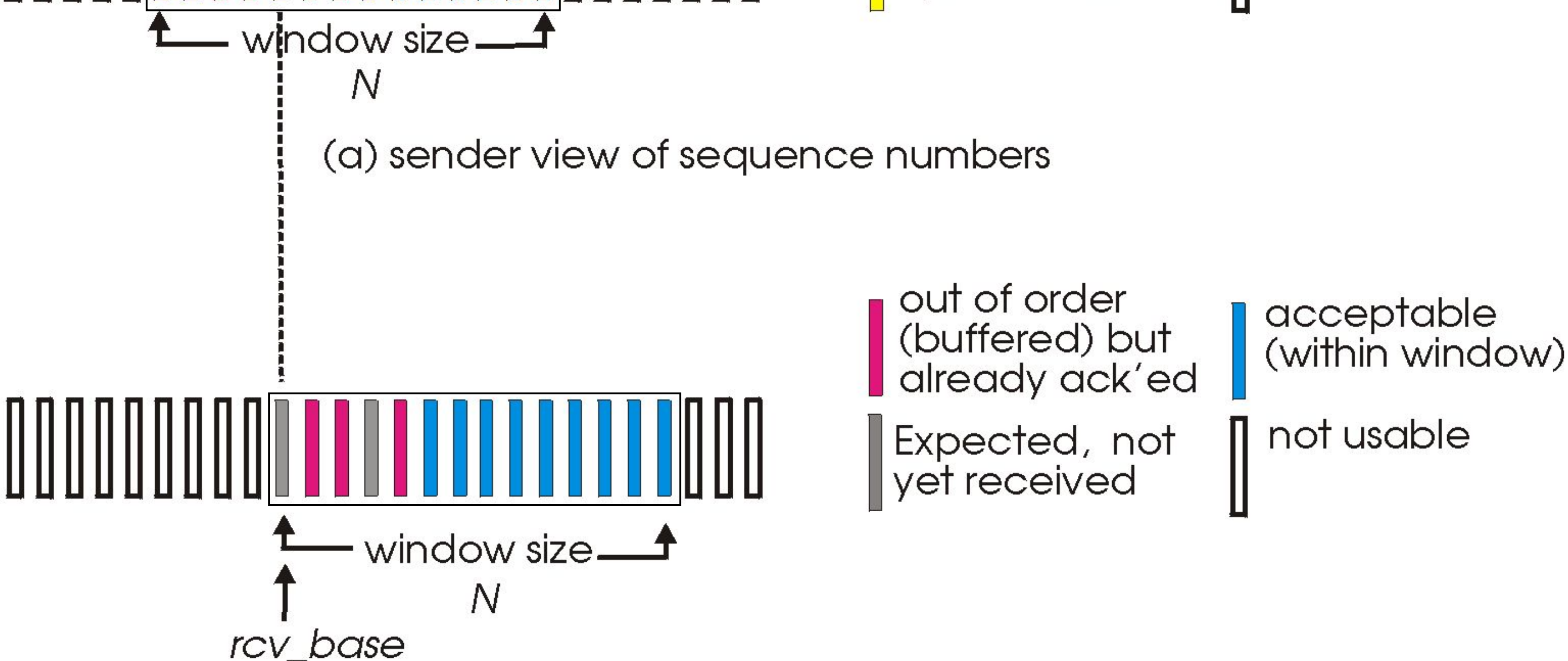
# GBN in action



# Selective repeat: sender, receiver windows




(a) sender view of sequence numbers




(b) receiver view of sequence numbers

# Selective repeat

## sender

**data from above:**

- if next available seq # in window, send pkt

**timeout(n):**

- resend pkt n, restart timer

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

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

## receiver

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

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

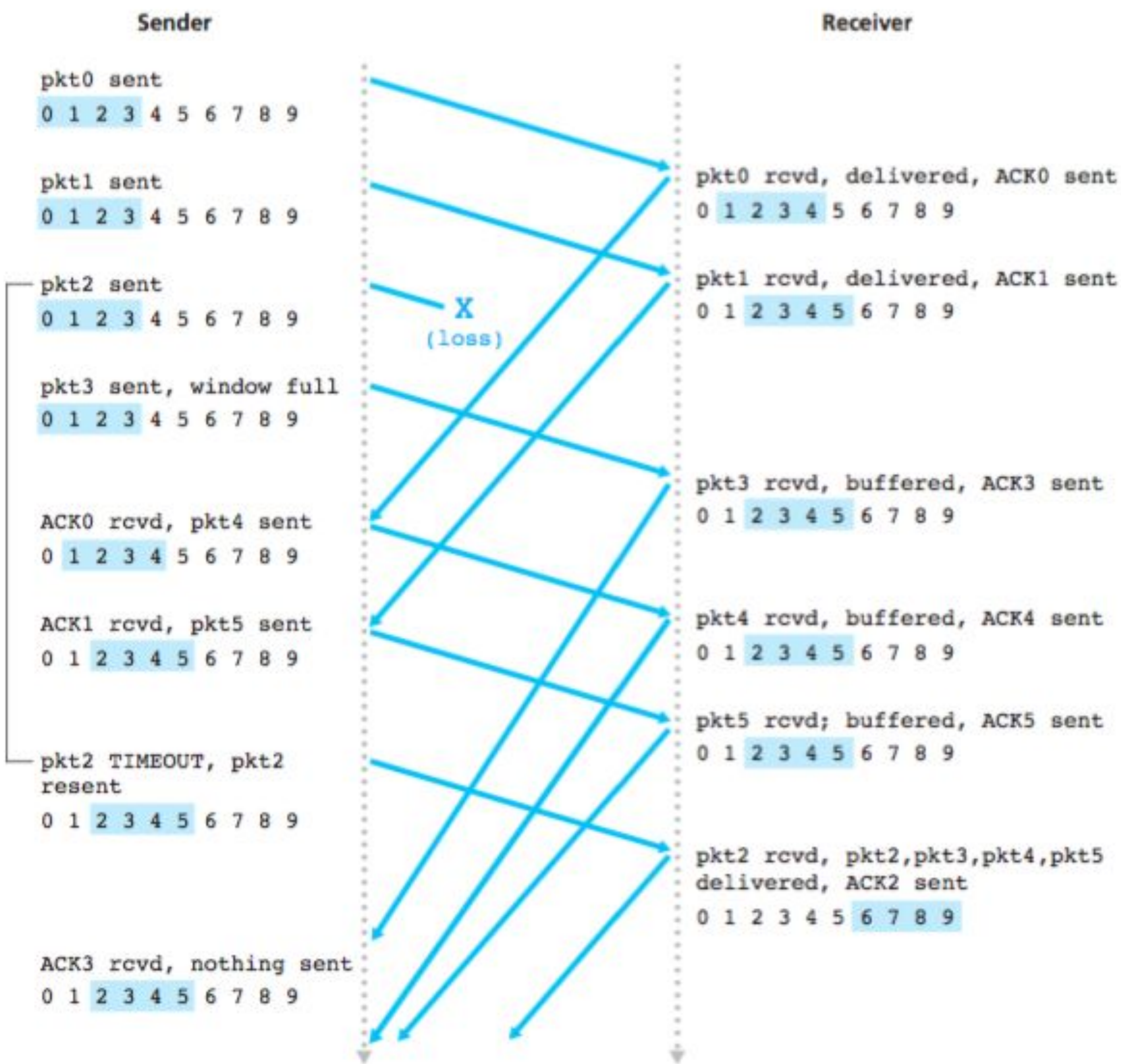
**pkt n in** [rcvbase-N,rcvbase-1]

- ACK(n)

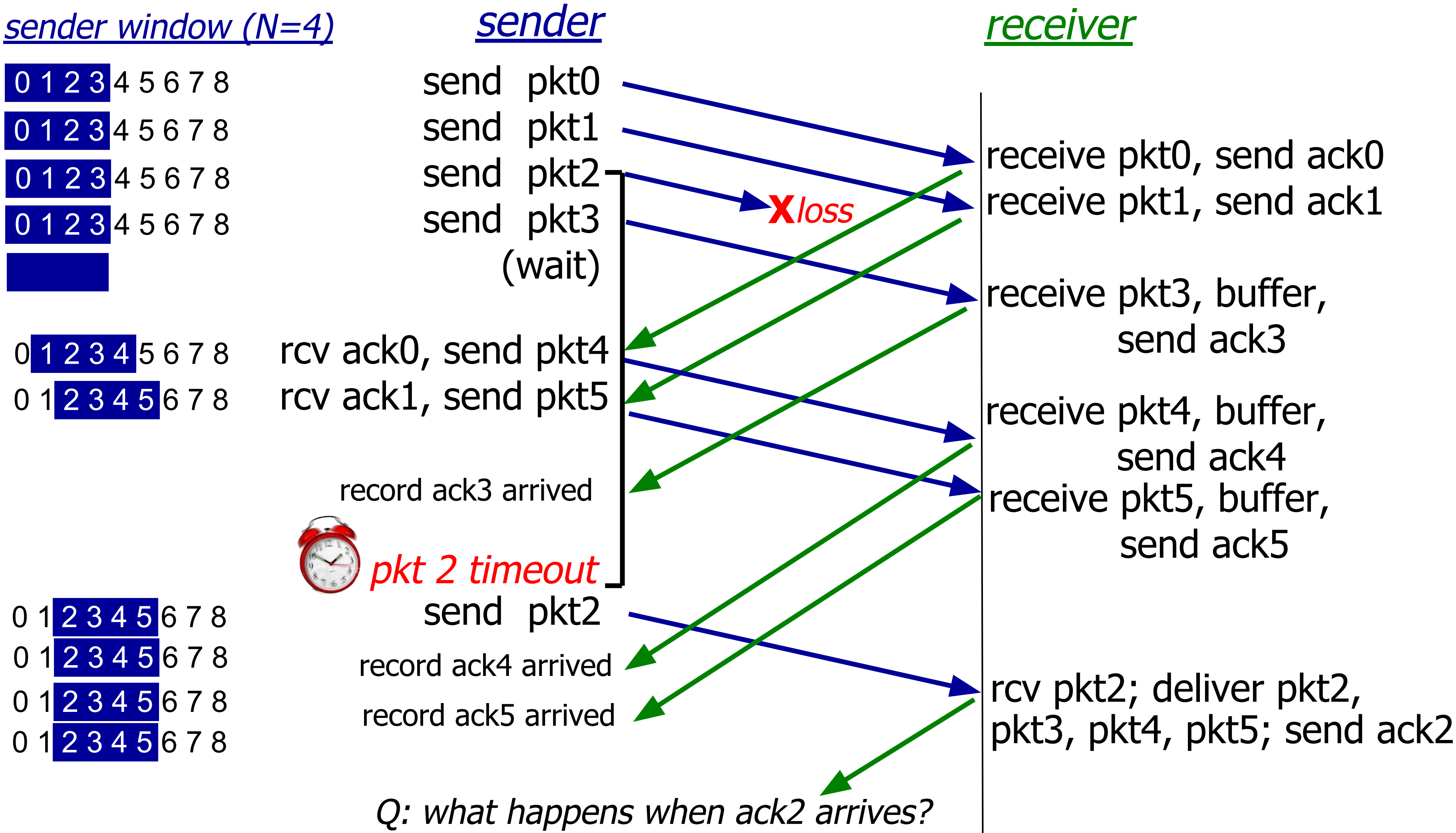
**otherwise:**

- ignore

# Selective repeat in action



# Selective repeat in action

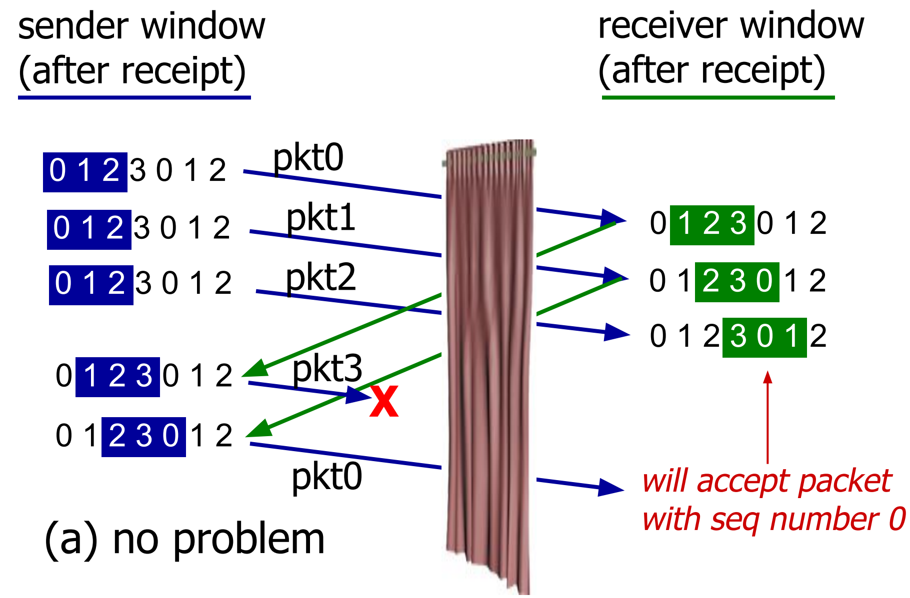


# Selective repeat: dilemma

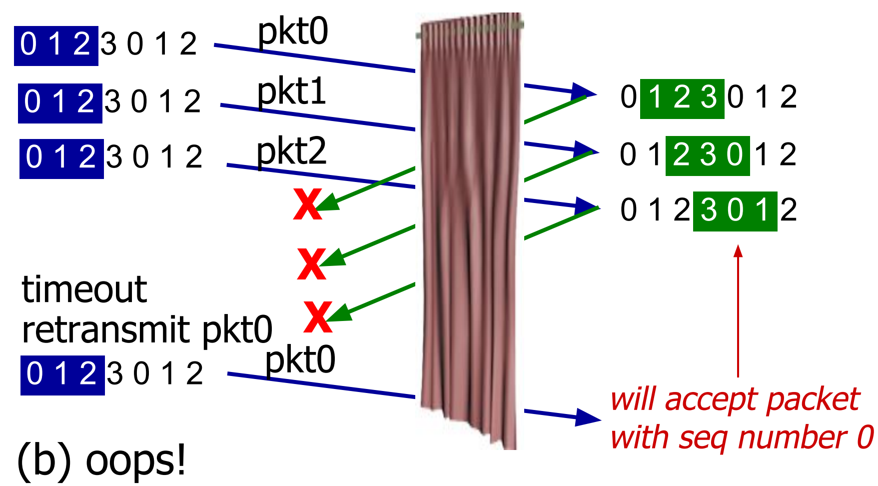
example:

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

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




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



# Transport Layer

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

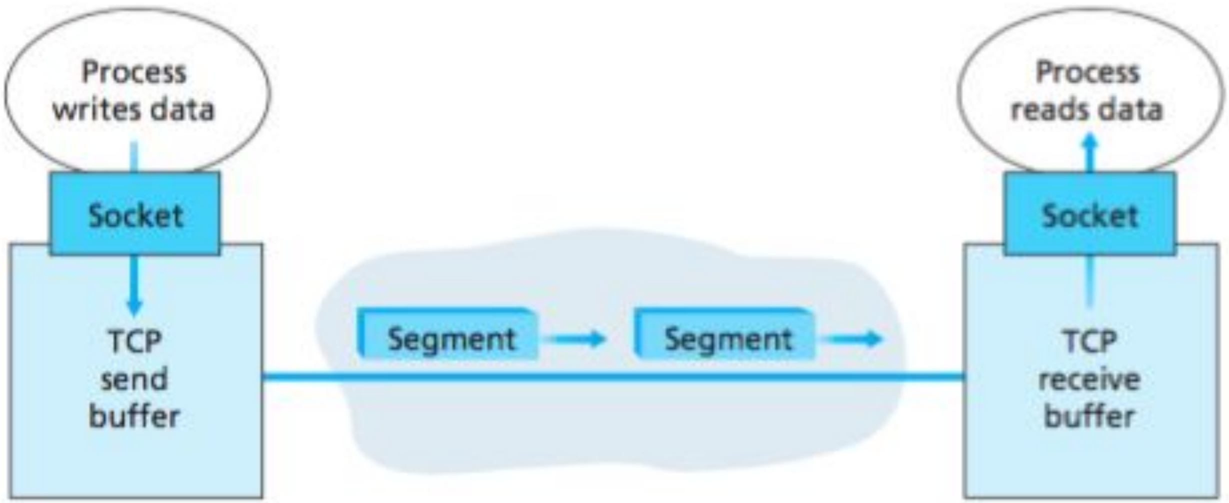
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

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

- point-to-point (unicast)
- reliable, in-order *byte steam*
- pipelined
- full duplex data
- connection-oriented
- flow controlled
- congestion control

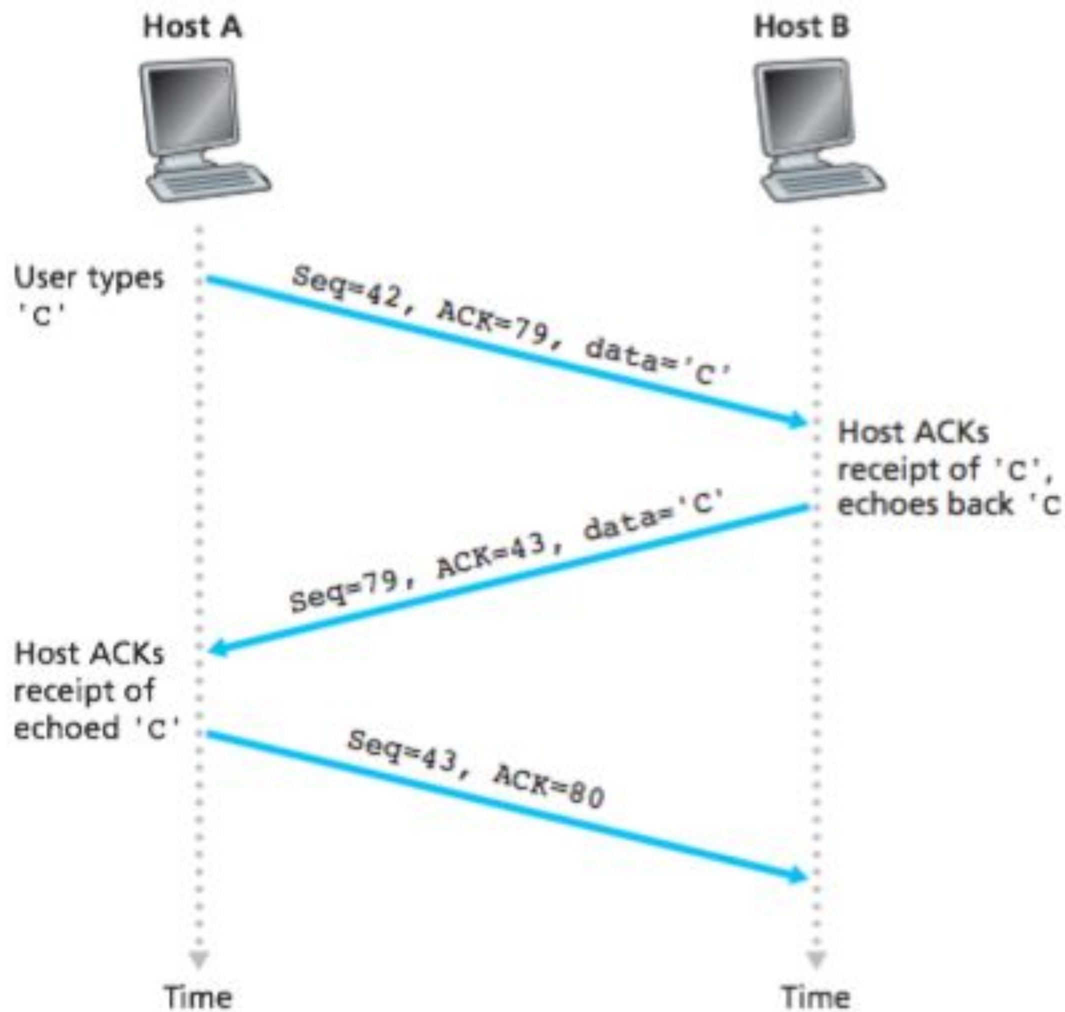


# TCP Seq #'s and ACKs

- Seq #'s

- ACKs

- Out-of-order segments?



# TCP round trip time, timeout

**Q:** how to set TCP timeout value?

- *too short?*
- *too long?*

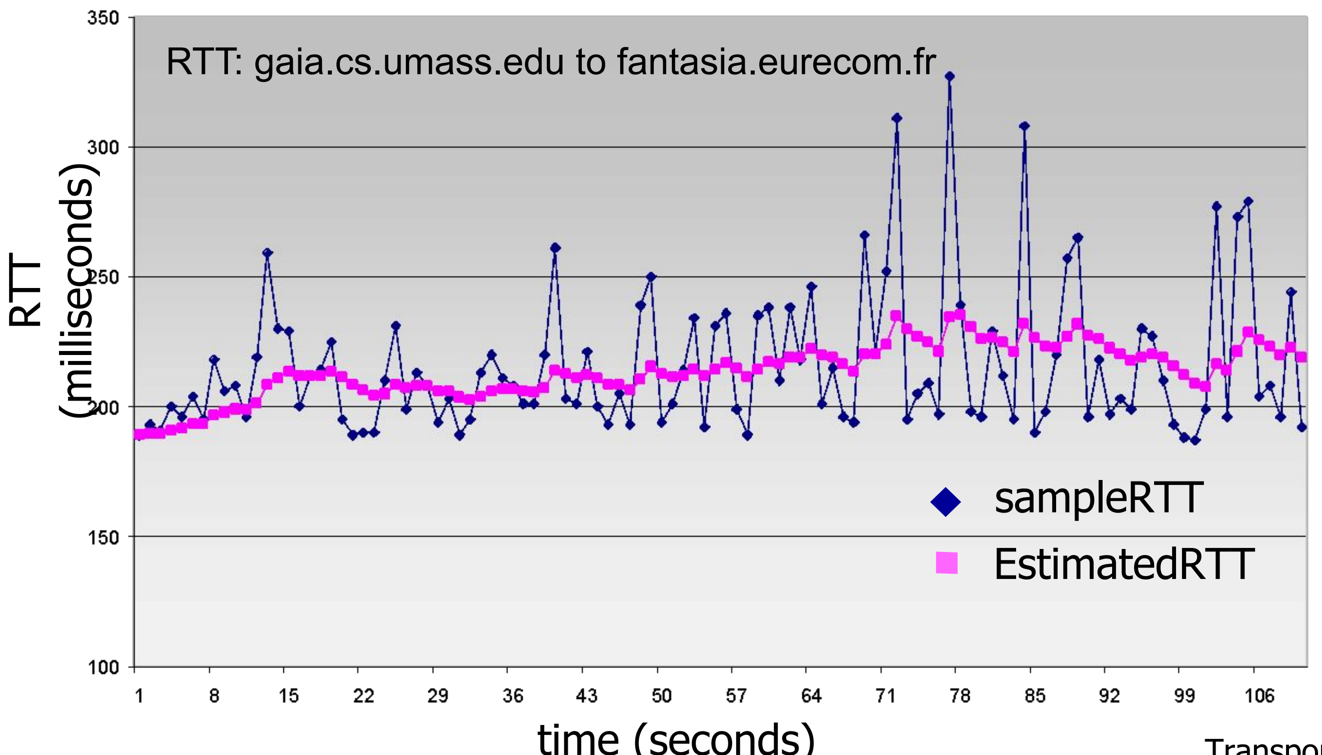
**Q:** how to estimate RTT?

- `SampleRTT?`
- `AverageRTT?`

# TCP round trip time, timeout

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

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



# TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** + "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"

# Transport Layer

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 rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
  - retransmissions triggered by:
    - Timeout events
    - Duplicate ACKs

# TCP sender events:

1. *Data rcvd from app*
2. *Timeout*
3. *ACK rcvd*

# TCP sender events:

*1. 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`** `(based on EstimatedRTT)`

# TCP sender events:

*2. Timeout:*

- retransmit segment that caused timeout
- restart timer

# TCP sender events:

*3. 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

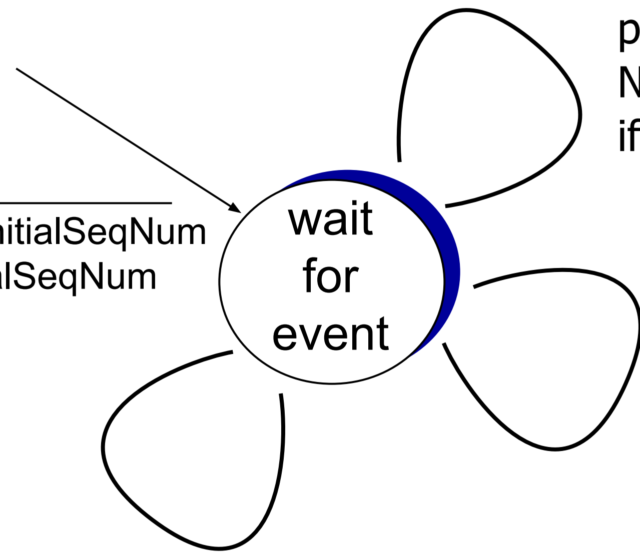
---

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

$\Lambda$

---

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum




timeout

---

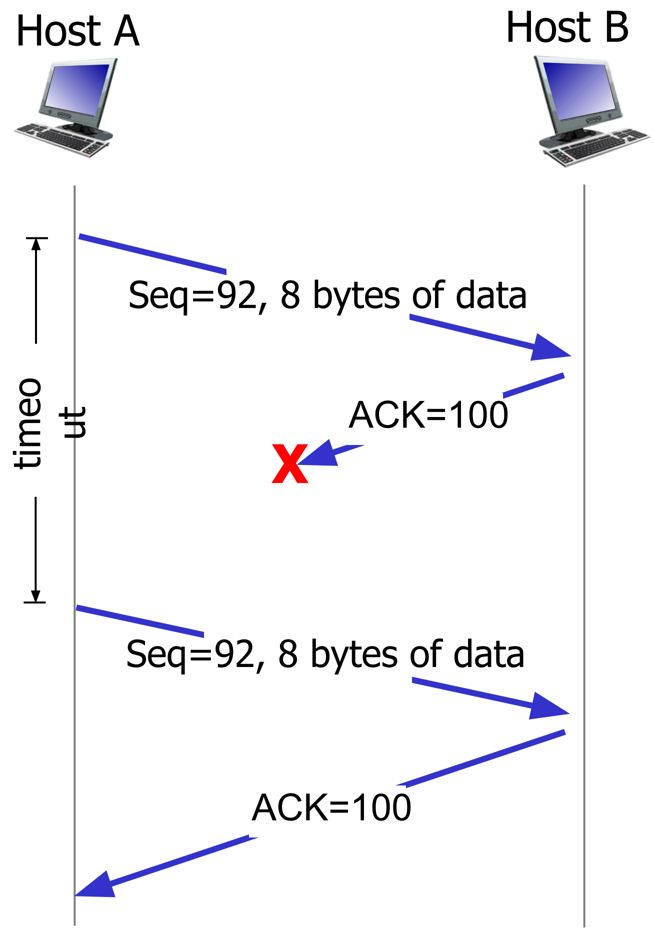
retransmit not-yet-acked segment
    with smallest seq. #
start timer

ACK received, with ACK field value y

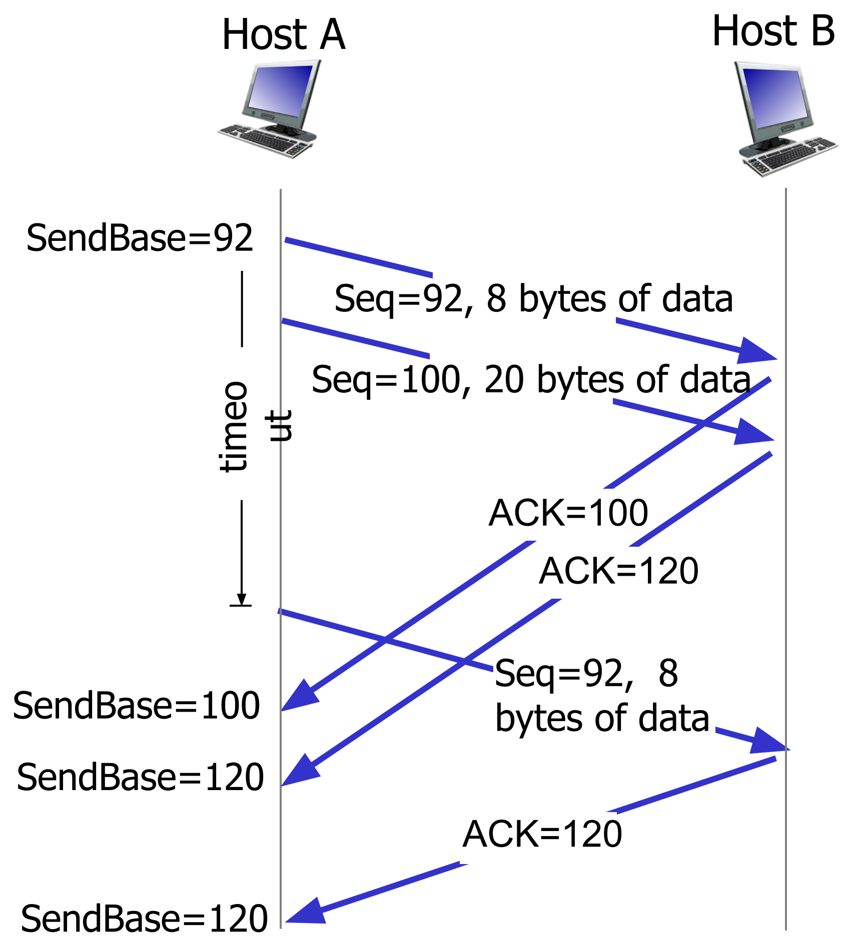
---

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

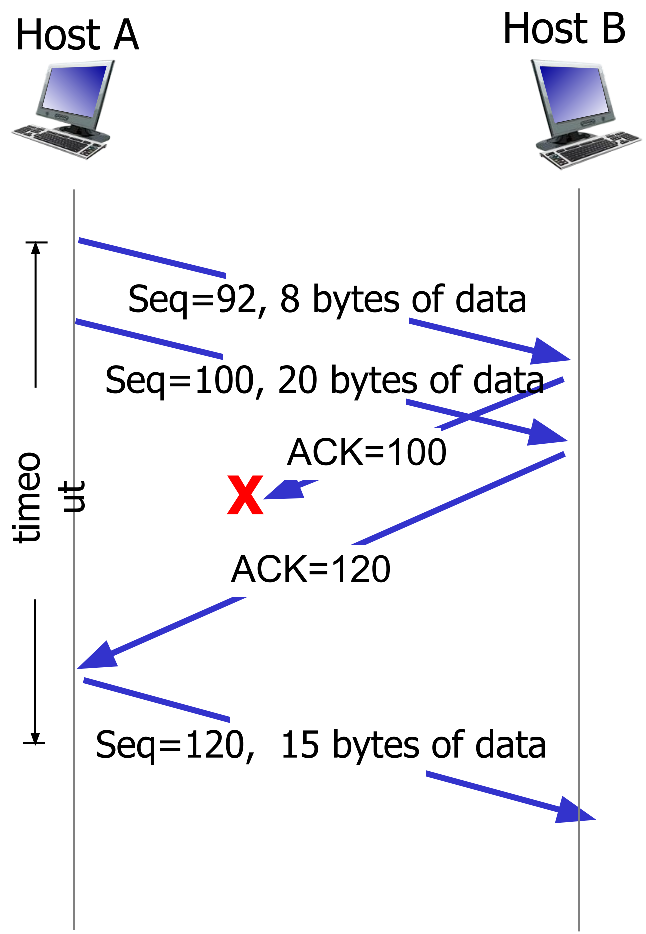
# TCP: lost ACK scenario



# TCP: premature timeout



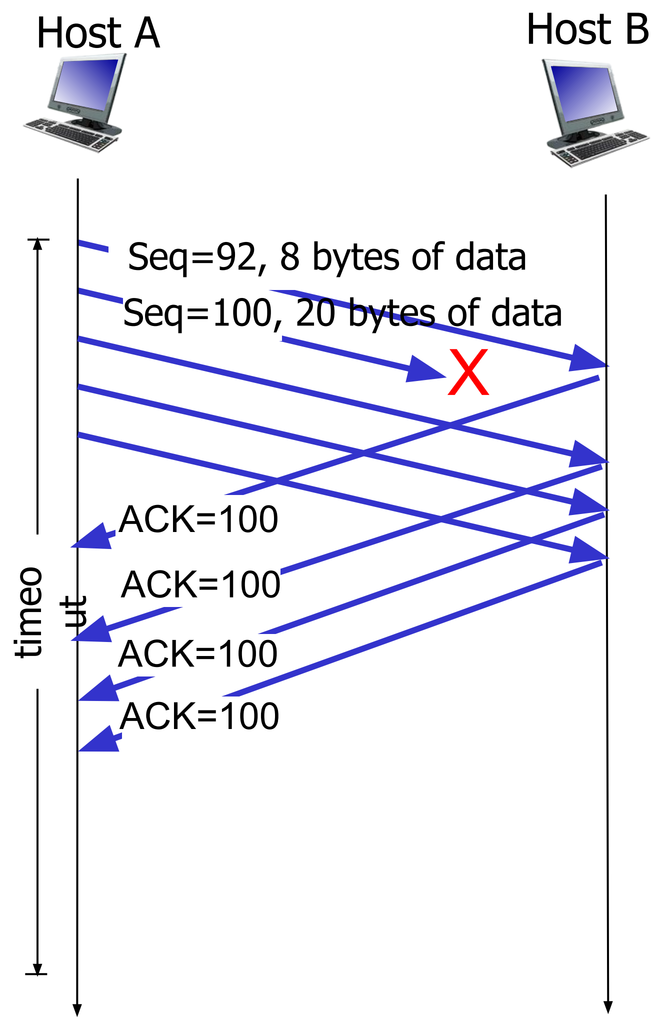
# TCP: cumulative ACK



# 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




fast retransmit after sender receipt of triple duplicate ACK

# 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*

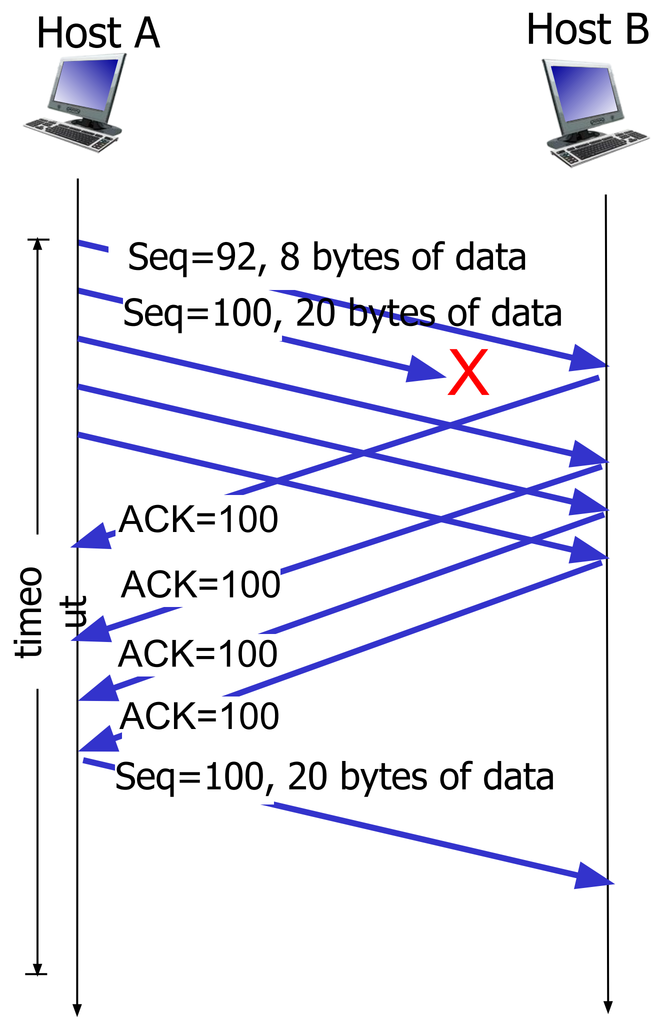
if sender receives 3 ACKs for same data

("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

# Transport Layer

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

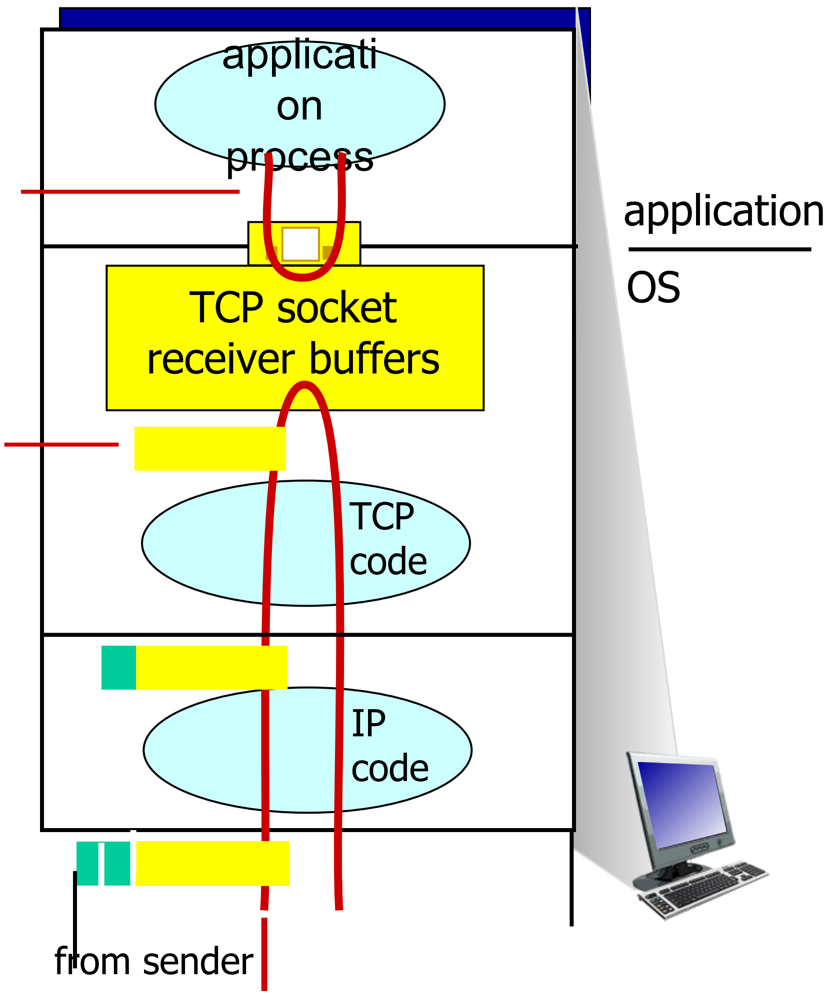
3.6 principles of congestion control

3.7 TCP congestion control

# TCP flow control

application may remove data from TCP socket buffers ….

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



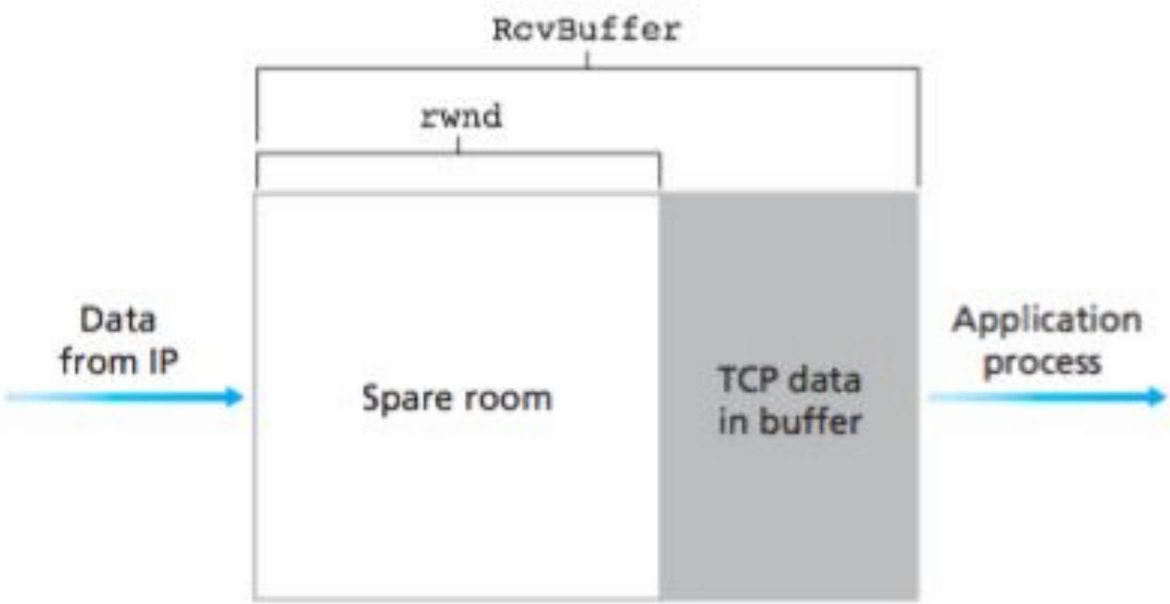

receiver protocol stack

***flow control***

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

# TCP flow control

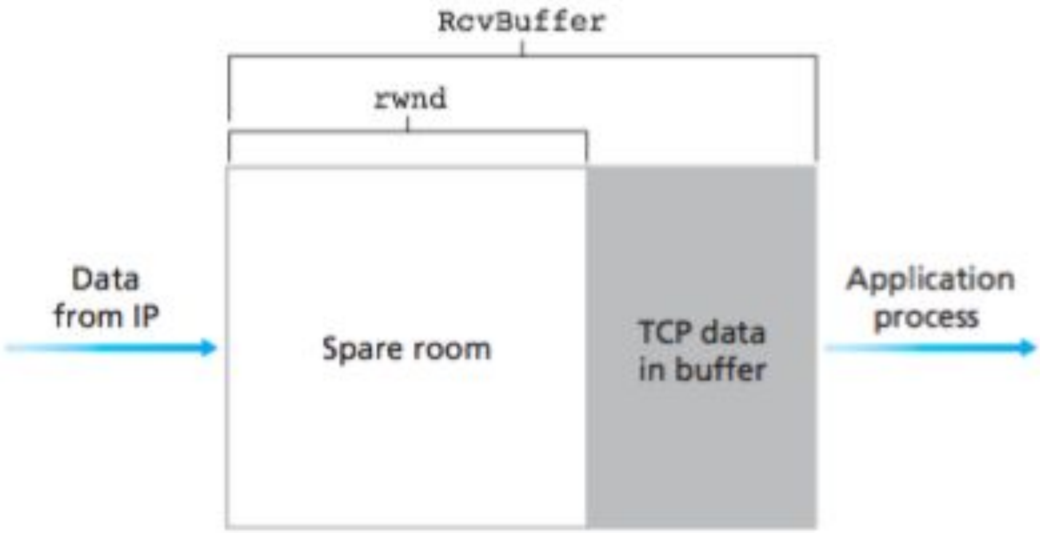
- Unused Buffer Space
  - rwnd




rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

# TCP flow control

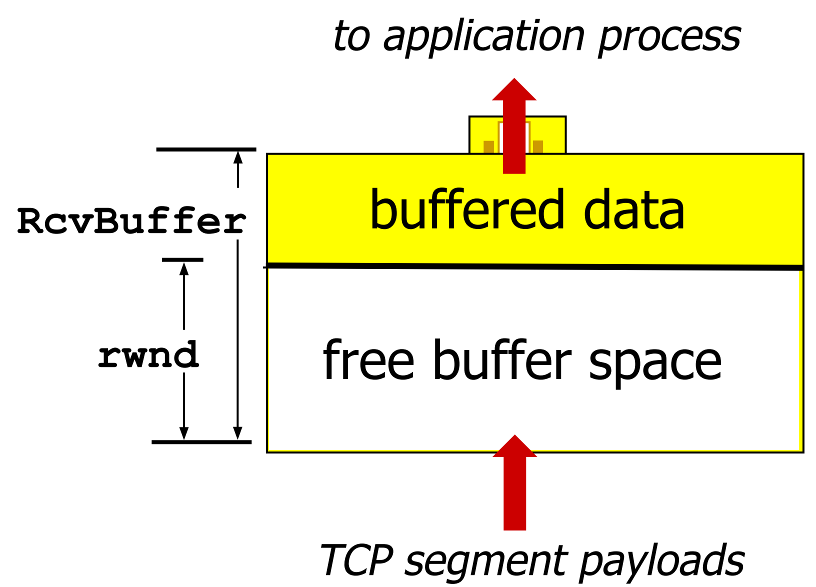
- Receiver
  - Sends rwnd to Sender
- Sender
  - Limits # of unACKed bytes to rwnd




LastByteRcvd – LastByteRead ≤ RcvBuffer

# 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*

# Transport Layer

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

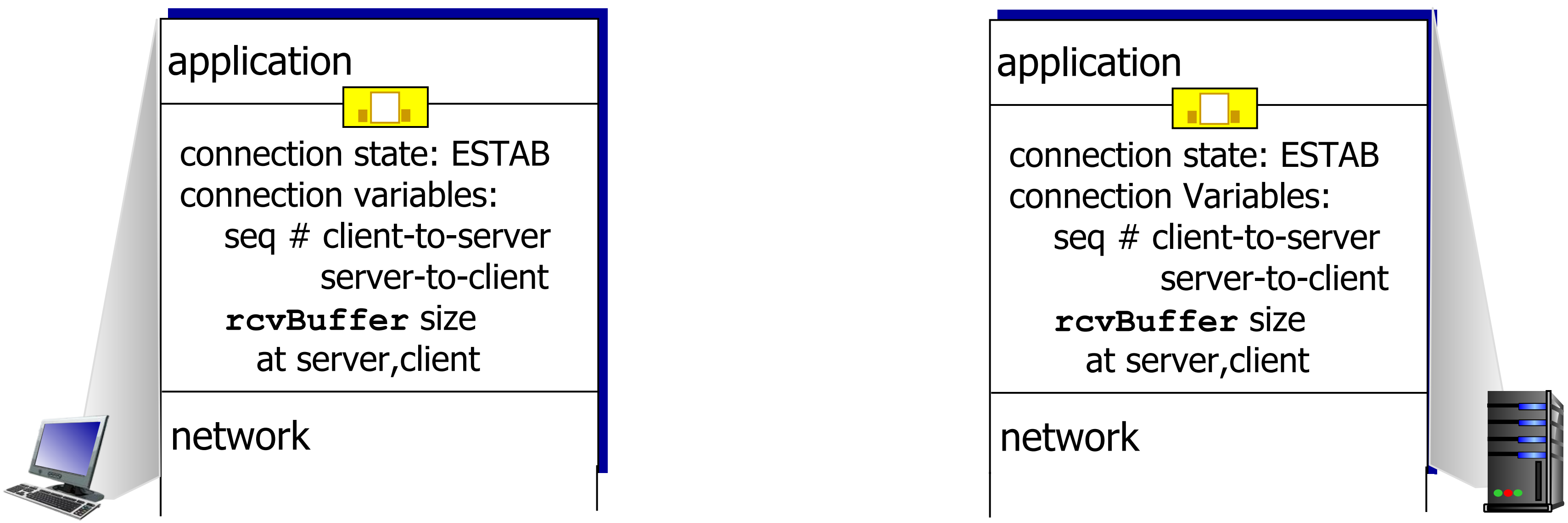
# TCP Connection Management

- Connection-Oriented
- TCP Variables
  - Seq #s
  - Buffers
  - Flow Control (rwnd)

# Connection Management

before exchanging data, sender/receiver "handshake":

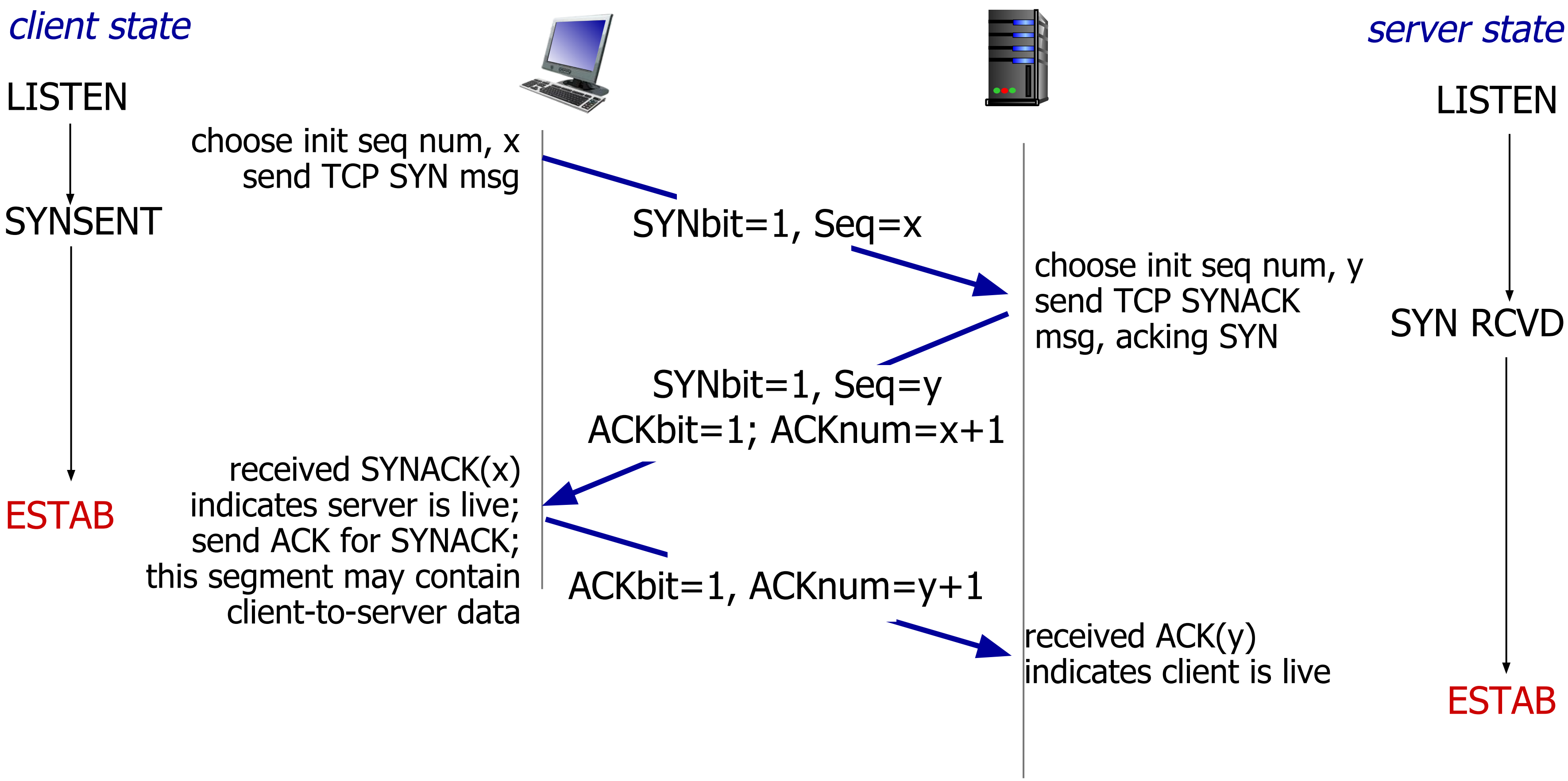
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



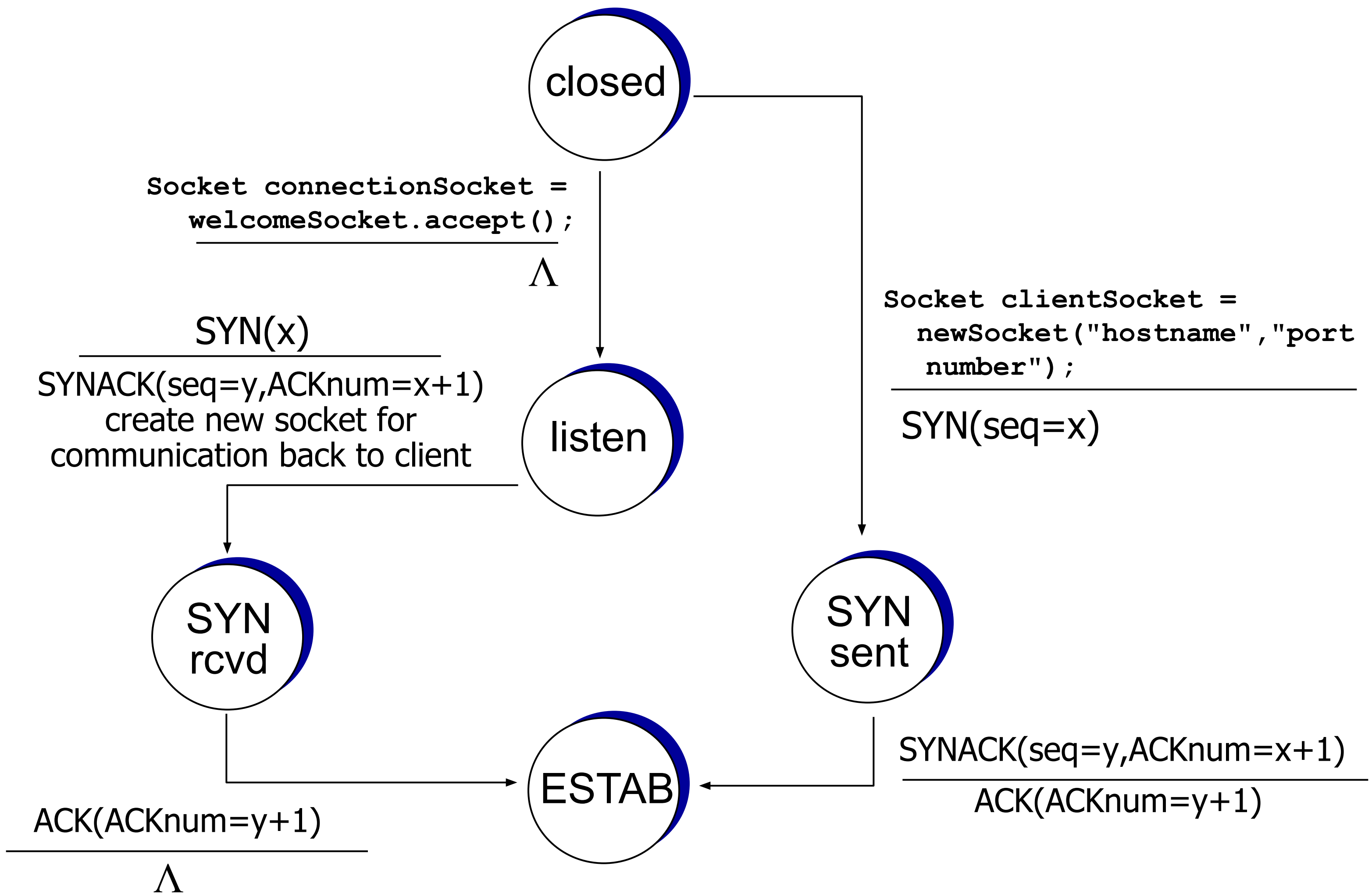

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

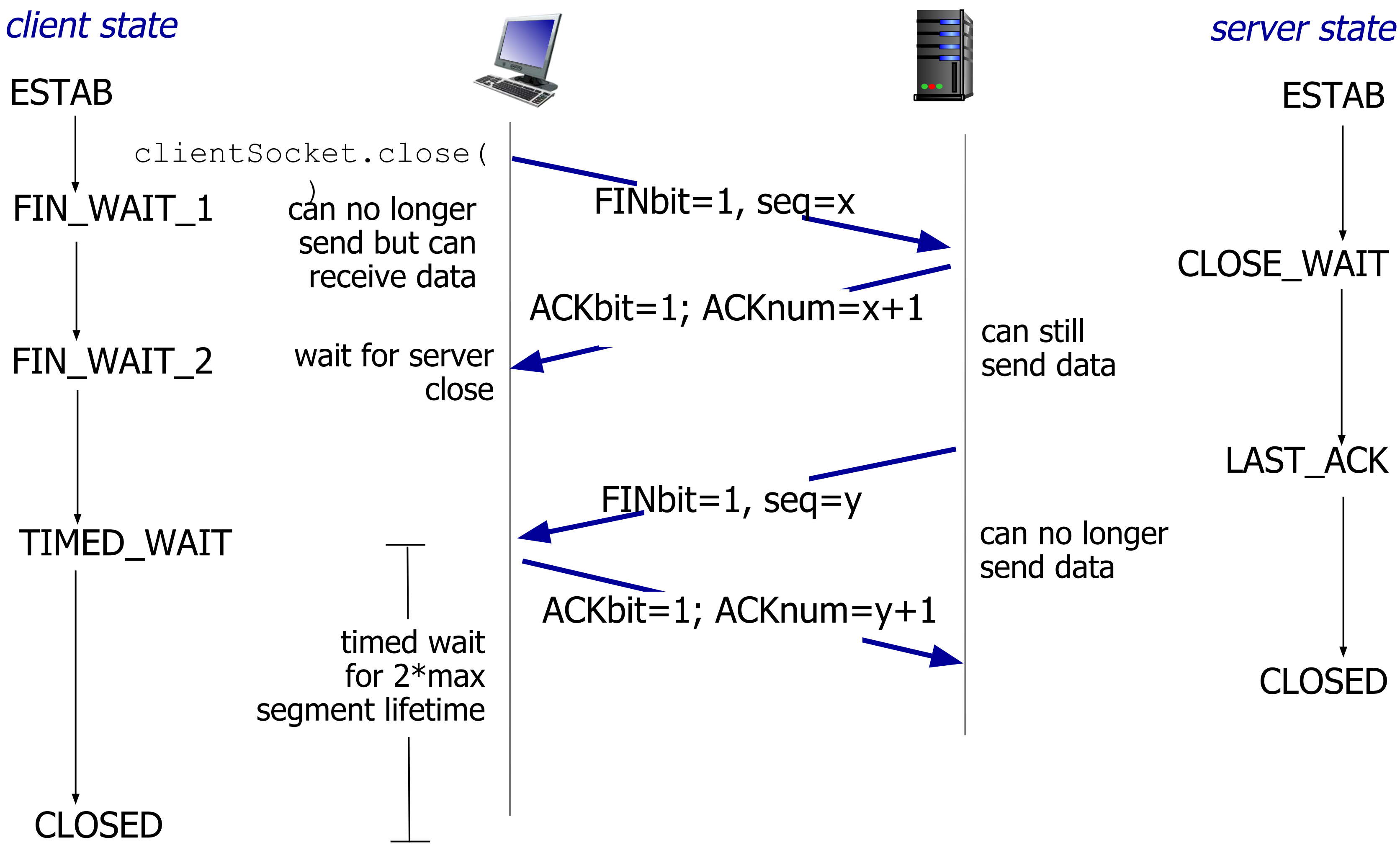
# TCP 3-way handshake



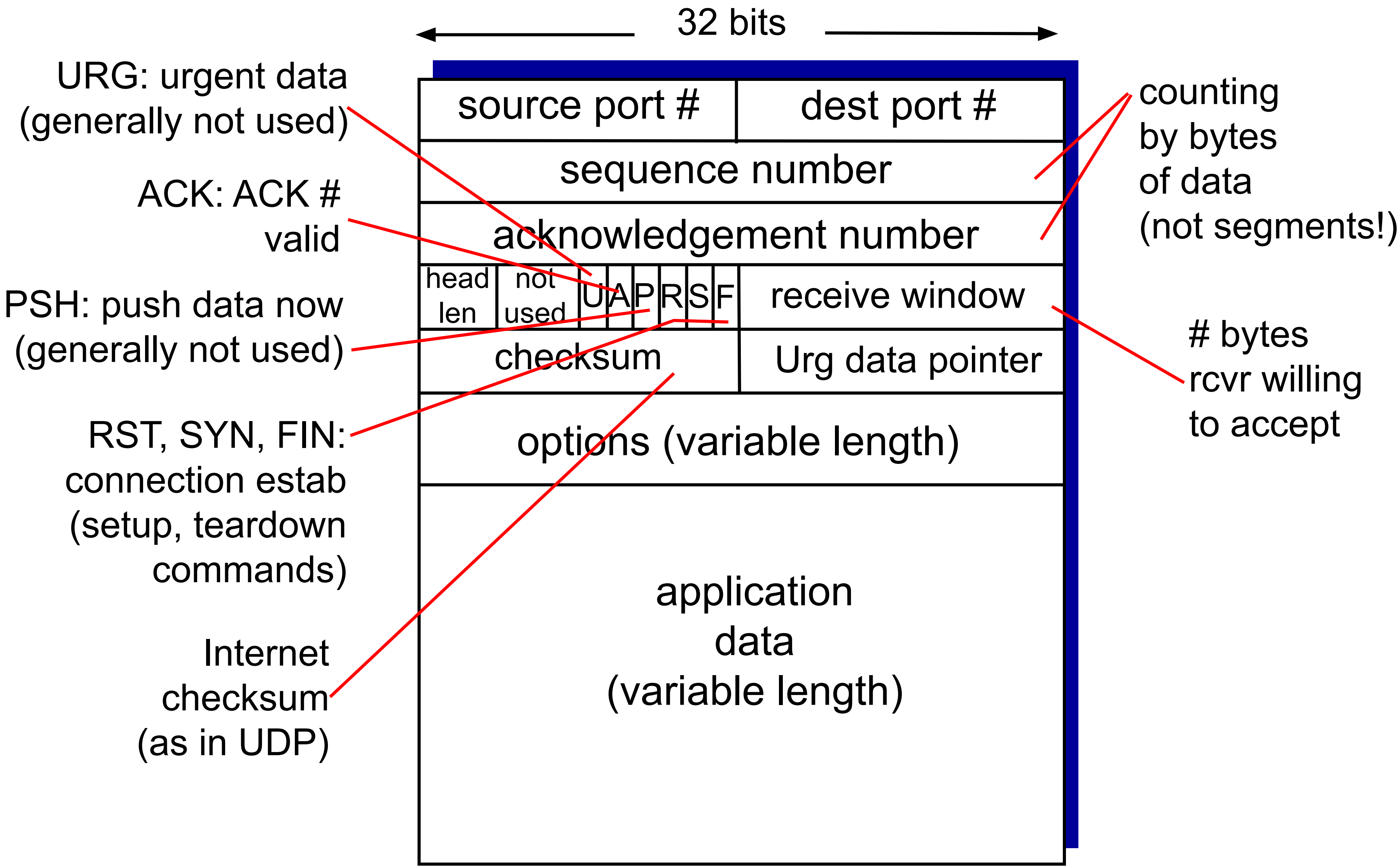
# TCP 3-way handshake: FSM



# TCP: closing a connection



# TCP segment structure



# Transport Layer

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

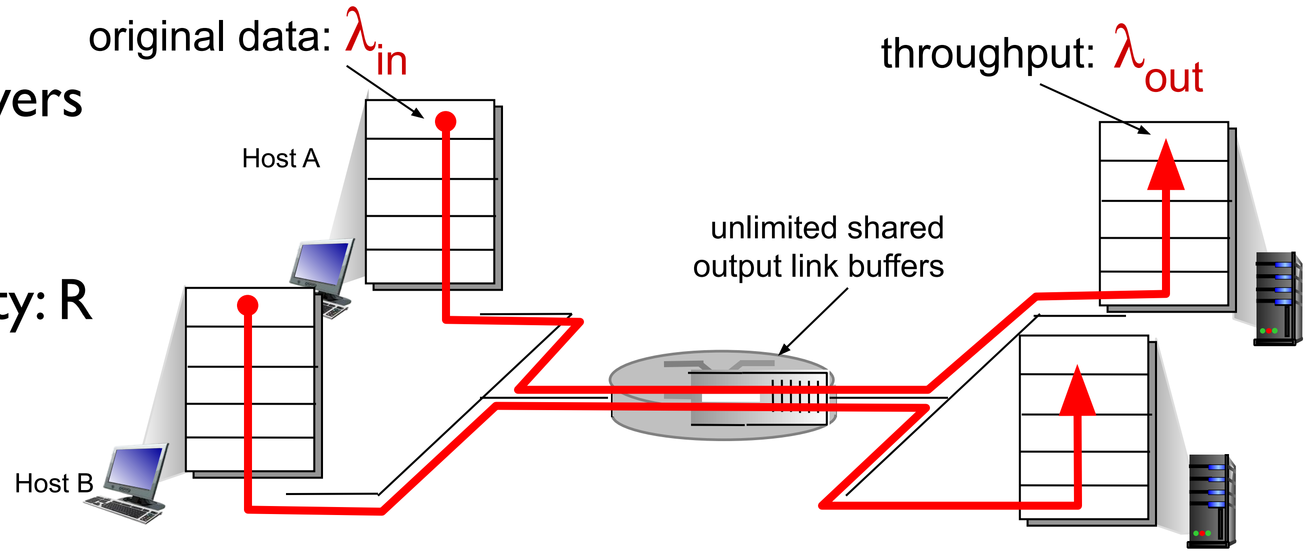
# Principles of congestion control

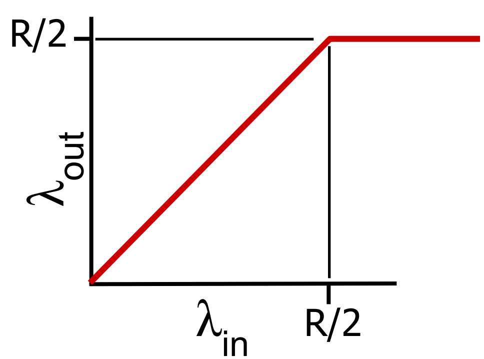
*congestion:*

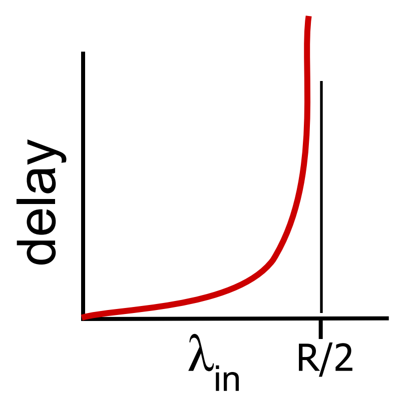
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

# Causes/costs of congestion: scenario 1

- 2 senders, 2 receivers
- 1 router, infinite buffers
- output link capacity: R
- no retransmission



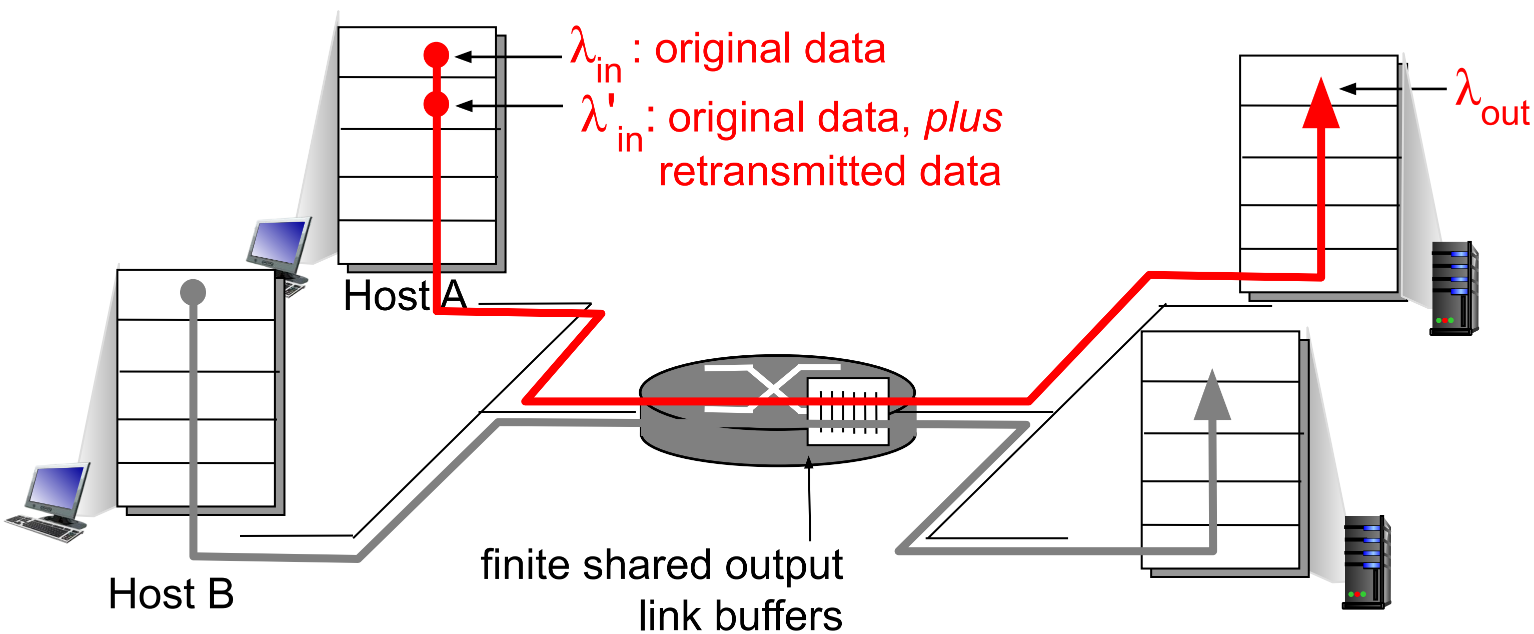




- maximum per-connection throughput: R/2
- large delays as arrival rate, $\lambda_{in}$, approaches capacity

# Causes/costs of congestion: scenario 2

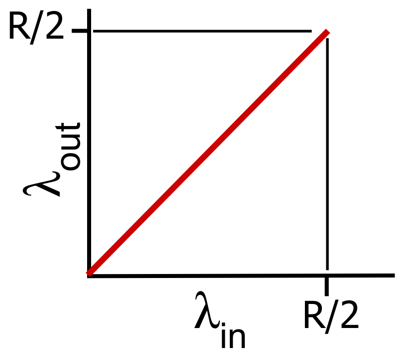
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in}$ = $\lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

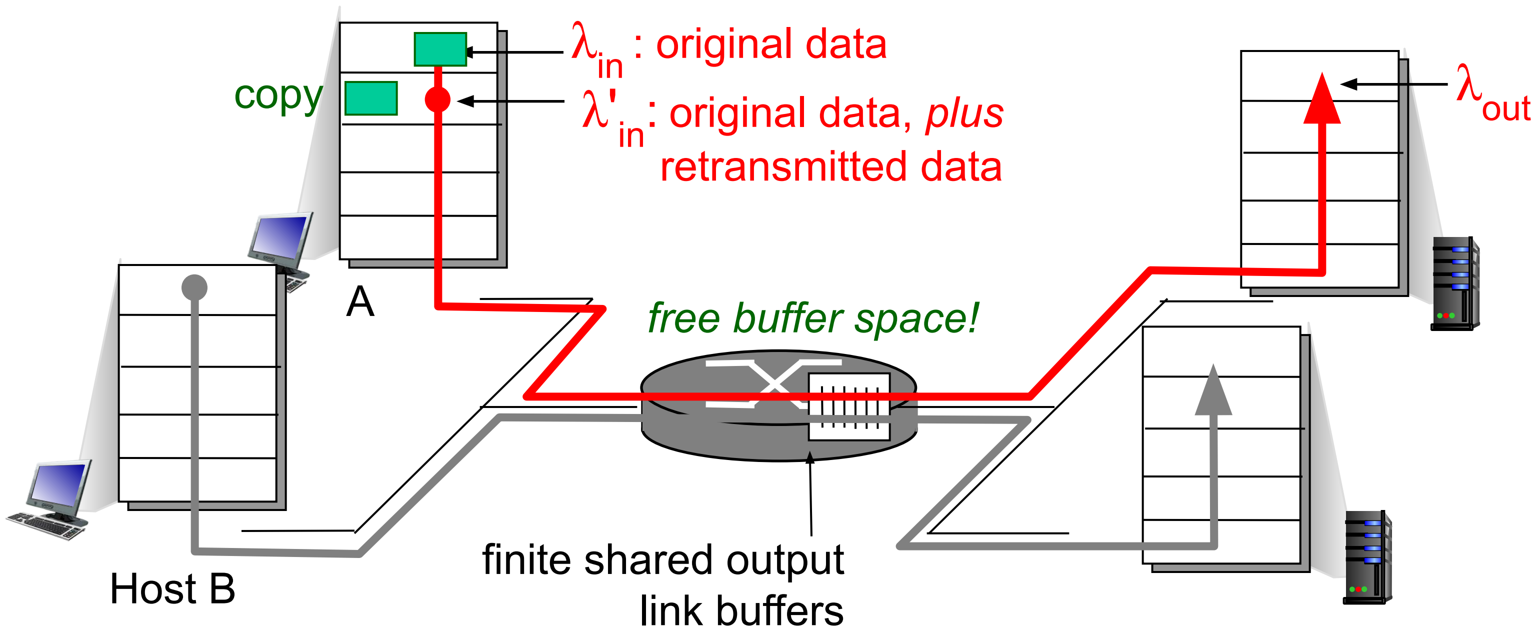


# Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

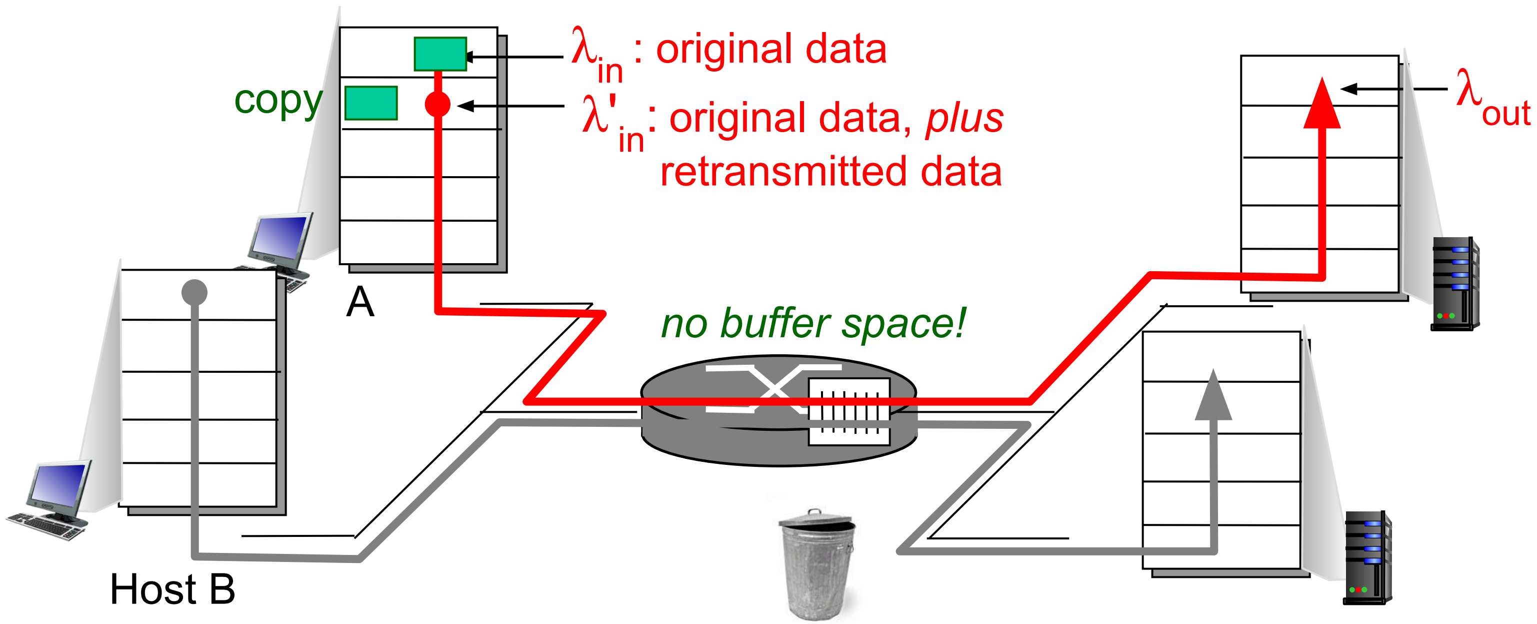


# Causes/costs of congestion: scenario 2

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

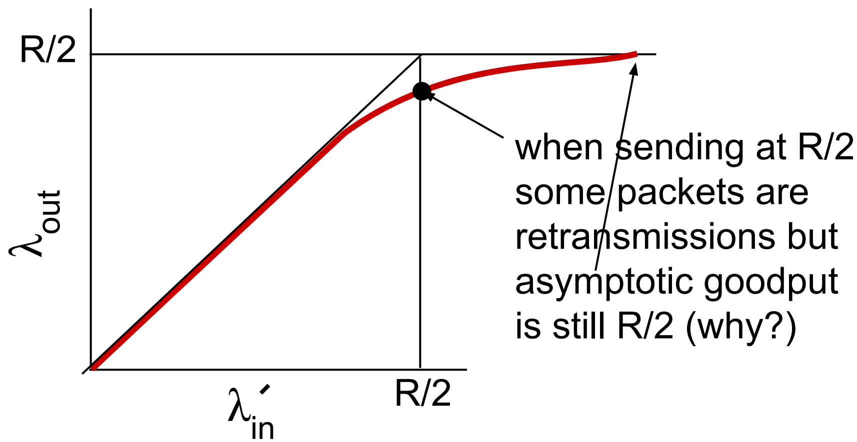
- sender only resends if packet *known* to be lost

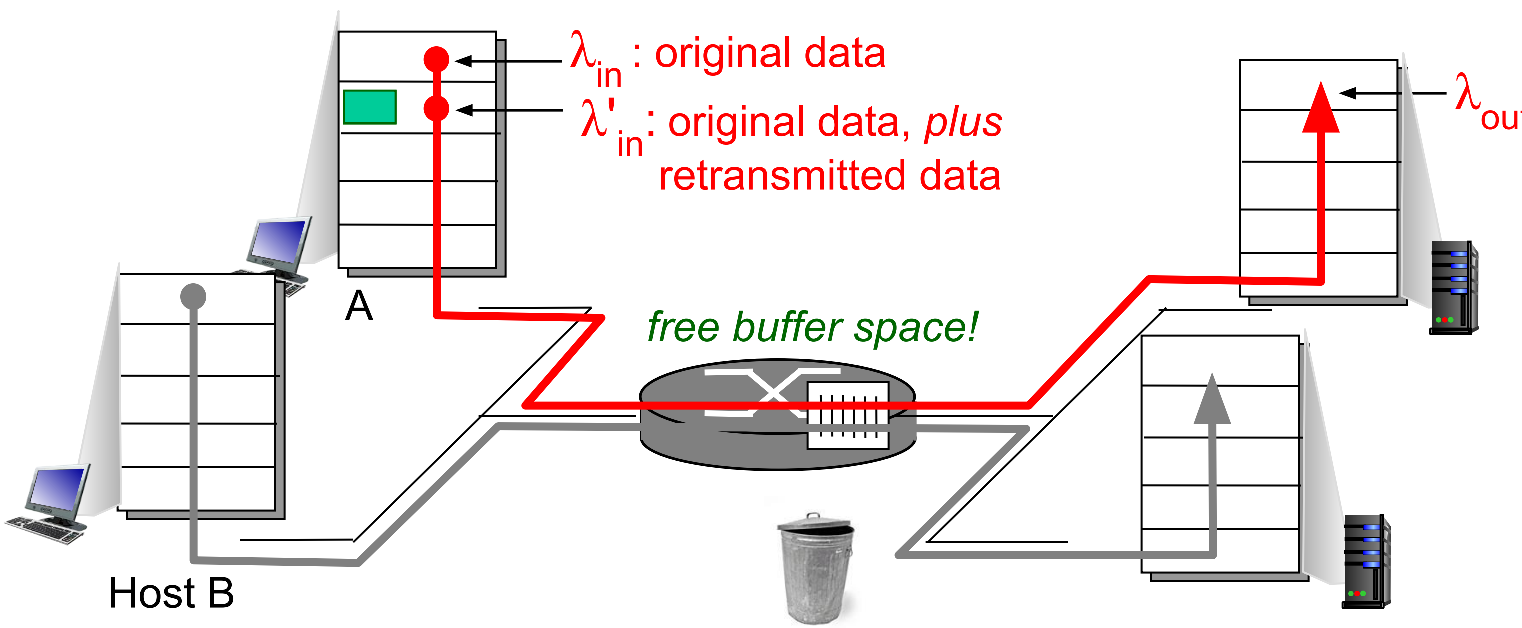


# Causes/costs of congestion: scenario 2

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

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

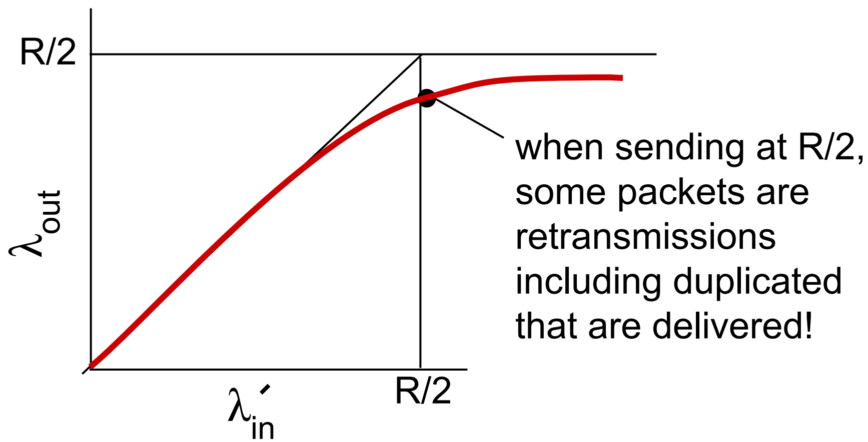


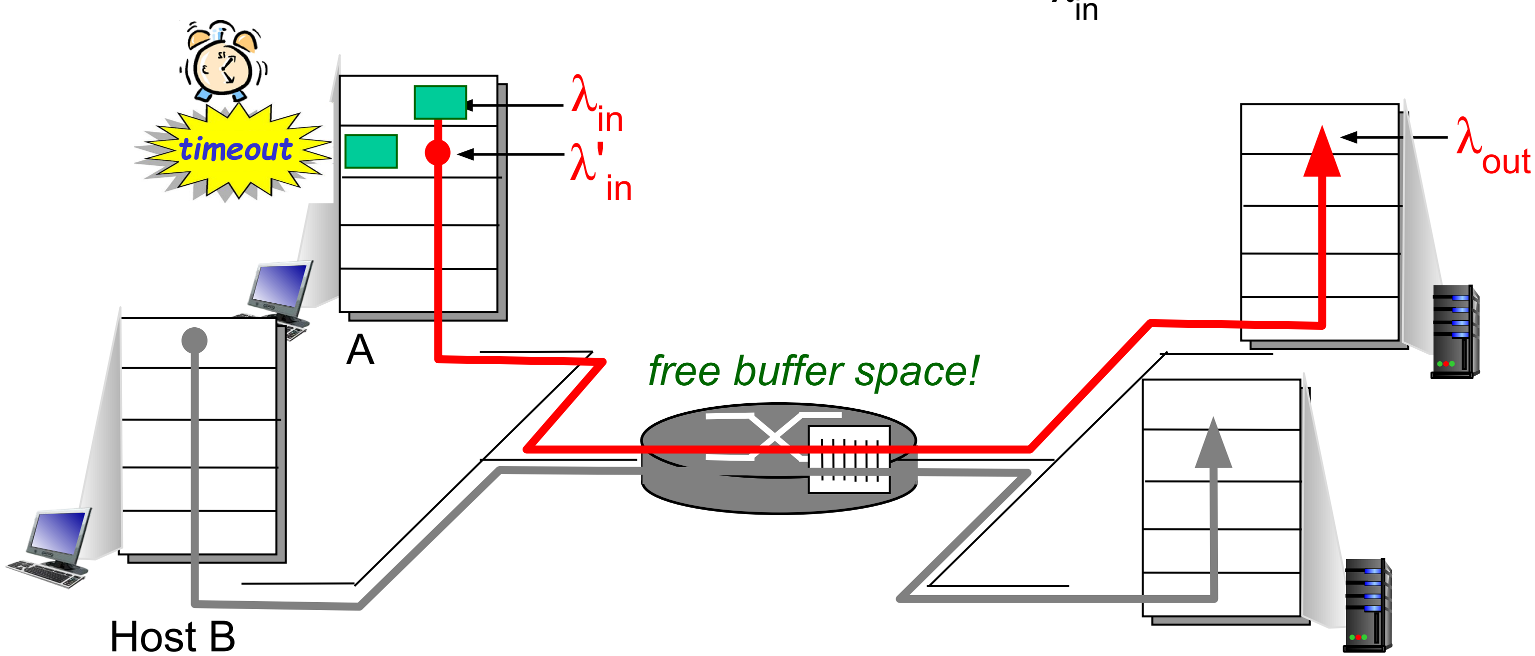



# Causes/costs of congestion: scenario 2

## *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

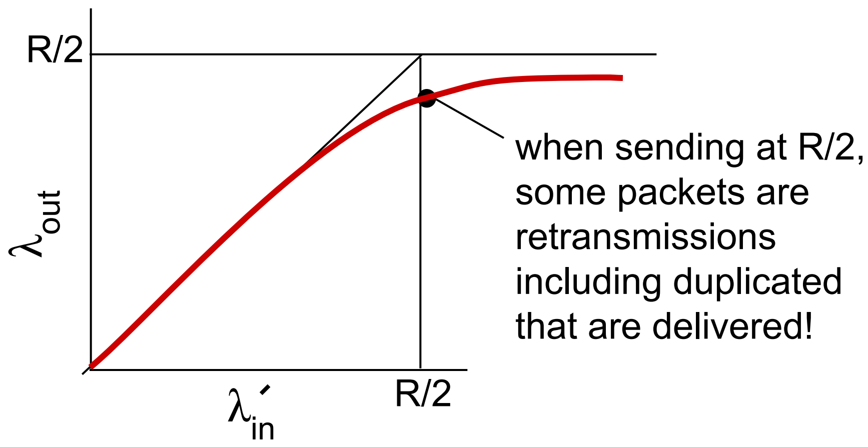


# Causes/costs of congestion: scenario 2

## *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:

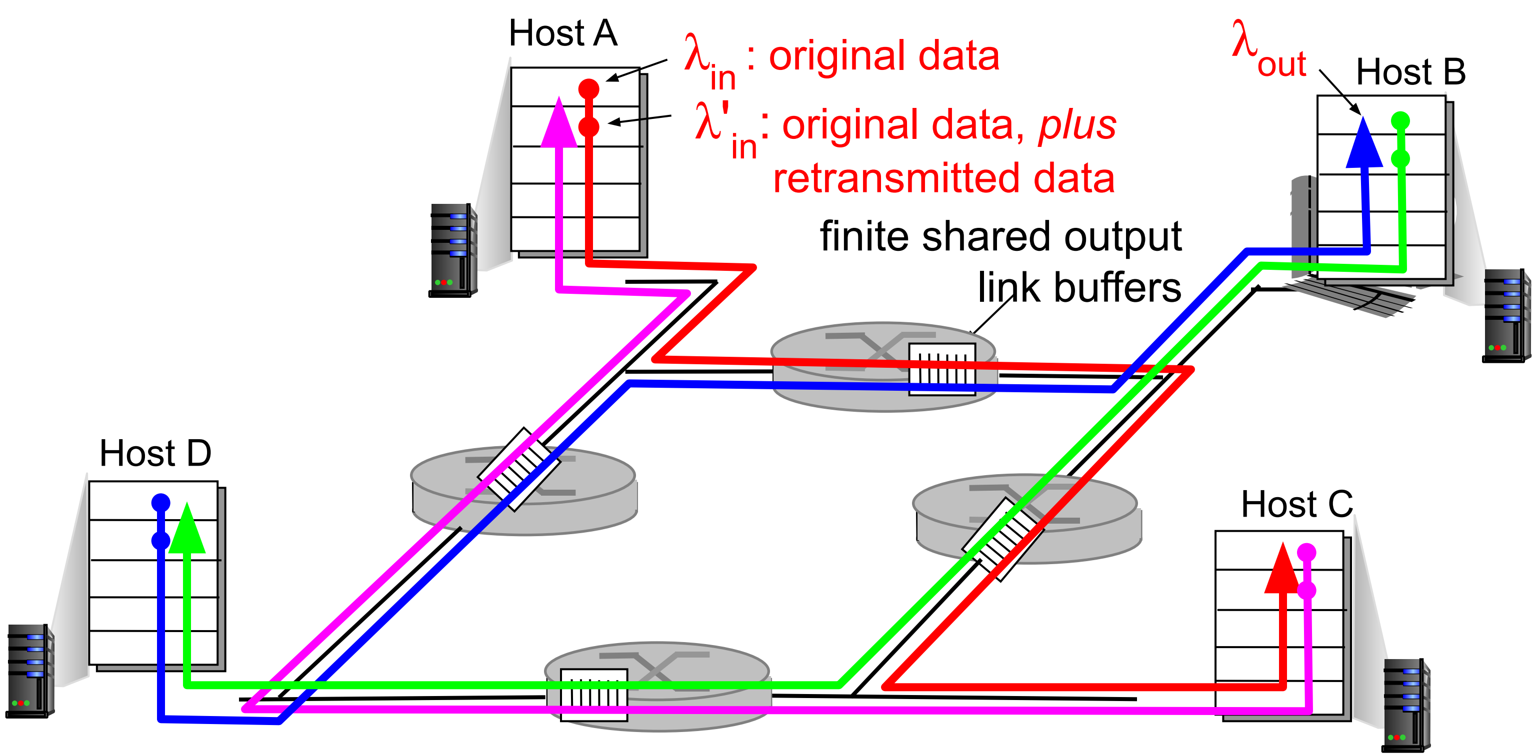
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

# Causes/costs of congestion: scenario 3

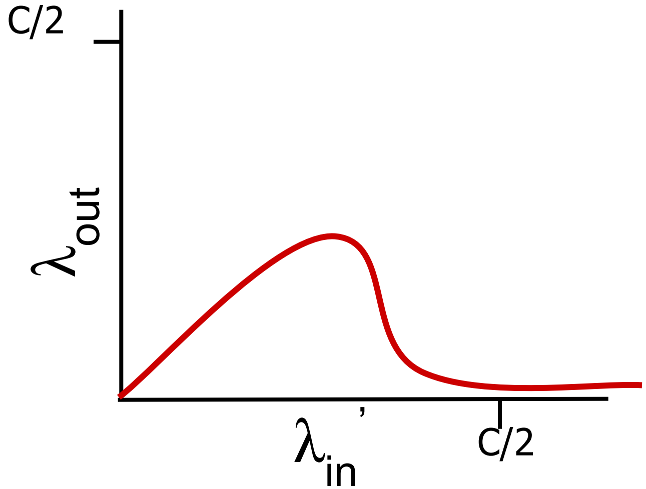
- four senders
- multihop paths
- timeout/retransmit

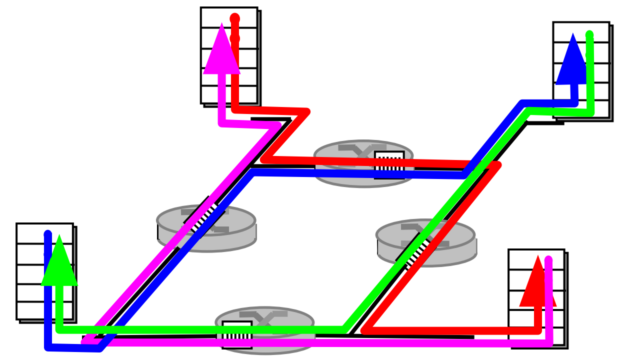
Q: what happens as $\lambda_{in}$ and $\lambda_{in}'$ increase ?

A: as red $\lambda_{in}'$ increases, all arriving blue pkts at upper queue are dropped, blue throughput $\rightarrow$ 0



# Causes/costs of congestion: scenario 3



another "cost" of congestion:

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

# Approaches towards congestion control

two broad approaches towards congestion control:

**end-end congestion control:**

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

**network-assisted congestion control:**

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

# Case study: ATM ABR congestion control

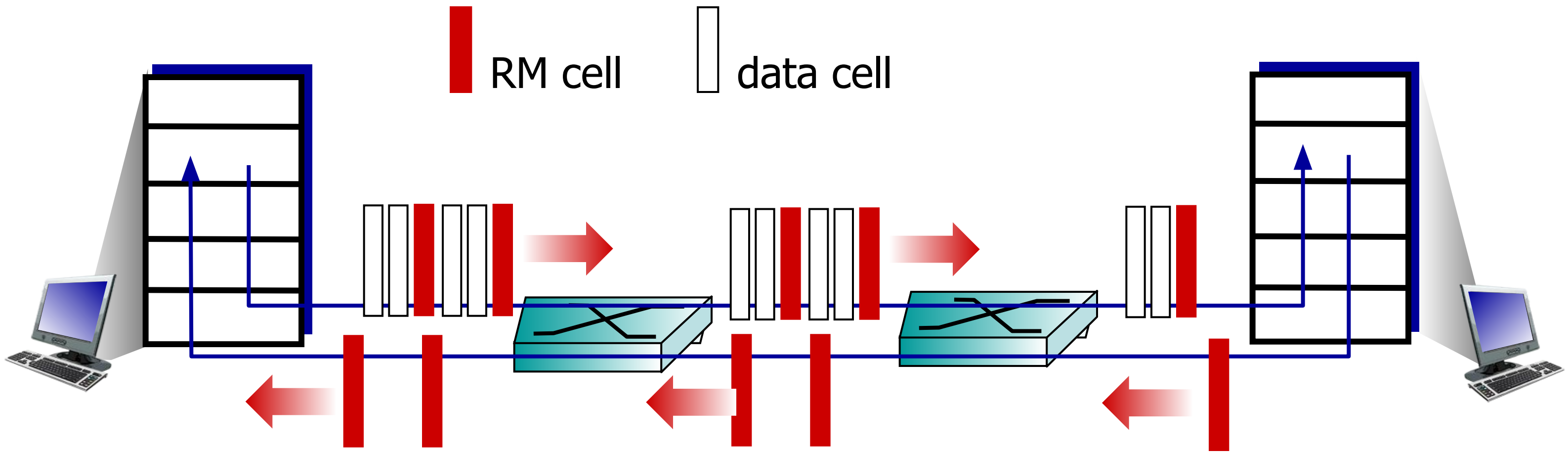
## ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

## RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - *NI bit:* no increase in rate (mild congestion)
  - *CI bit:* congestion indication
- RM cells returned to sender by receiver, with bits intact

# Case study: ATM ABR congestion control




- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - senders' send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

# Transport Layer

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-to-End*

- Limit send rate when network is congested

- Questions:
  - How to perceive congestion?
  - How to limit send rate?
  - How to change send rate?

# How to perceive congestion?

- *Implicit End-to-End Feedback*

- ACK Received:
  - ?

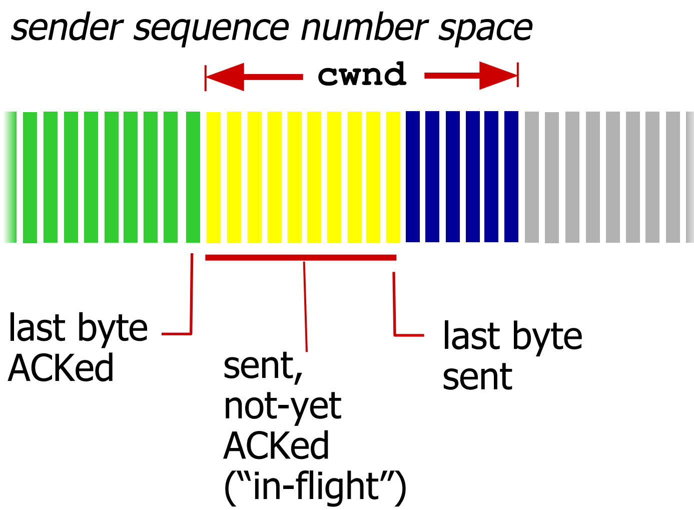
- ACK not Received:
  - ?

# How to limit send rate?

- *Limit # of unACKed bytes in pipeline*
  - *cwnd (congestion window)*
  - *Sender limited by min (cwnd, rwnd)*

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \le \texttt{min\{cwnd, rwnd\}}$$

# TCP Congestion Control: details




- sender limits transmission:

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

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

*TCP sending rate:*

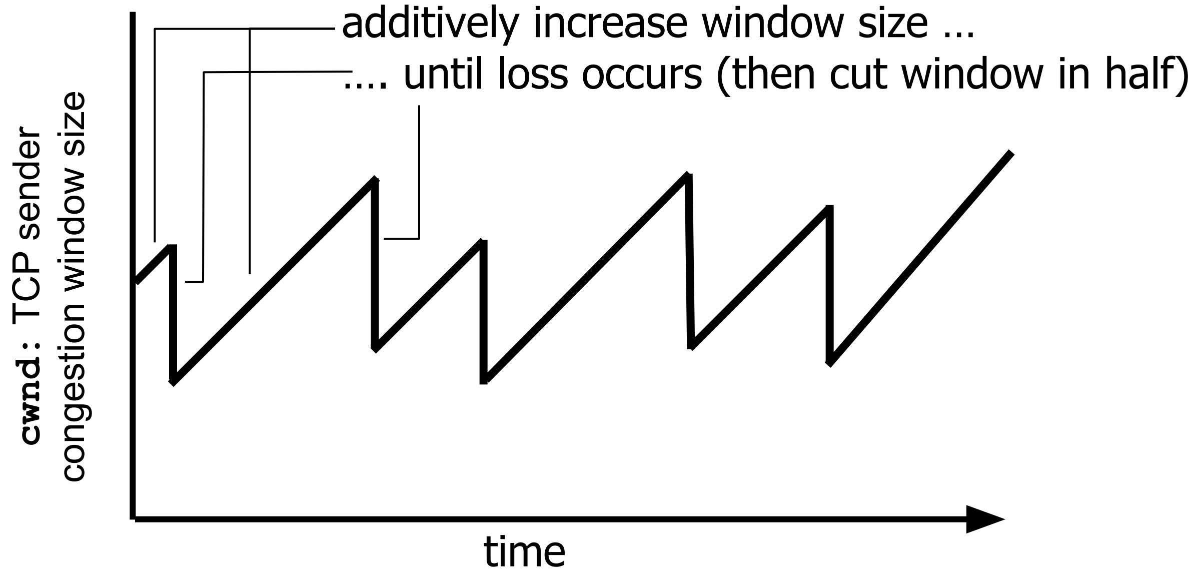
- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

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

# TCP 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



# Success Event

- *If ACK received – increase the cwnd*
  - *Slowstart*
    - *Increase Exponentially*
    - *Connection Start or After Timeout*
  - *Congestion Avoidance*
    - *Increase Linearly*
    - *Normal Operation*

# Loss Event

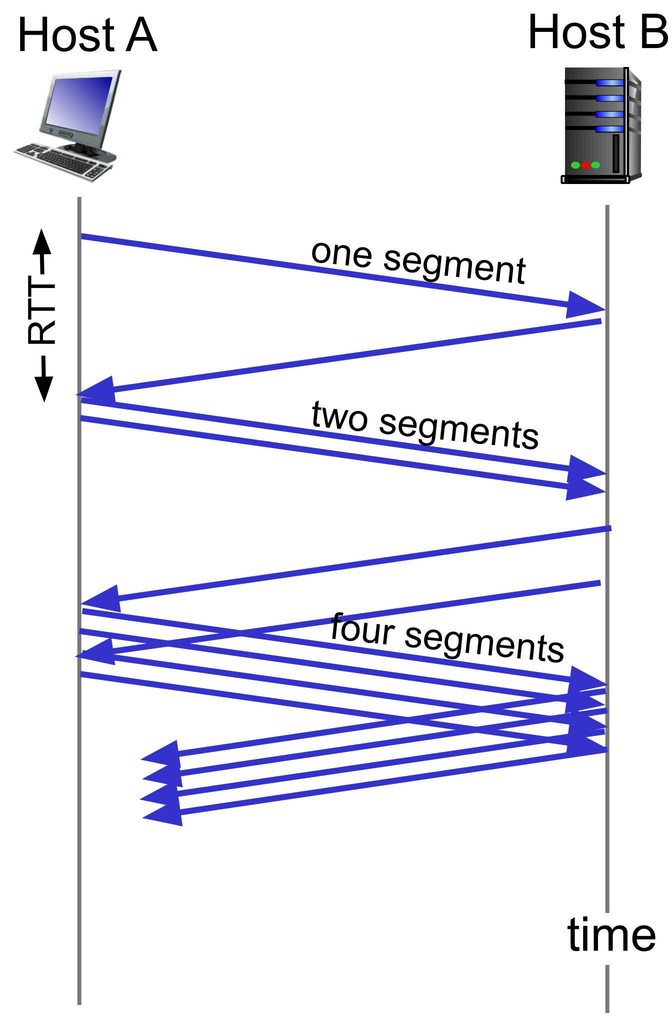
- *If segment lost – decrease the cwnd*
  - *Timeout*
    - *Cut cwnd to 1*
  - *3 Duplicate ACKs*
    - *Cut cwnd in half*

# TCP: detecting, reacting to loss

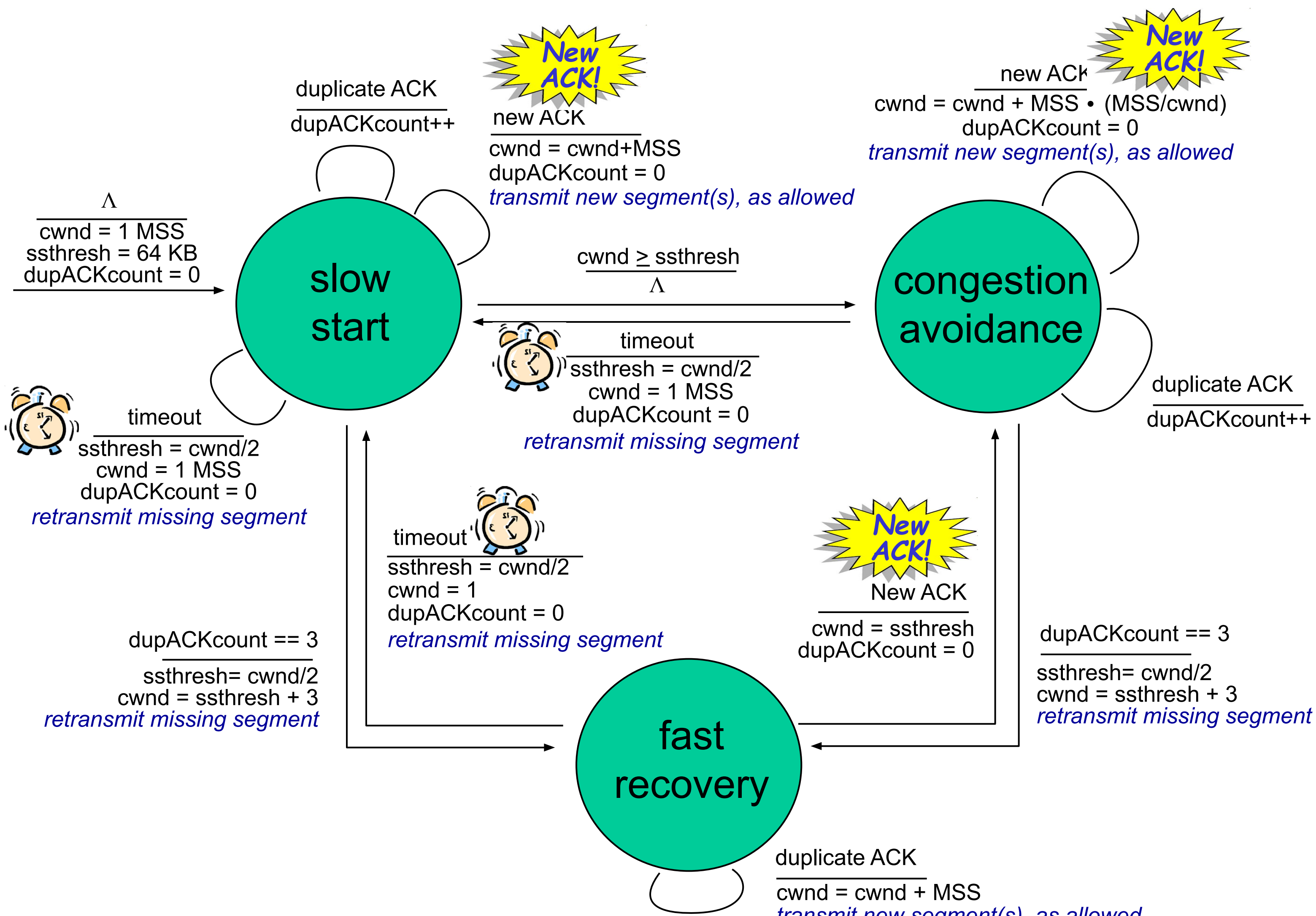
- loss indicated by timeout:
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - **`cwnd`** is cut in half window then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

# TCP 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



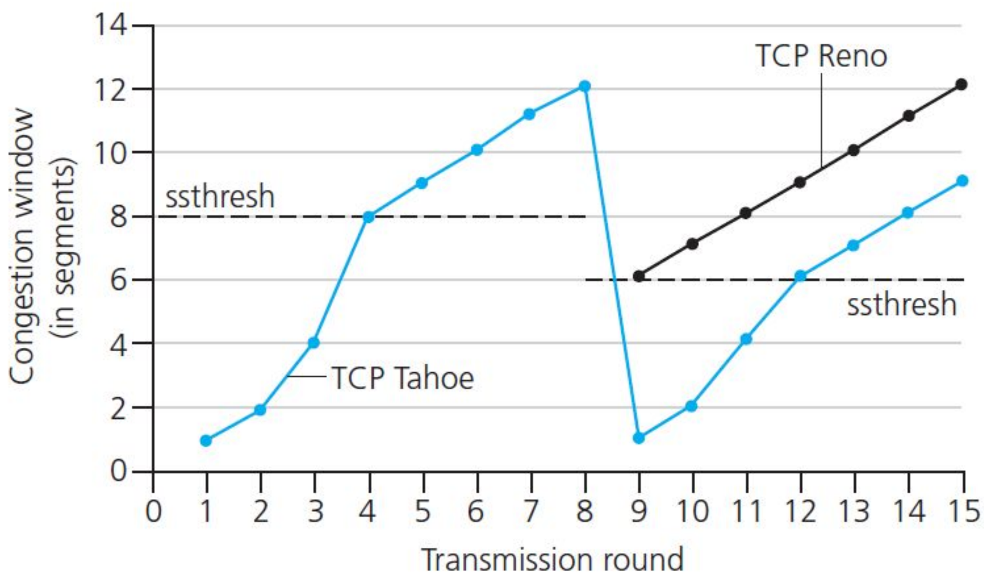
# Summary: TCP Congestion Control



# 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.



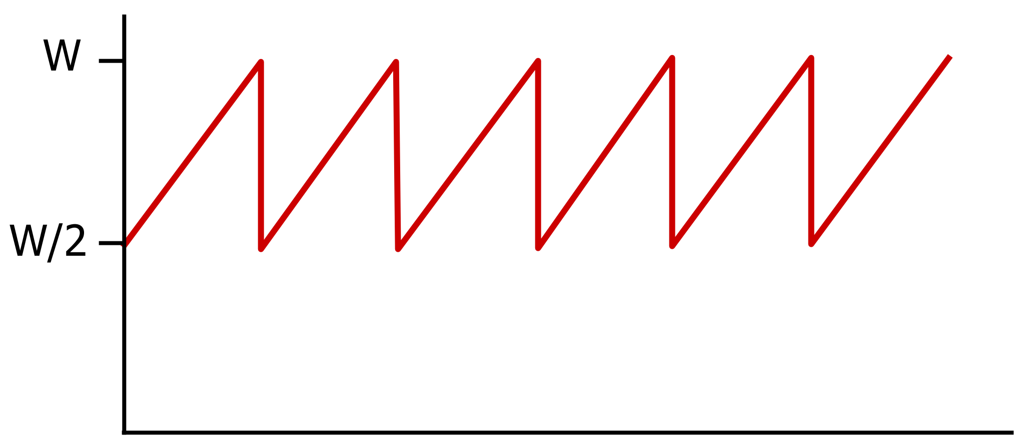

## Implementation:

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

# 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]:

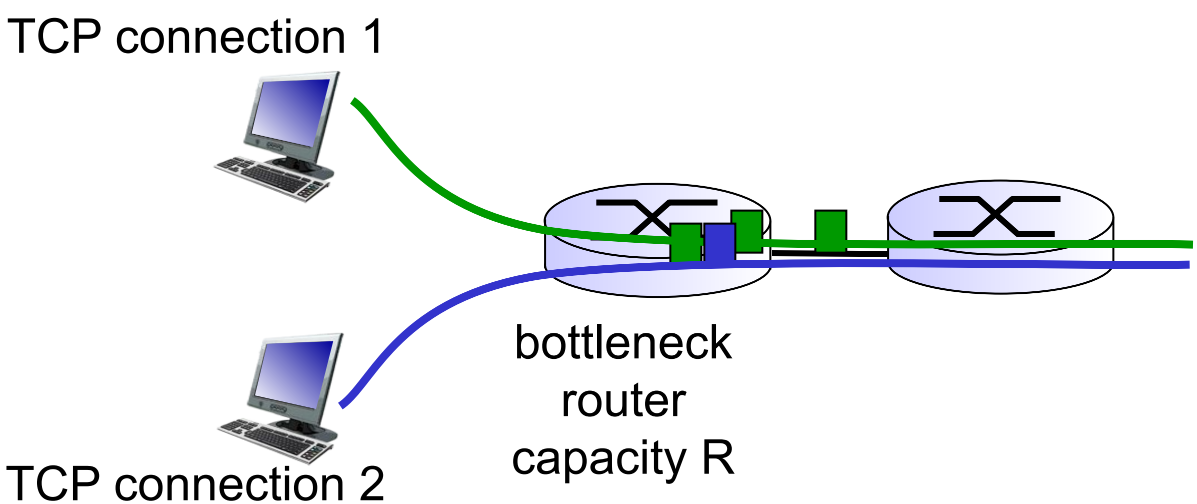
$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

➜ 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

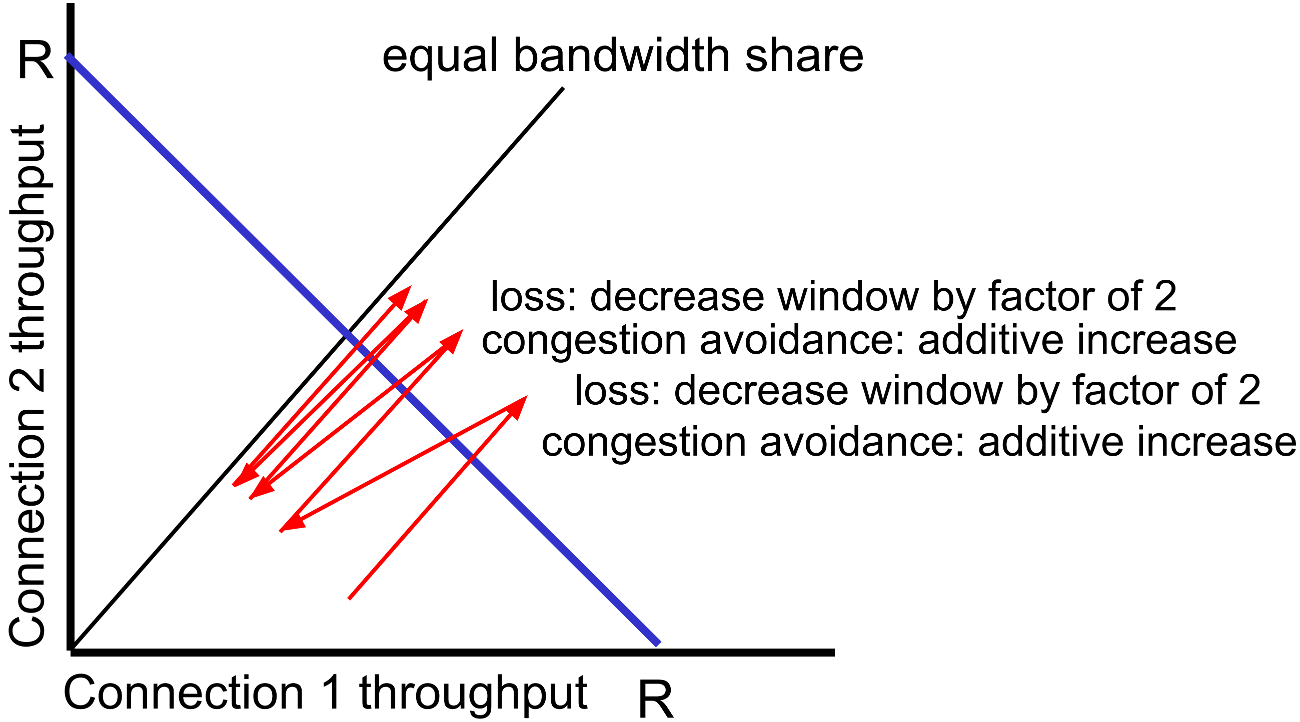
*fairness goal:* if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



# 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

# Transport Layer

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

<u>next:</u>

- leaving the network "edge" (application, transport layers)
- into the network "core"