# Module 2: 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

# Module 2 outline

2.1 transport-layer services

2.2 multiplexing and demultiplexing

2.3 connectionless transport: UDP

2.4 principles of reliable data transfer

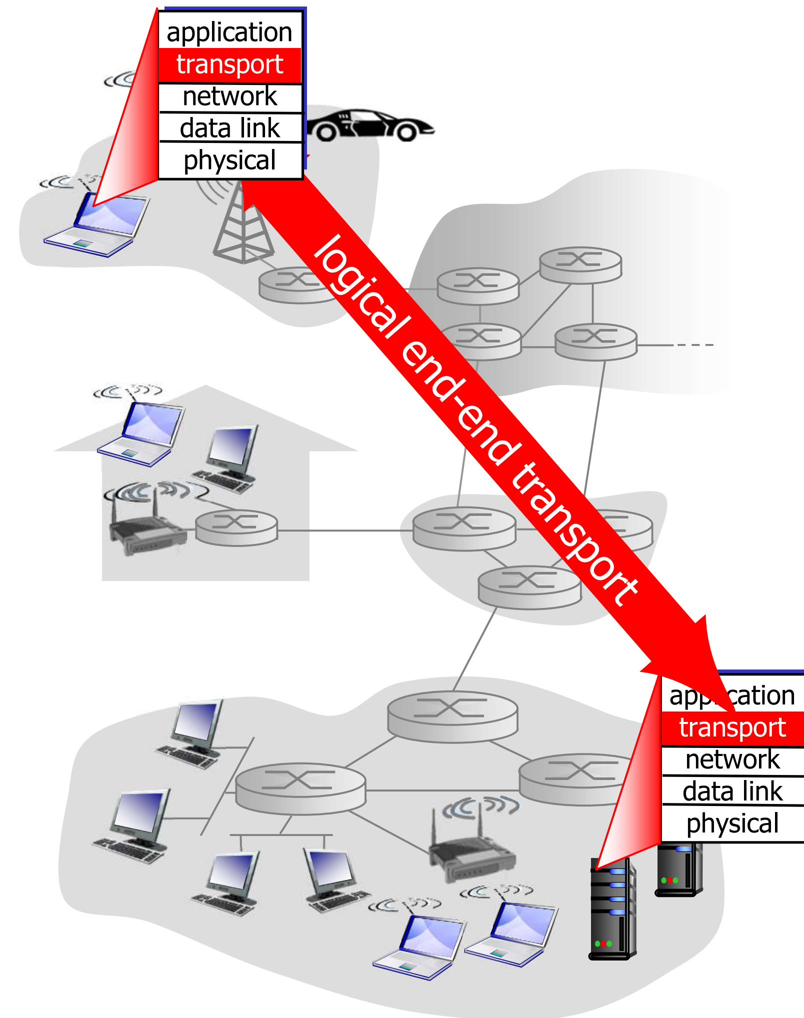
2.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

2.6 principles of congestion control

2.7 TCP congestion control

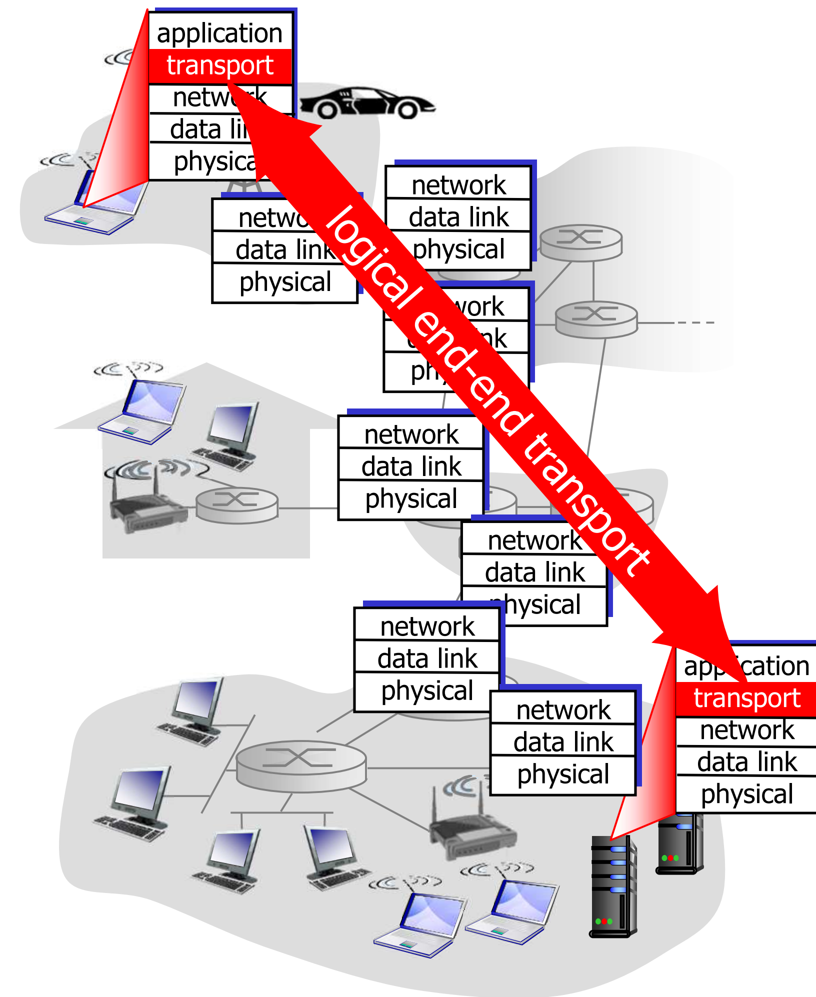
# Transport services and protocols

- 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



# Internet transport-layer protocols

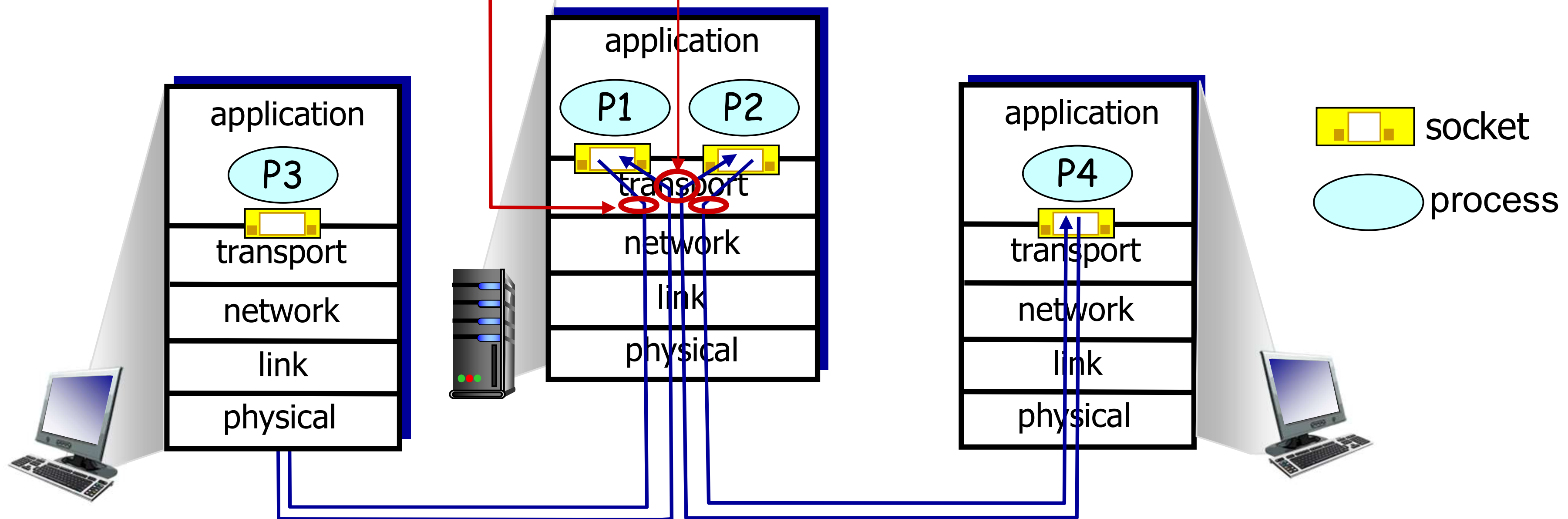
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



# Multiplexing/demultiplexing

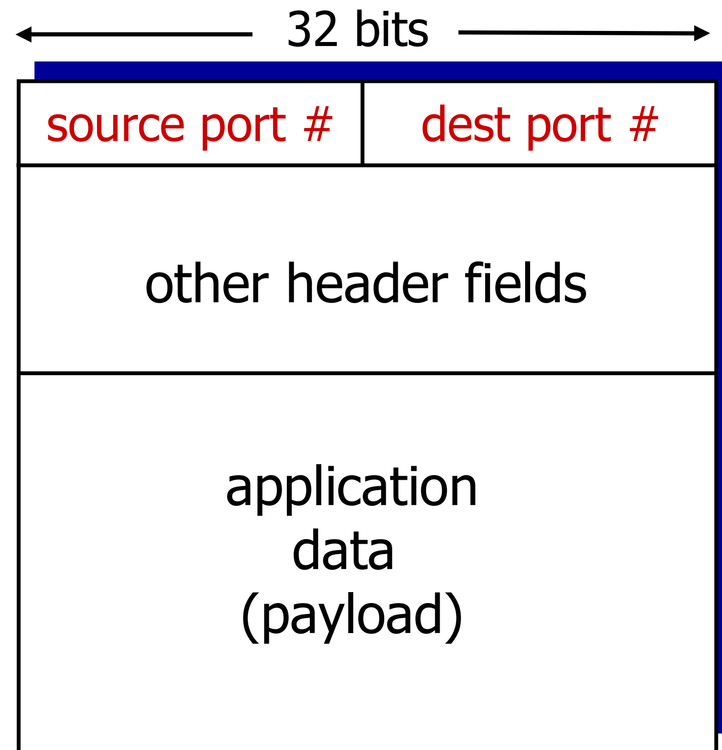
*multiplexing at sender:*
handle data from multiple sockets, add transport header (later used for demultiplexing)

*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

```
clientSocket =
    socket(socket.AF_INET,
    socket.SOCK_DGRAM)
```

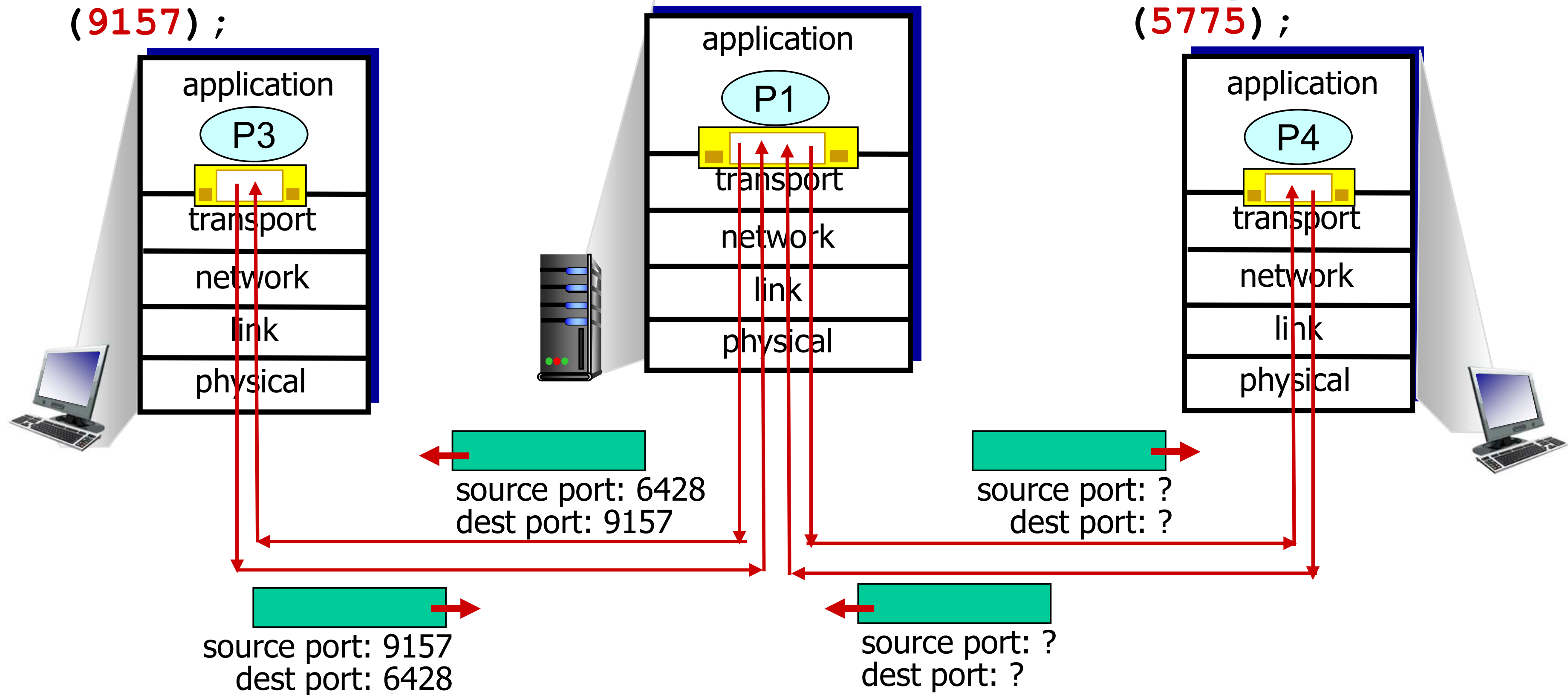
- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

---

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

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

# Connectionless demux: example

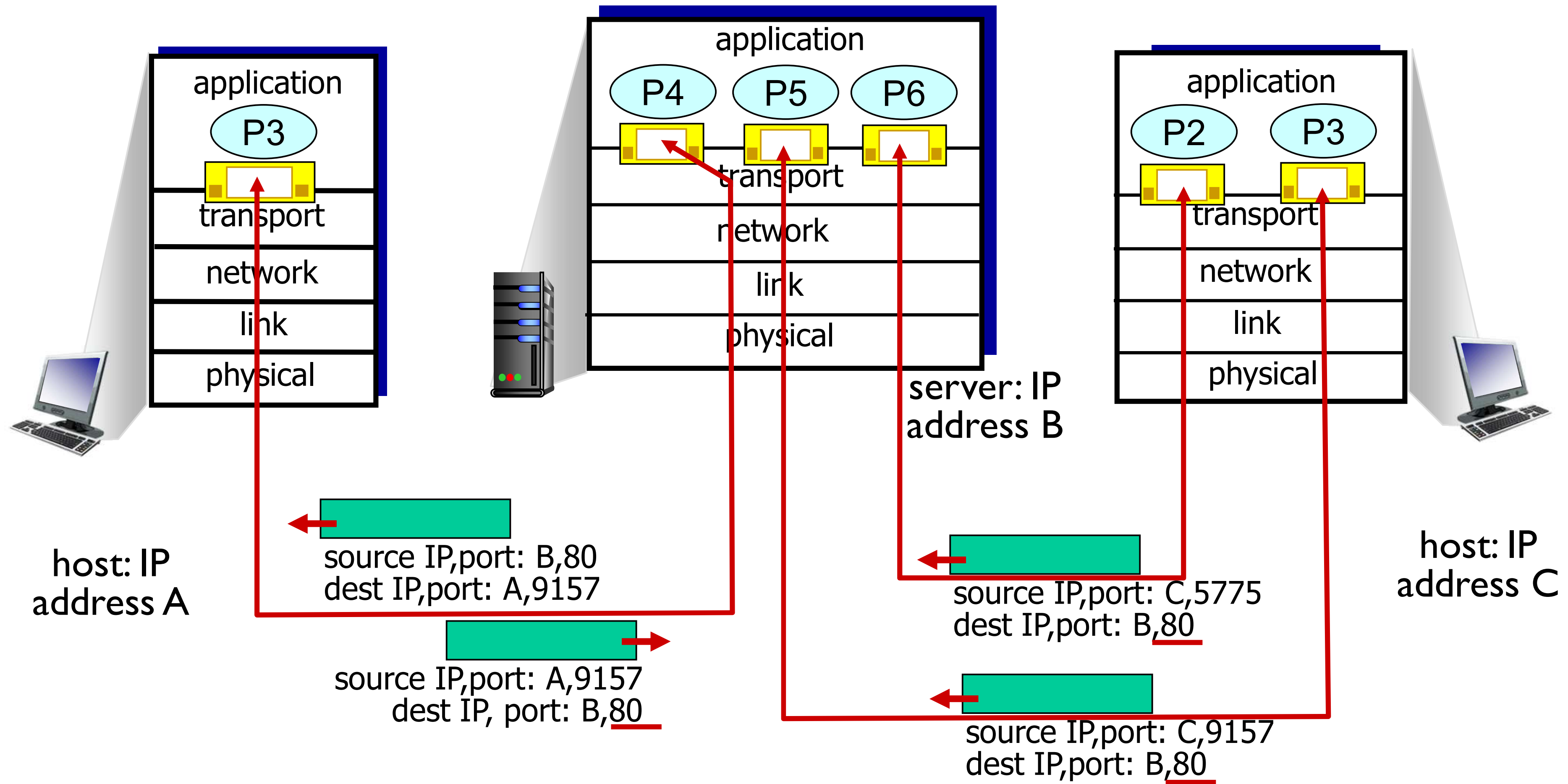


# Connection-oriented demux

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

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

# Connection-oriented demux: example

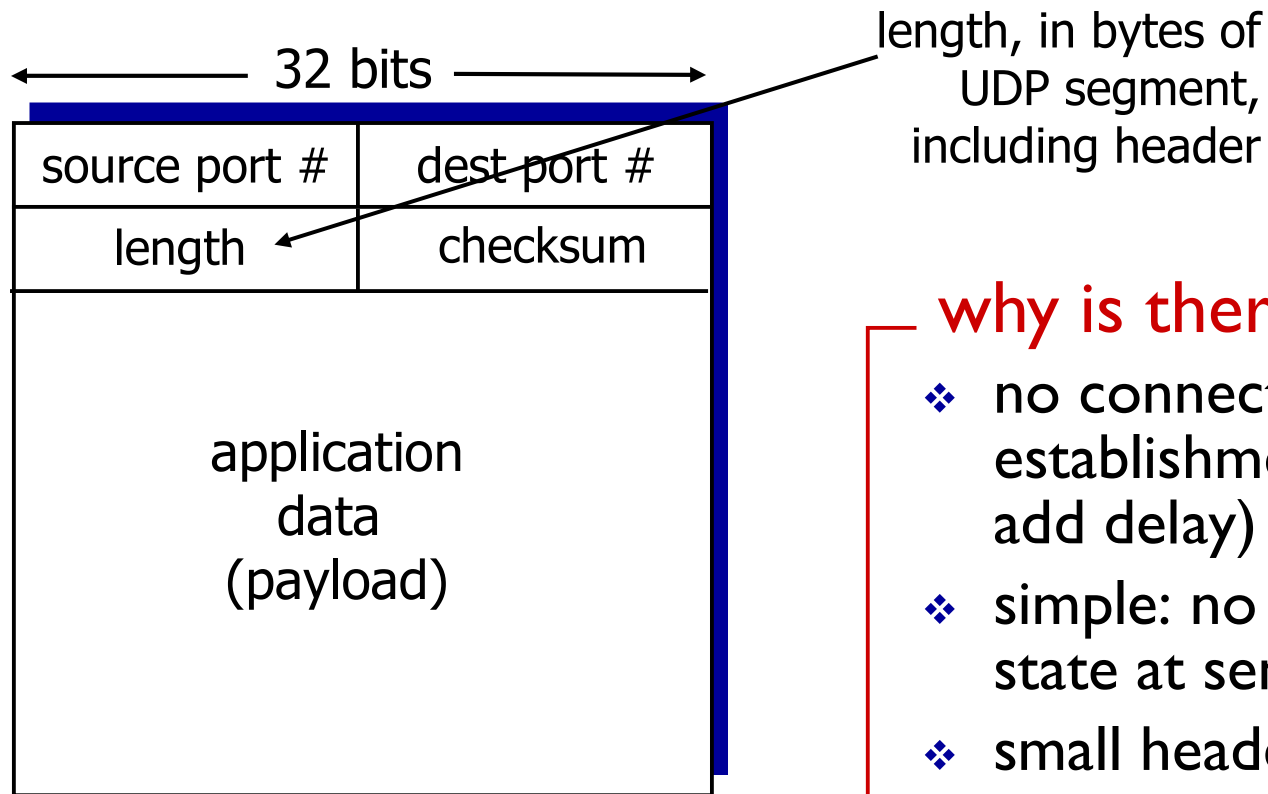


# UDP: User Datagram Protocol [RFC 768]

- "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

| Application | Application-Layer Protocol | Underlying Transport Protocol |
| --- | --- | --- |
| Electronic mail | SMTP | TCP |
| Remote terminal access | Telnet | TCP |
| Web | HTTP | TCP |
| File transfer | FTP | TCP |
| Remote file server | NFS | Typically UDP |
| Streaming multimedia | typically proprietary | UDP or TCP |
| Internet telephony | typically proprietary | UDP or TCP |
| Network management | SNMP | Typically UDP |
| Routing protocol | RIP | Typically UDP |
| Name translation | DNS | Typically UDP |

# 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: all 3 16-bit words are added, including the checksum.
- If no errors are introduced into the packet,
- sum at the receiver will be 1111111111111111.
- If one of the bits is a 0, errors have been introduced into the packet.

# 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 |

# Principles of reliable data transfer

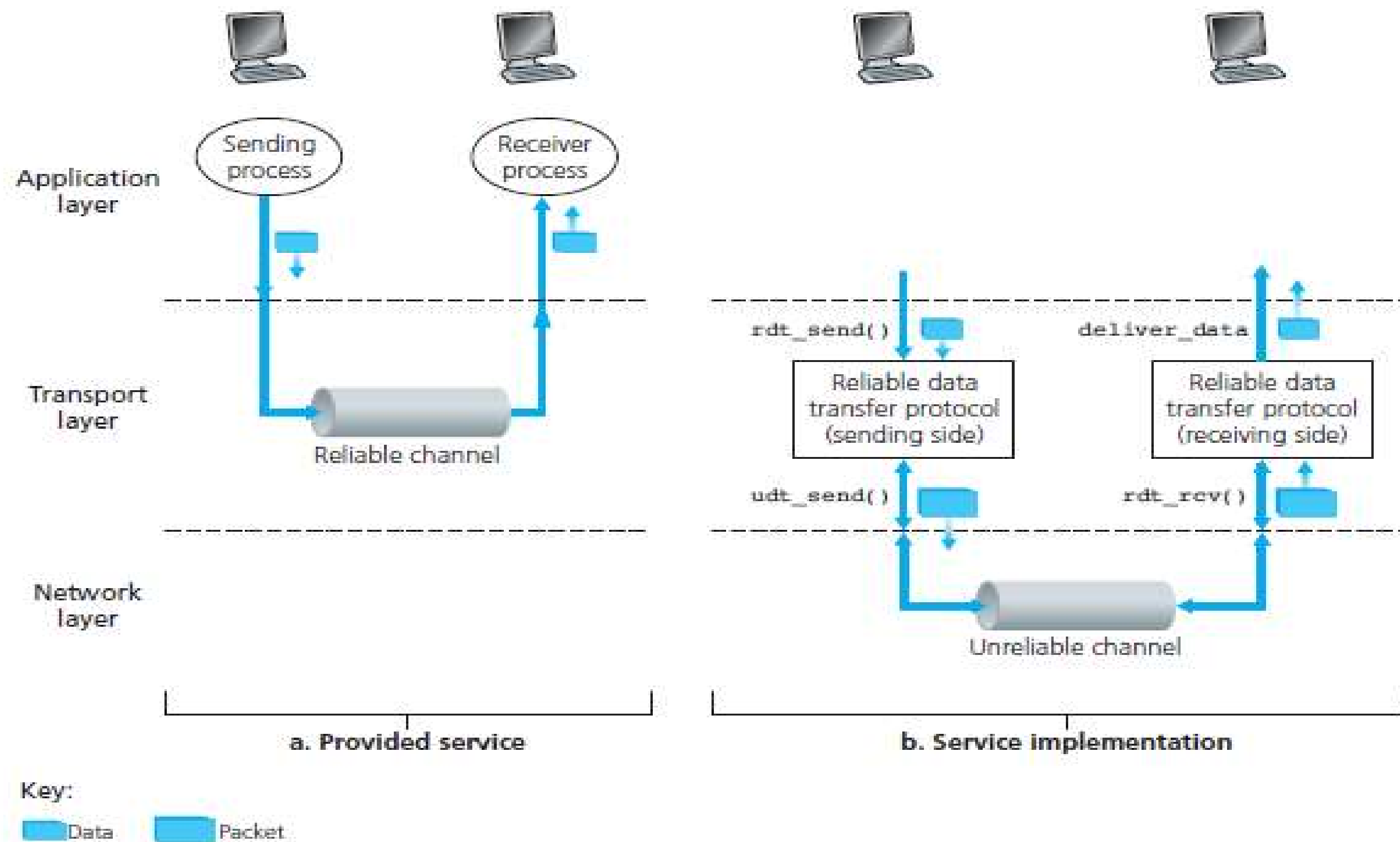
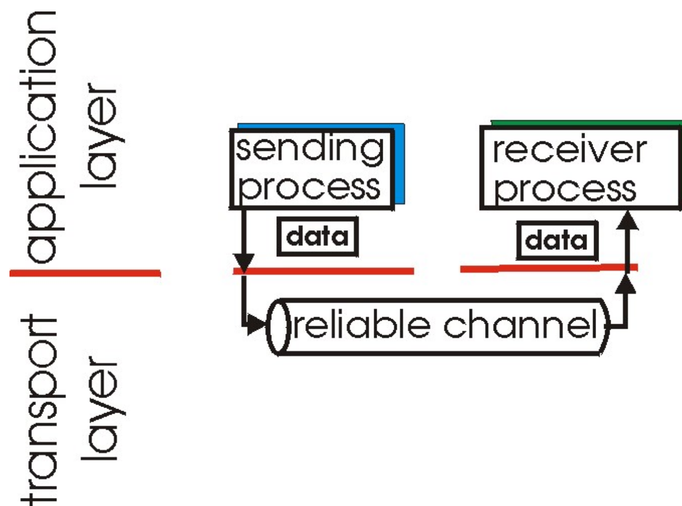



**Figure 3.8** ♦ Reliable data transfer: Service model and service implementation

# Principles of reliable data transfer

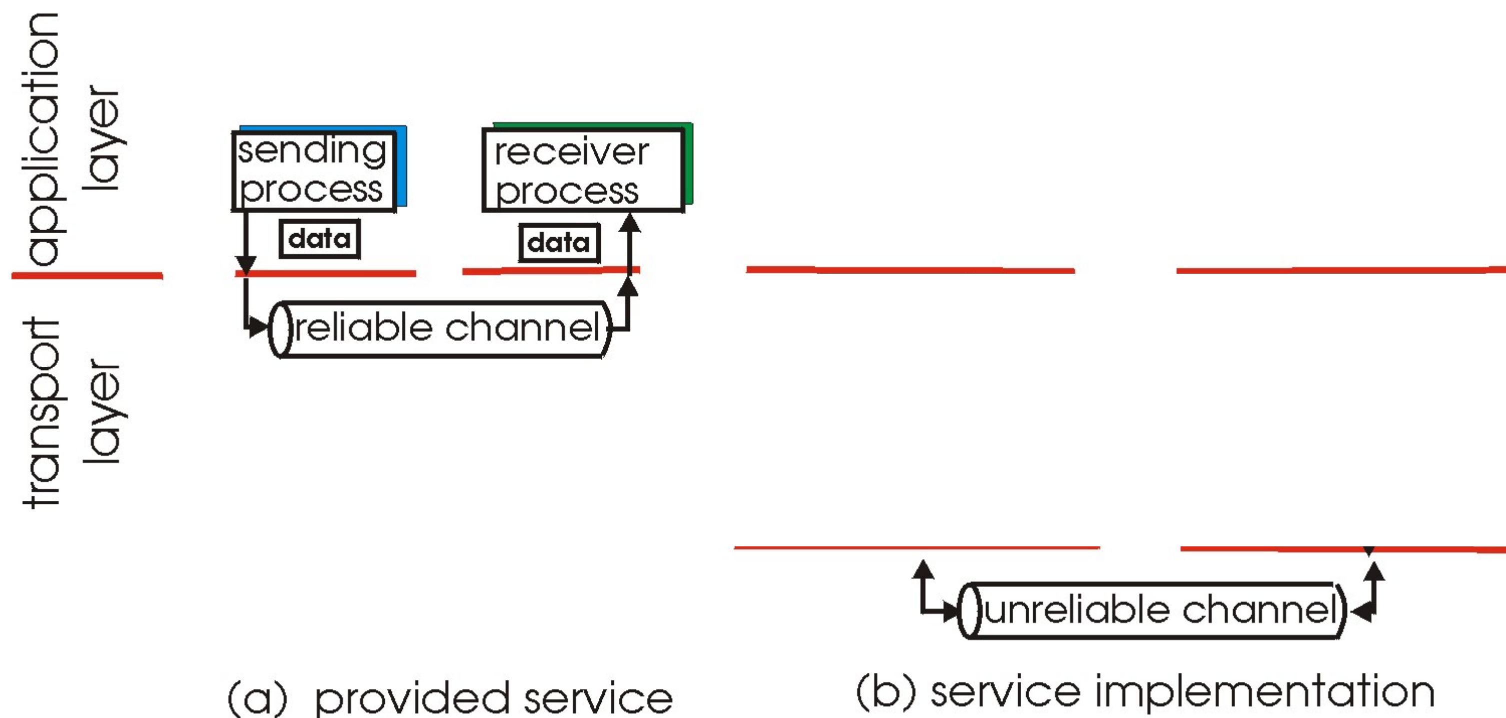
- important in application, transport, link layers




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

# Principles of reliable data transfer

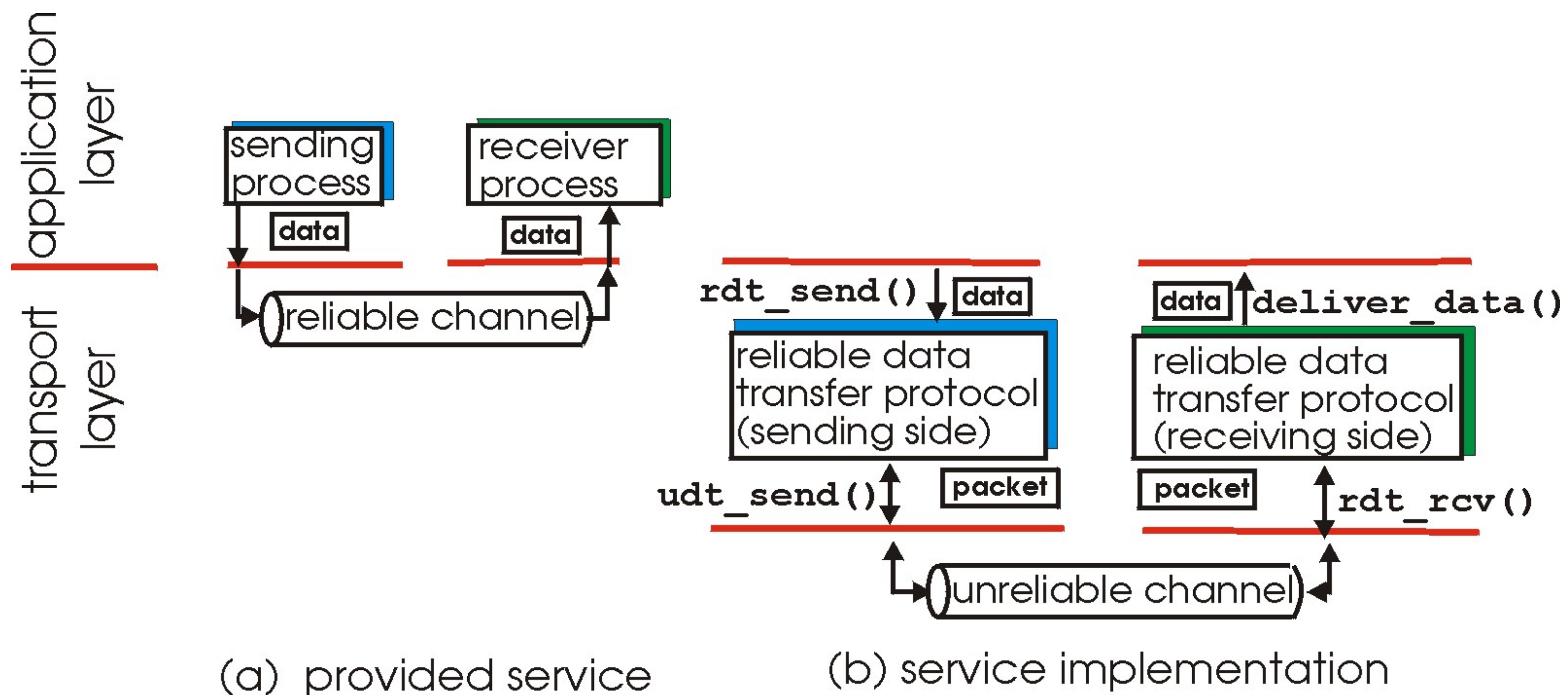
- important in application, transport, link layers




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

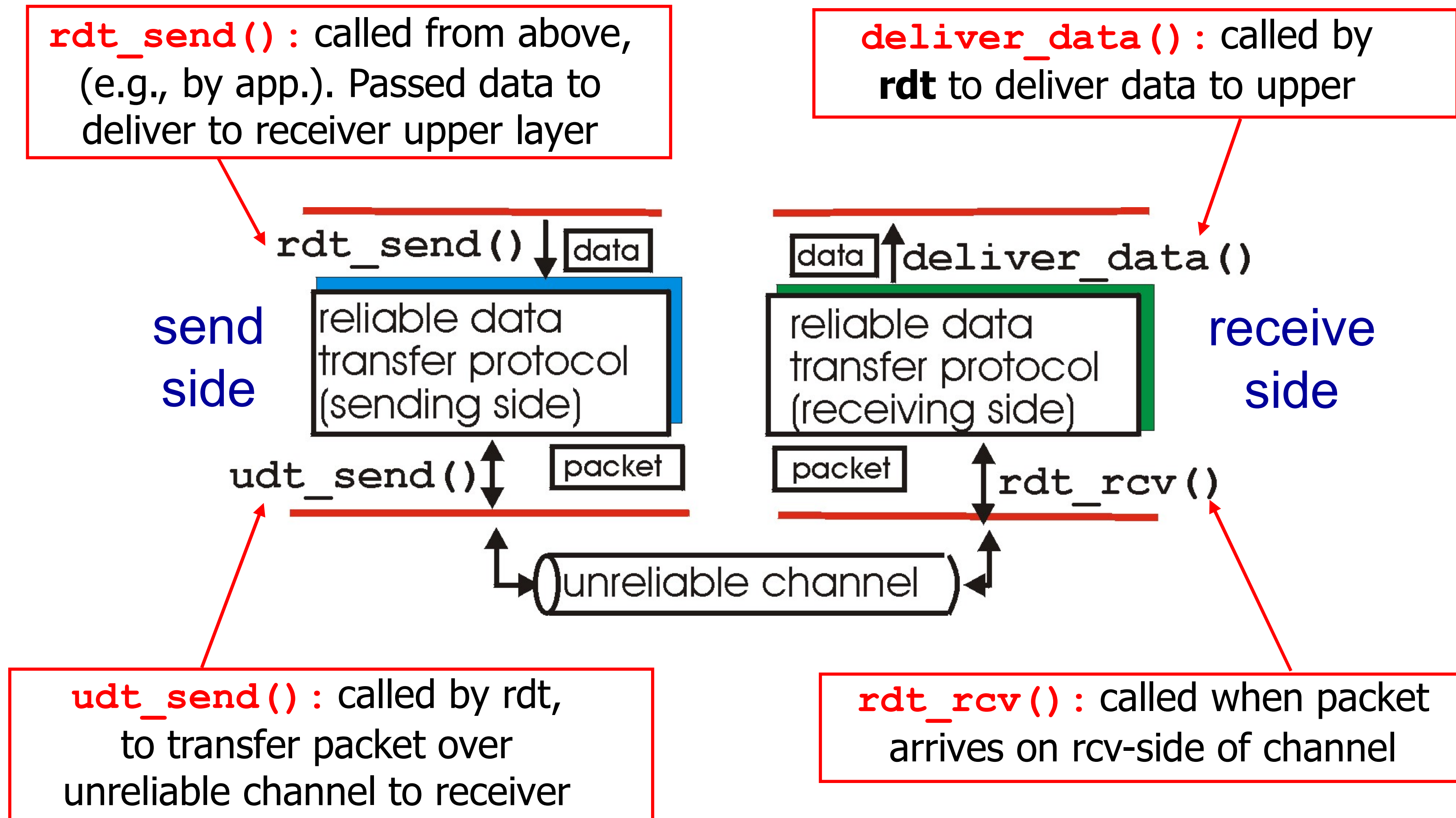
# Principles of reliable data transfer

- important in application, transport, link layers




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

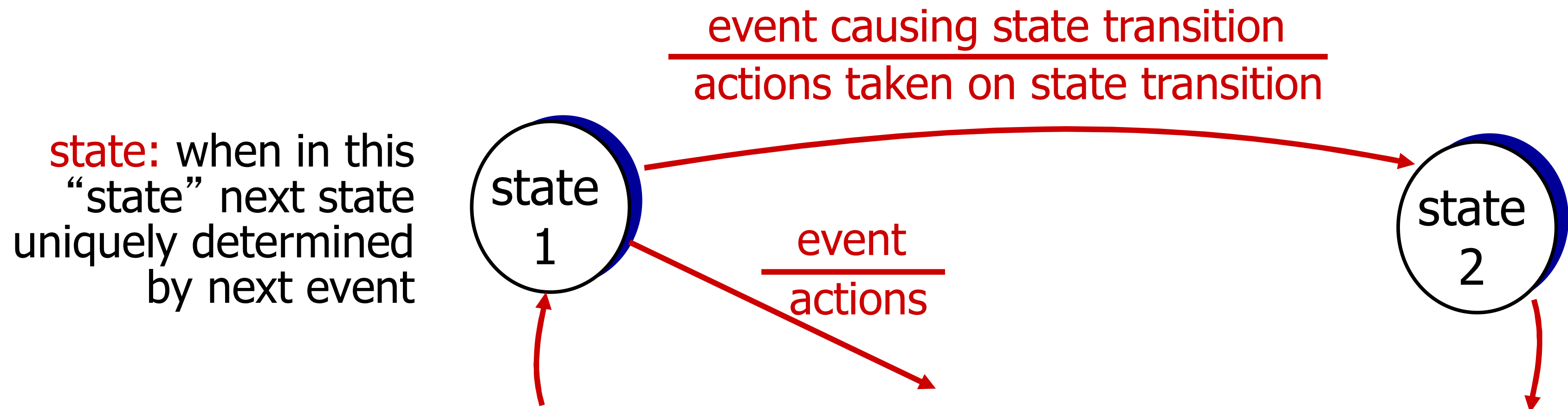
# Reliable data transfer: getting started



# Reliable data transfer: 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

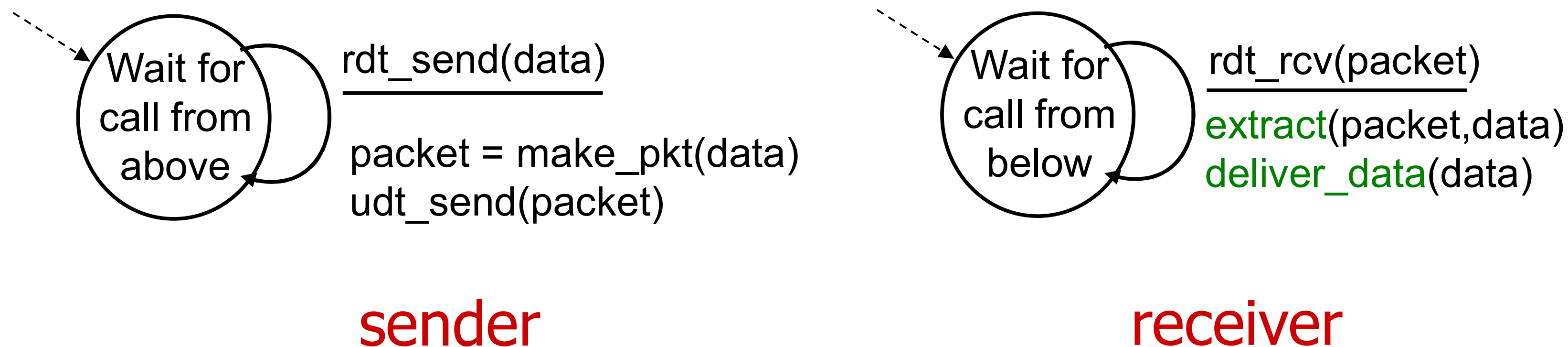


# Finite State Machines

- A **finite state machine** or **finite automaton** is a model of behavior composed of states, transitions and actions.
  - A **state** stores information about the past, i.e. it reflects the input changes from the system start to the present moment.
  - A **transition** indicates a **state change** and is described by a **condition/event** that would need to be fulfilled to enable the transition.
  - An **action** is a description of an **activity** that is to be performed at a given moment.

# 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



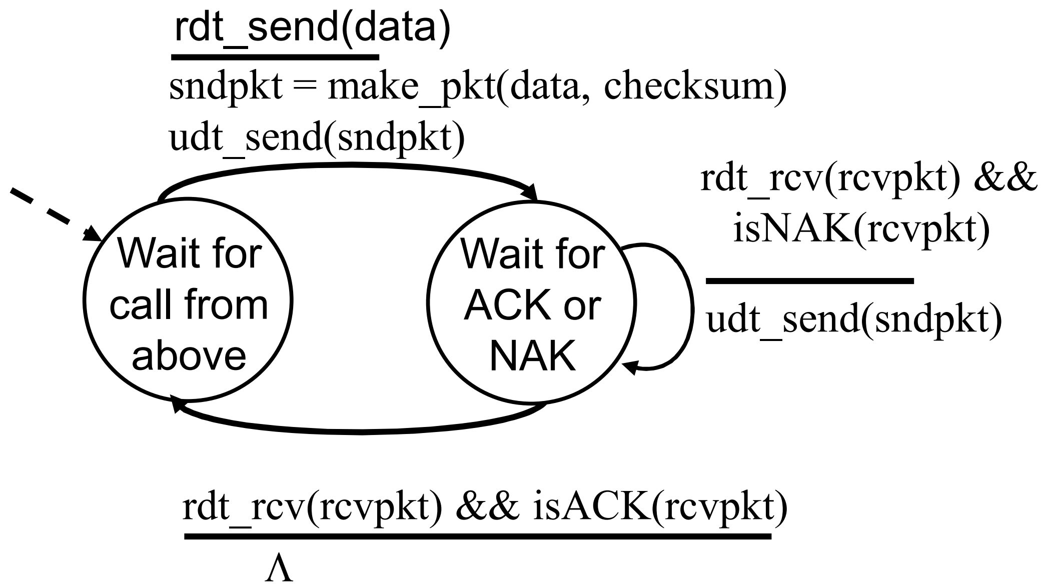

sender

receiver

# rdt2.0: channel with bit errors(ARQ)

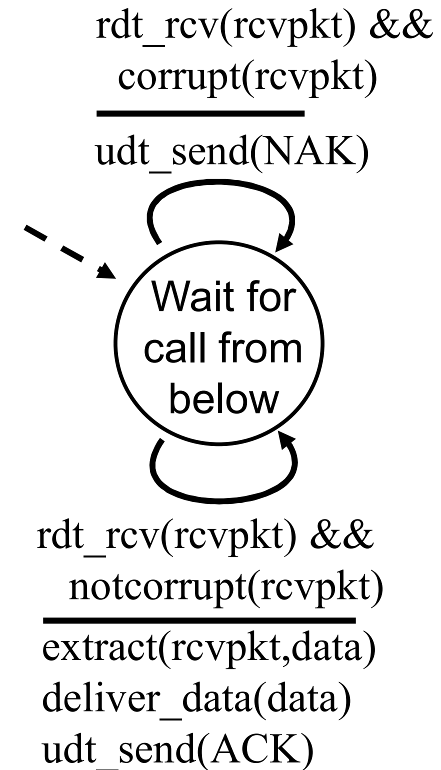
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors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

# rdt2.0: FSM specification

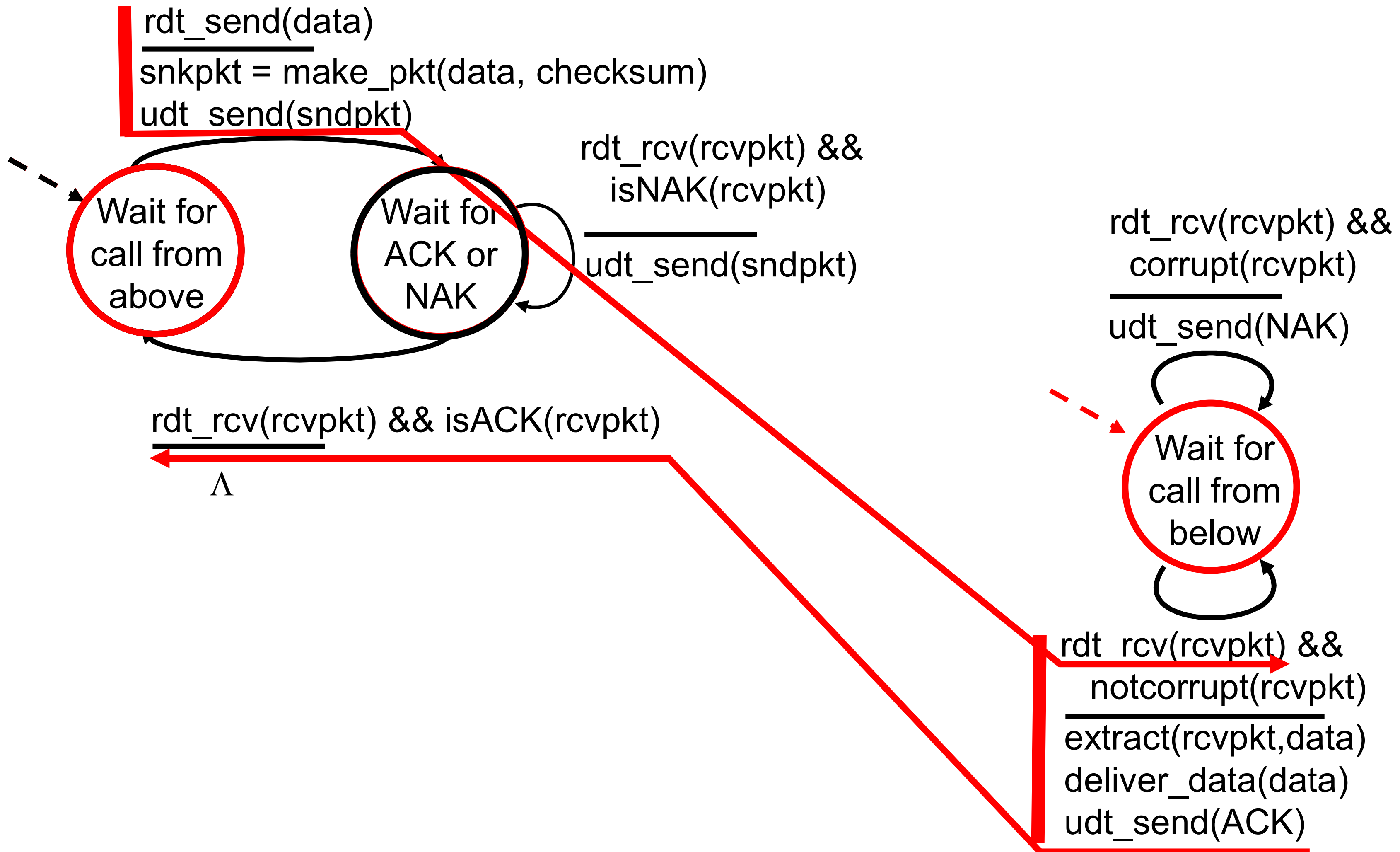



sender

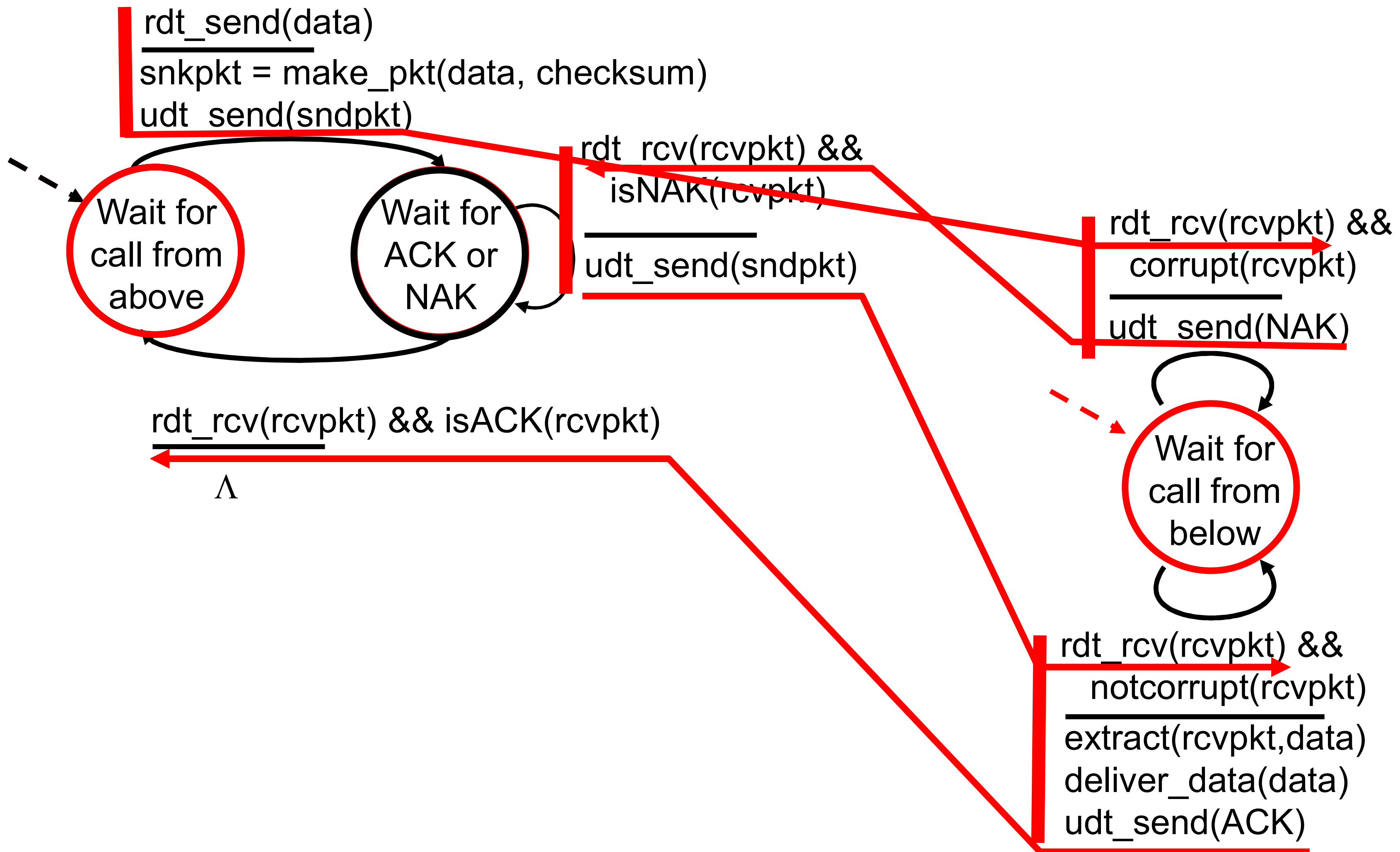
receiver



# rdt2.0: operation with no errors



# rdt2.0: error scenario



# rdt2.0 has a fatal flaw!

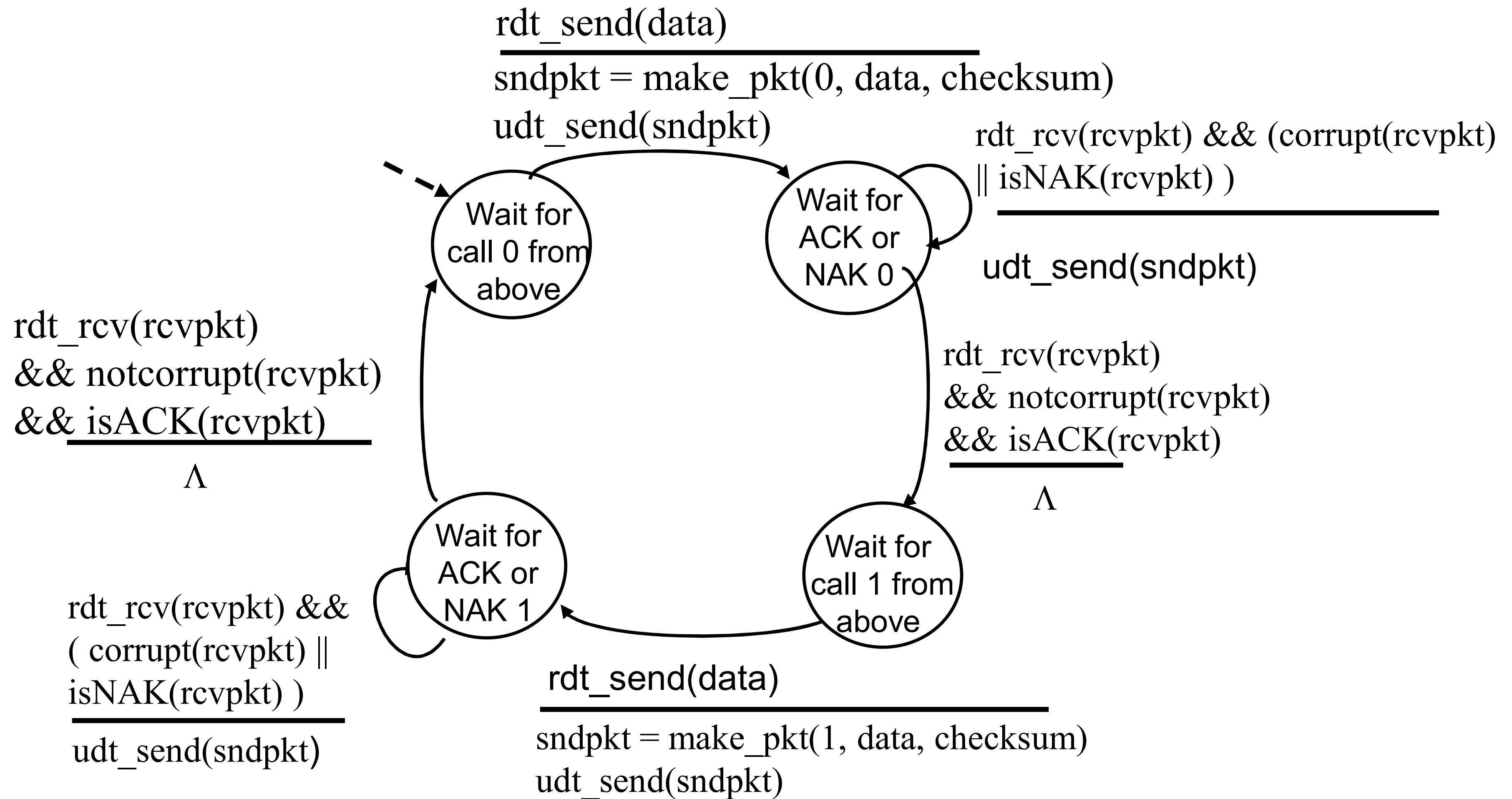
what happens if ACK/NAK corrupted?

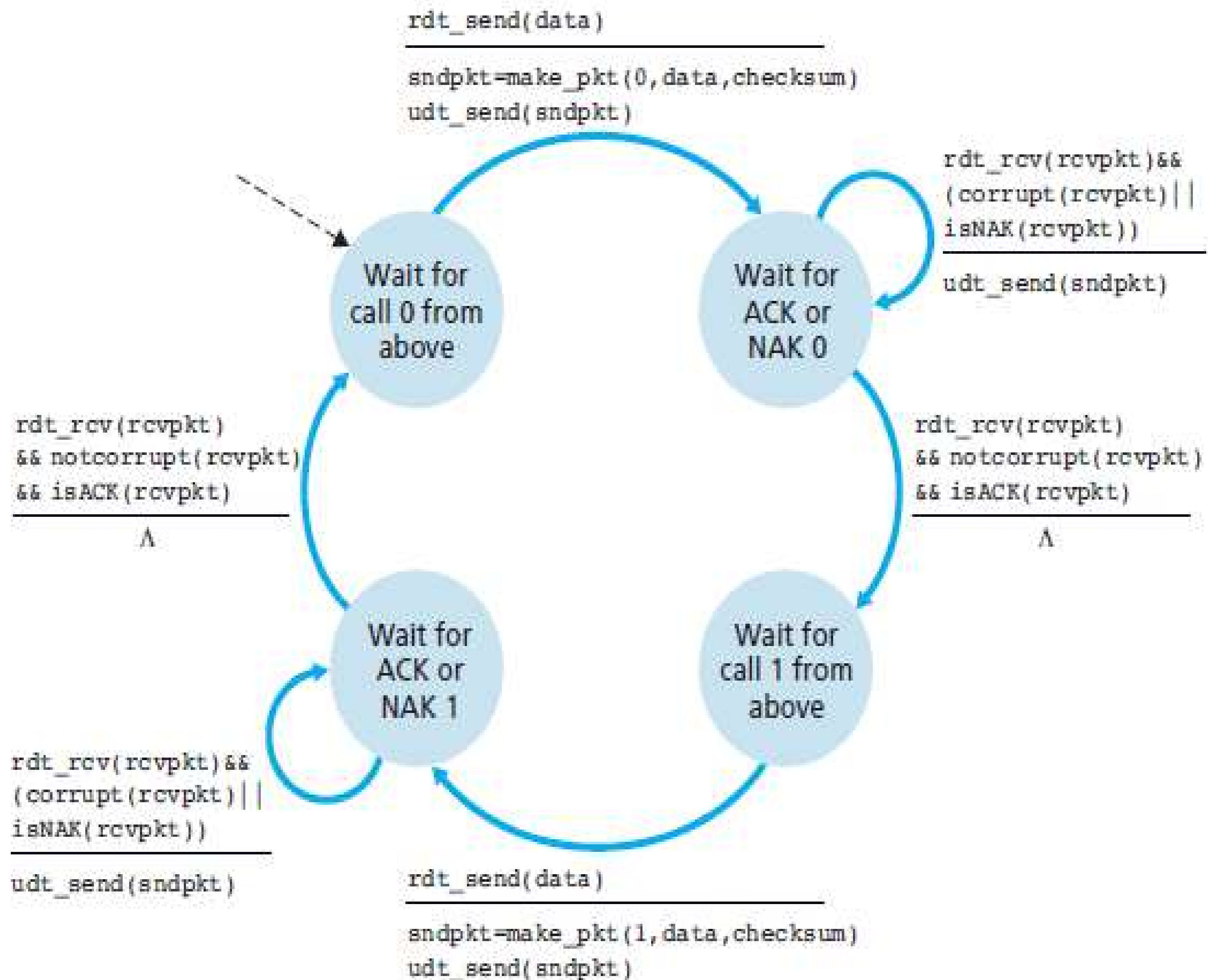
- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

handling duplicates:

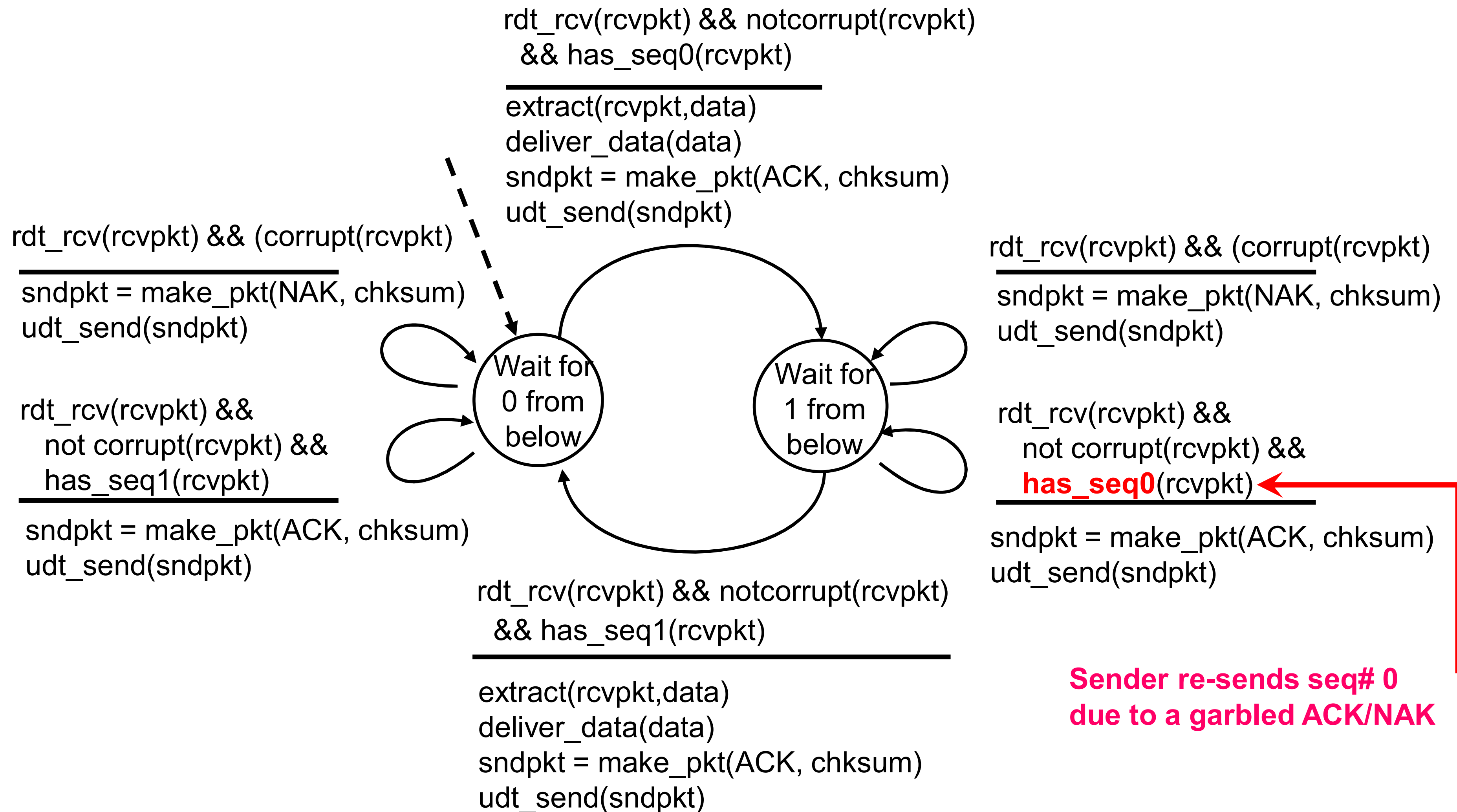
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

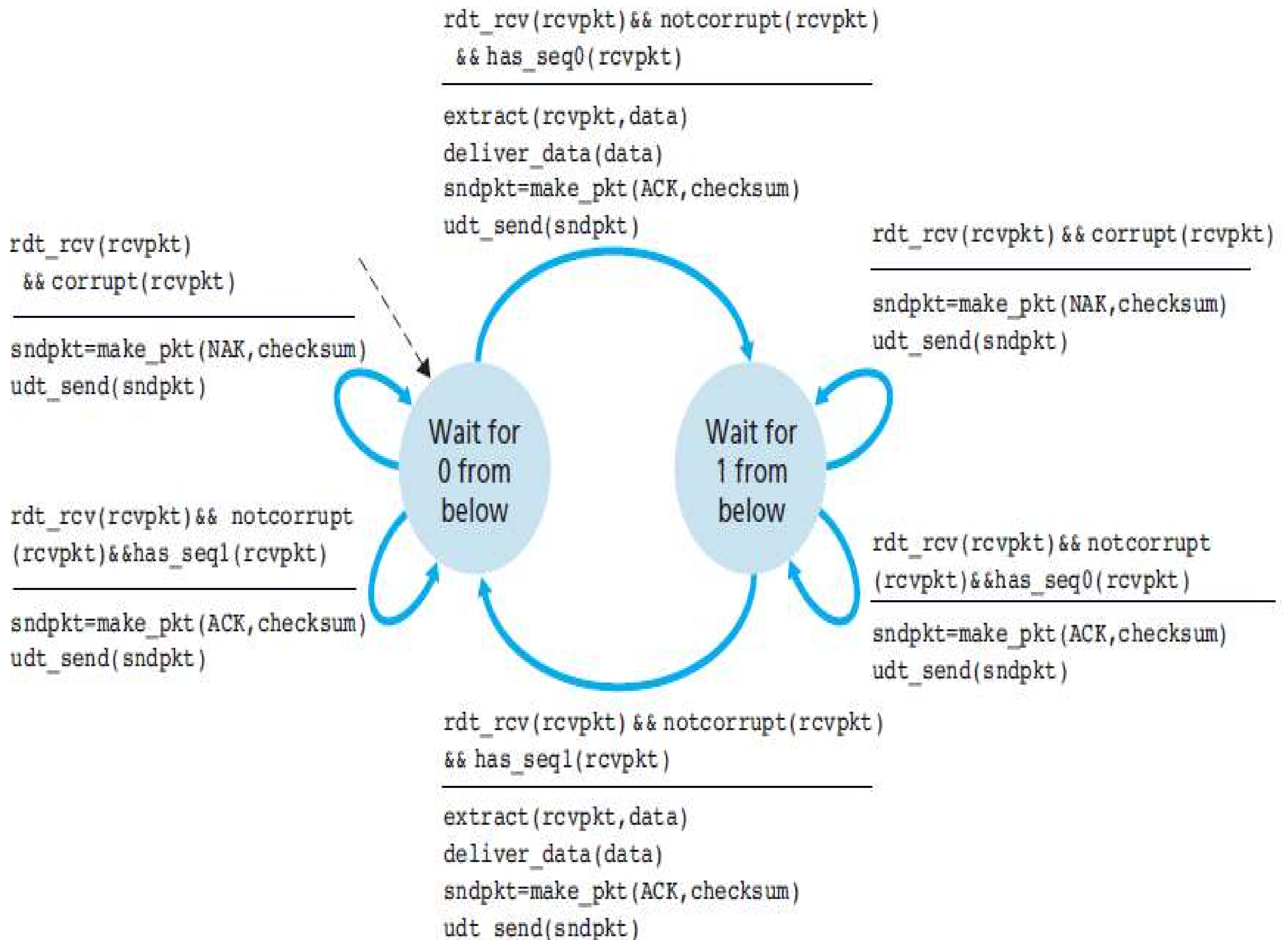
# rdt2.1: sender, handles garbled ACK/NAKs




rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)&&
(corrupt(rcvpkt)||
isNAK(rcvpkt))
udt_send(sndpkt)
Wait for
call 0 from
above
Wait for
ACK or
NAK 0
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
Λ
Wait for
ACK or
NAK 1
Wait for
call 1 from
above
rdt_rcv(rcvpkt)&&
(corrupt(rcvpkt)||
isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)

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




rdt_rcv(rcvpkt)&& notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
Wait for
0 from
below
Wait for
1 from
below
rdt_rcv(rcvpkt)&& notcorrupt
(rcvpkt)&&has_seq1(rcvpkt)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)&& notcorrupt
(rcvpkt)&&has_seq0(rcvpkt)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)

# rdt2.1: discussion

sender:

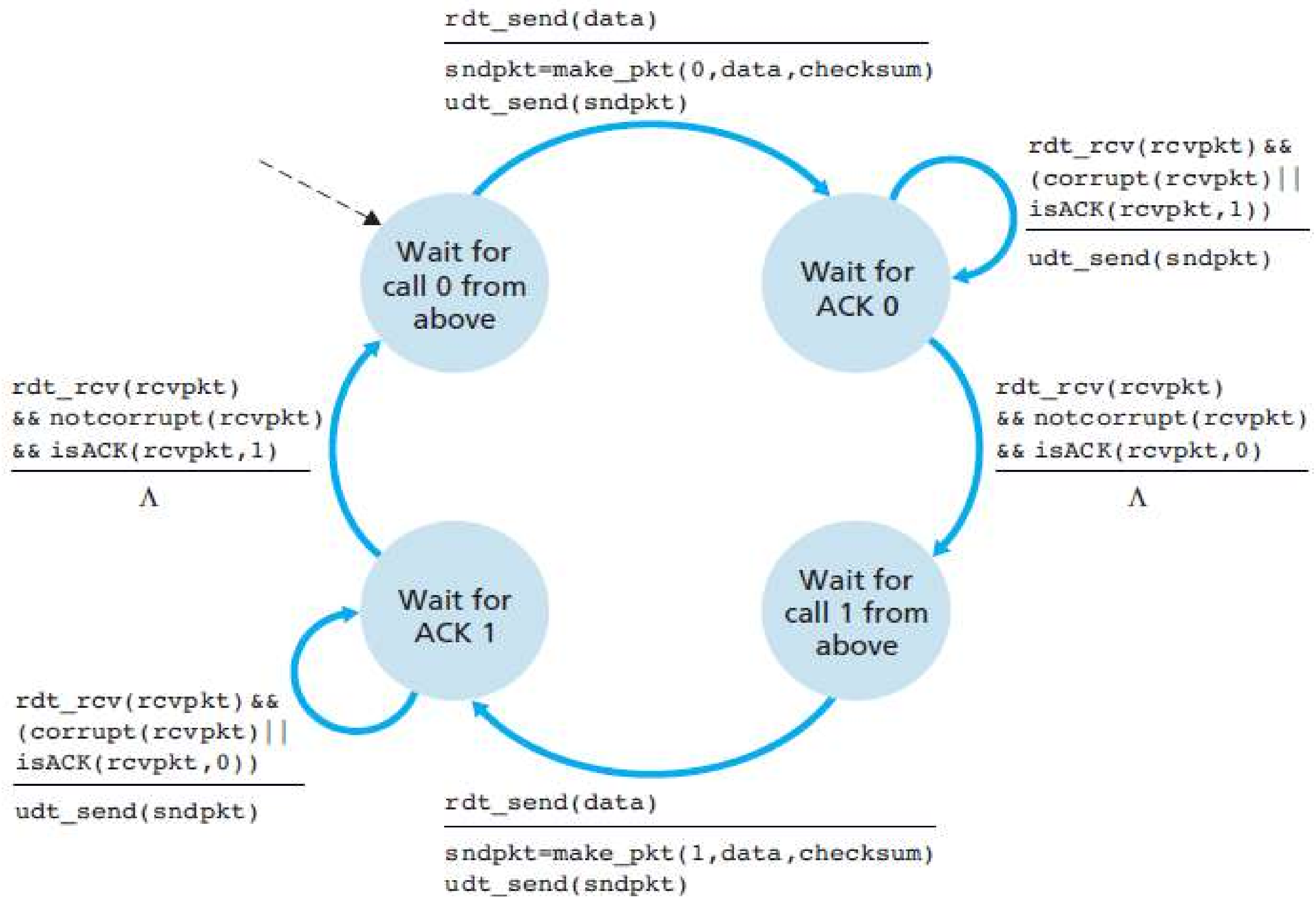
- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

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*




ure 3.13 ♦ rdt2.2 sender

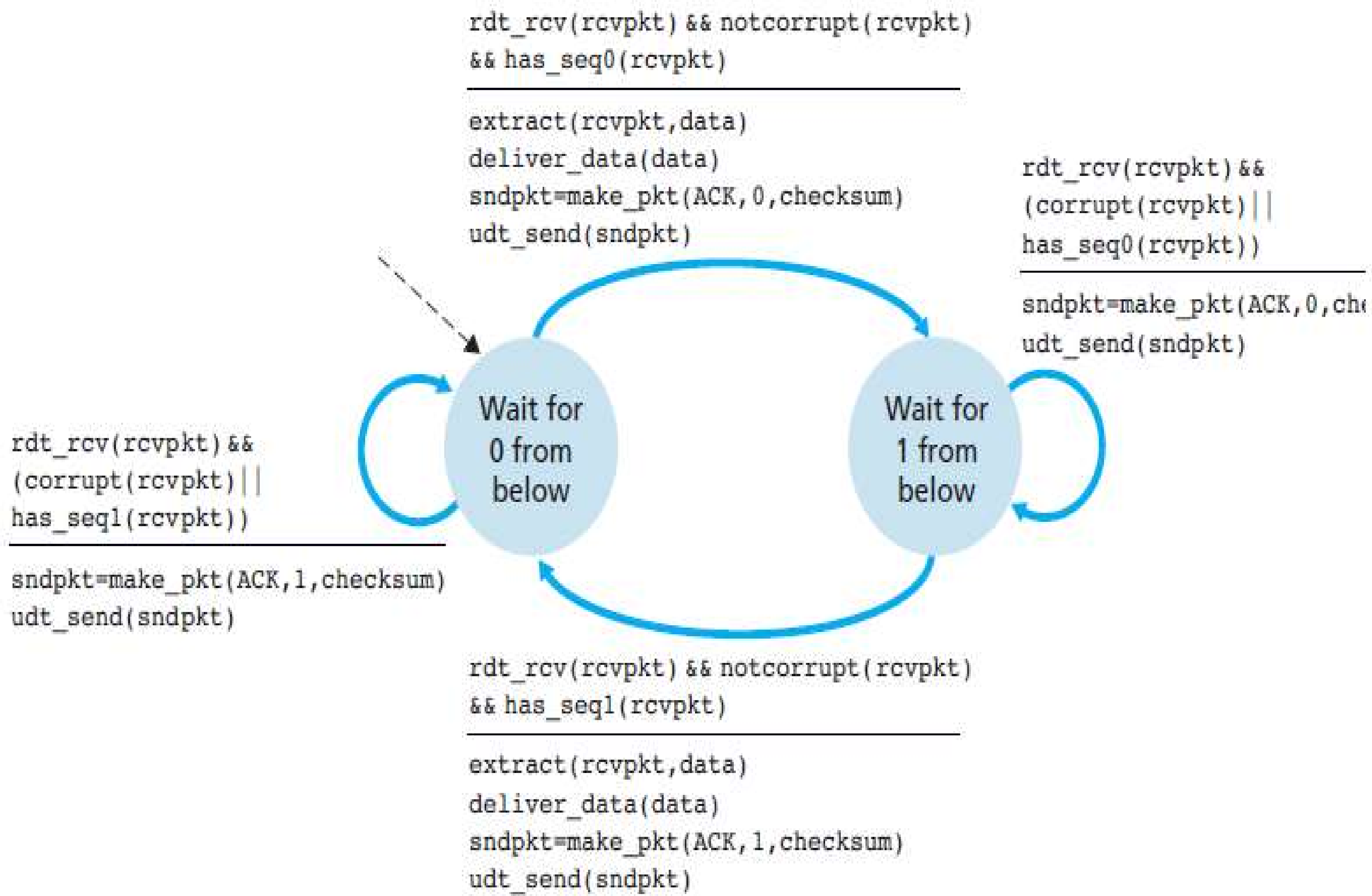
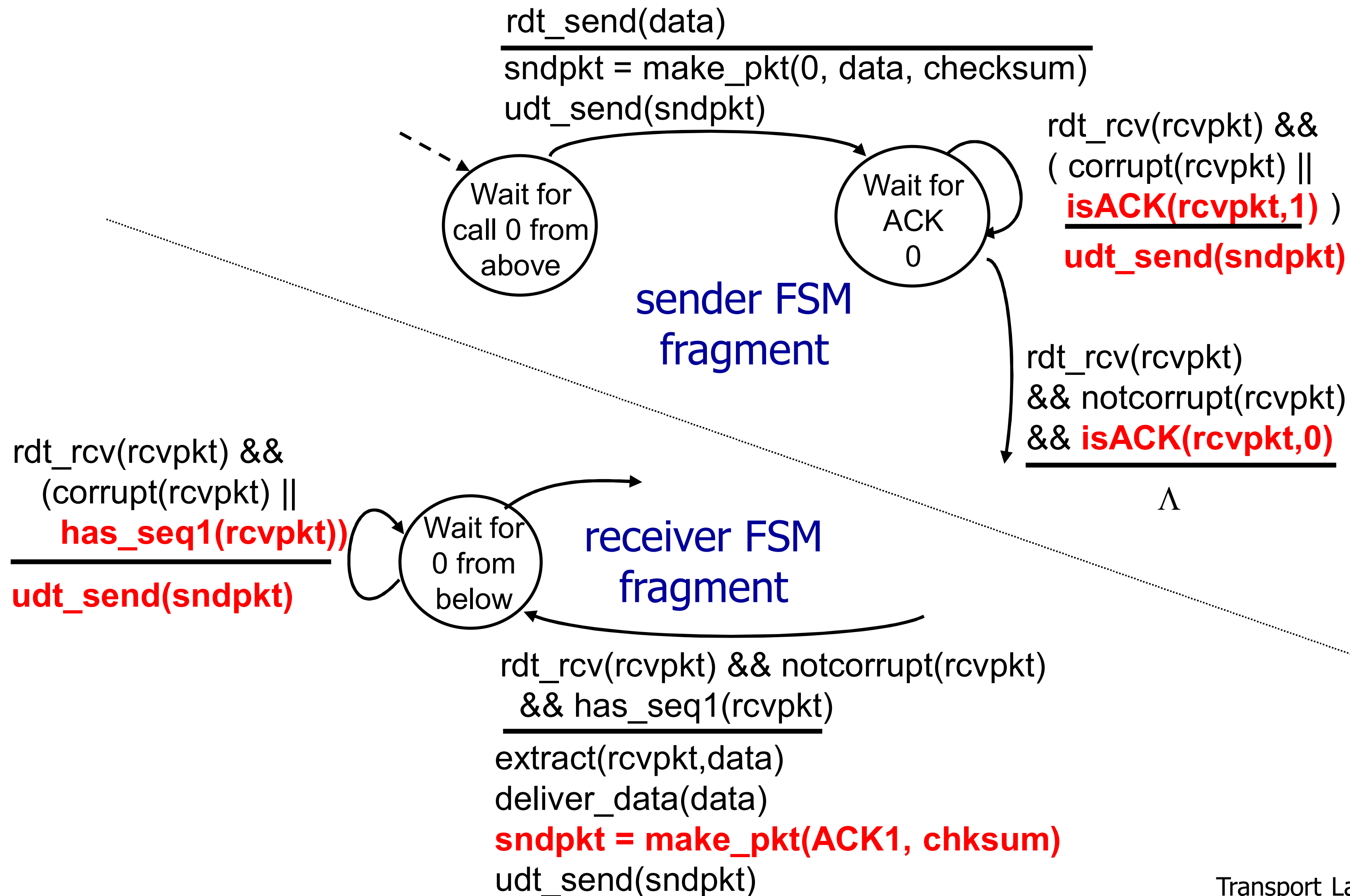



**Figure 3.14** ♦ rdt2.2 receiver

# rdt2.2: sender, receiver fragments



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

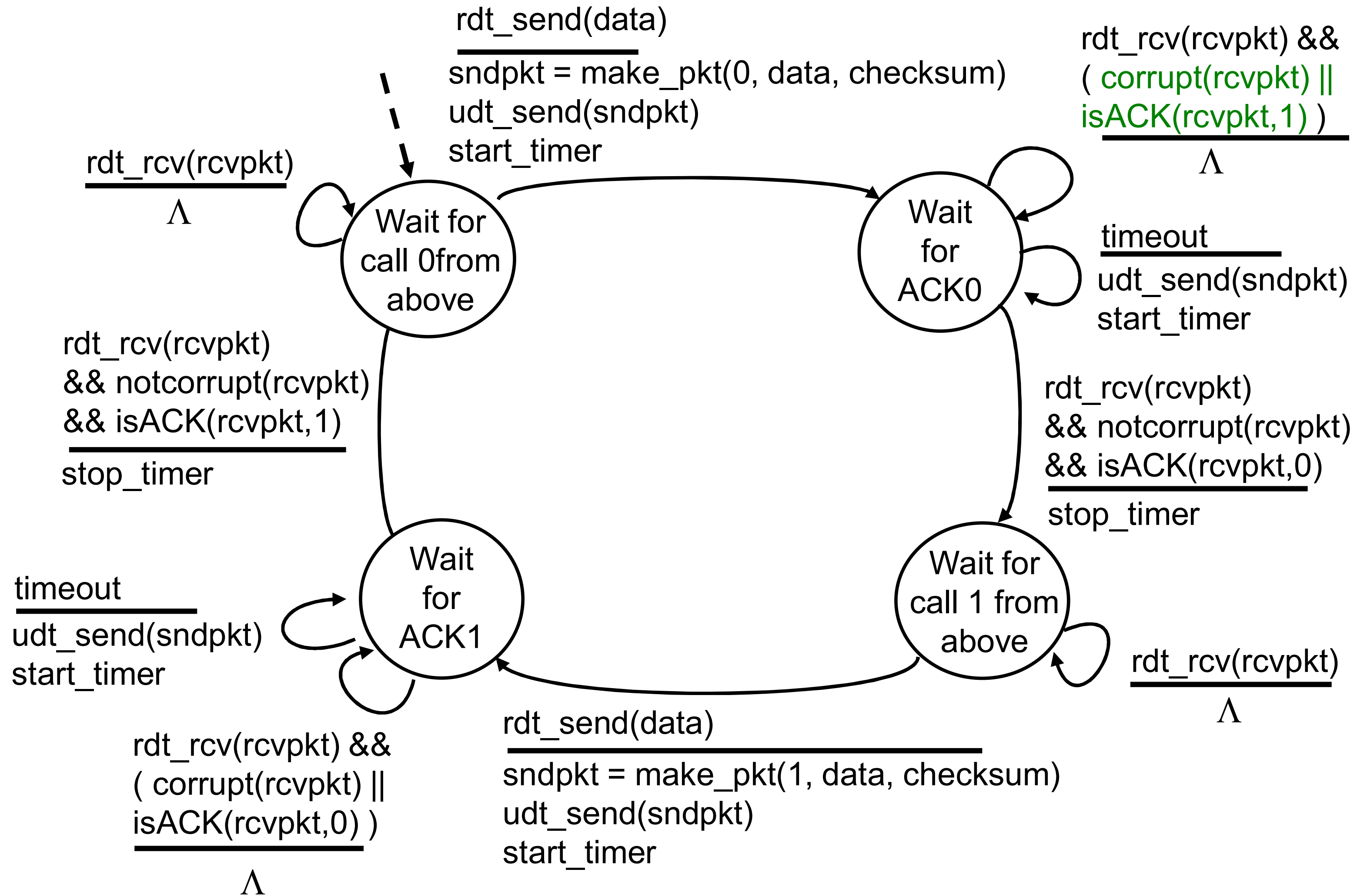
new assumption: underlying channel can also lose packets (data, ACKs)

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

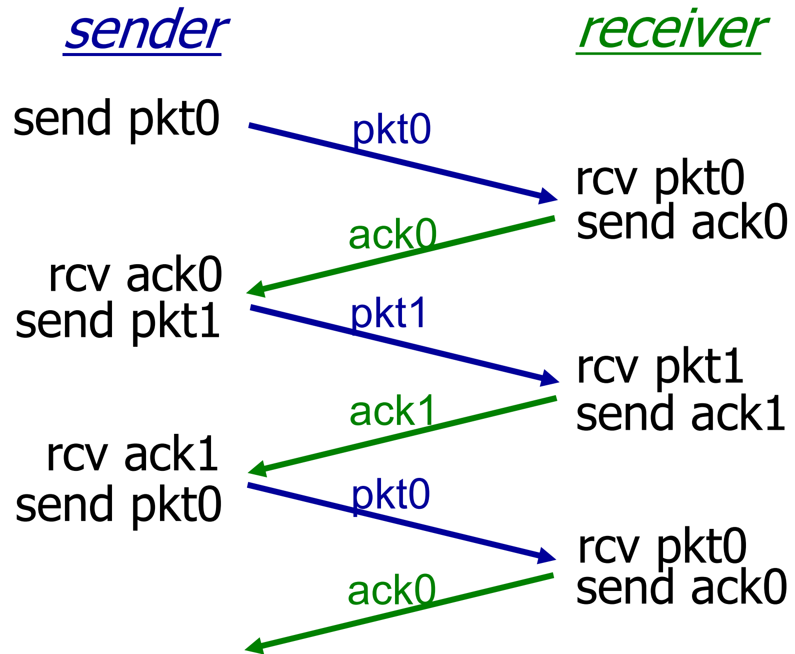
approach: sender waits "reasonable" amount of time for ACK

- 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

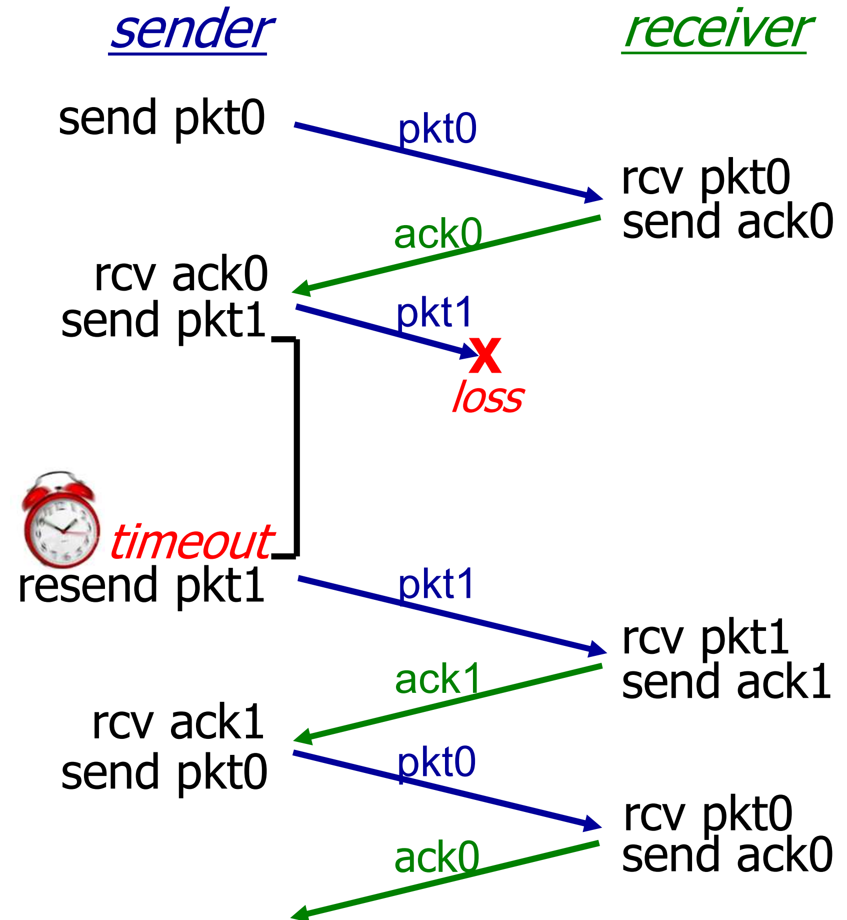
# rdt3.0 sender



# rdt3.0 in action

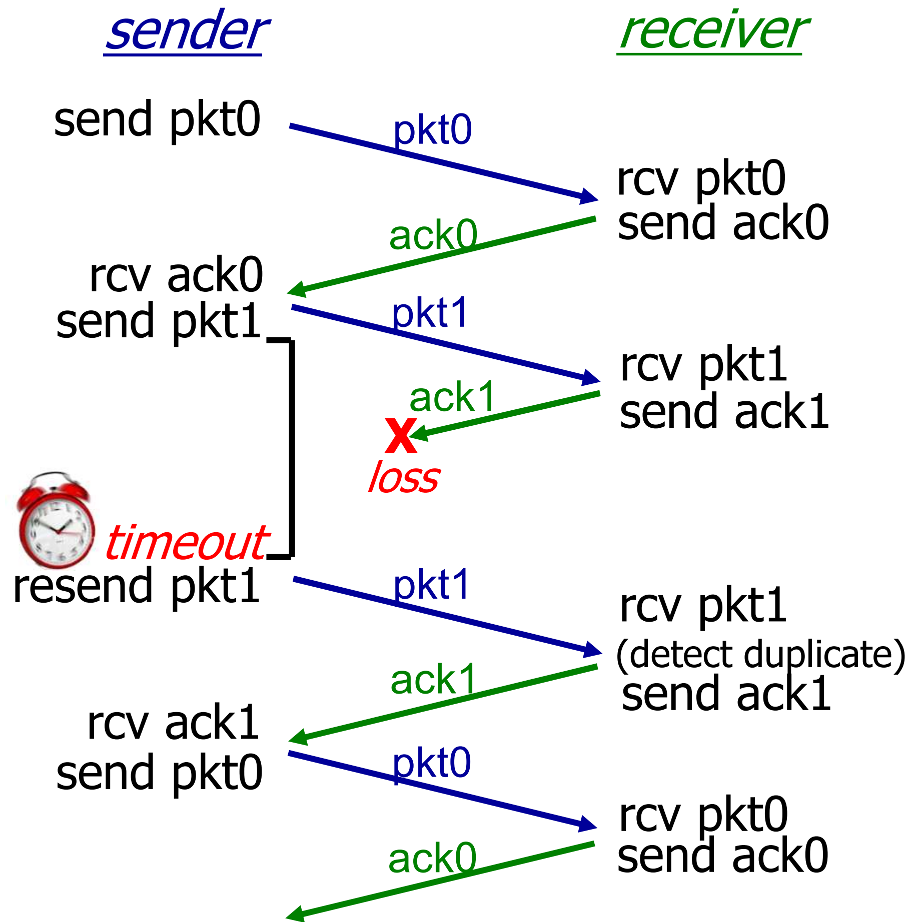



(a) no loss

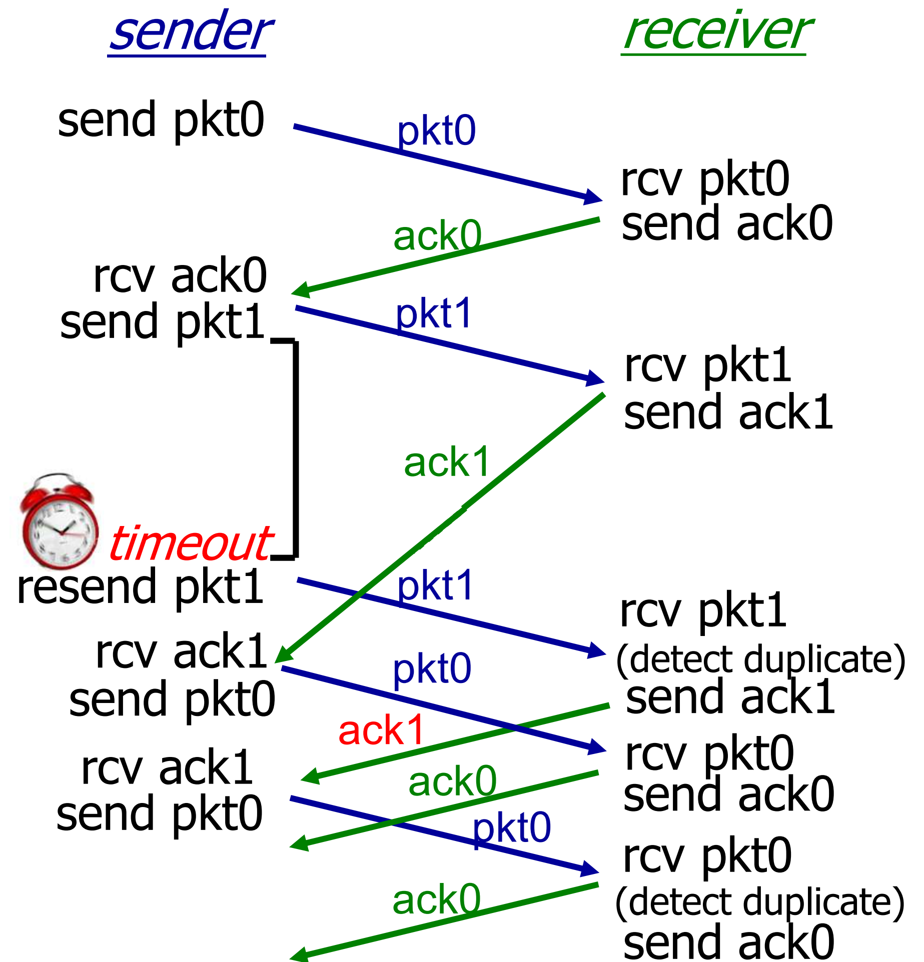



(b) packet loss

# 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:

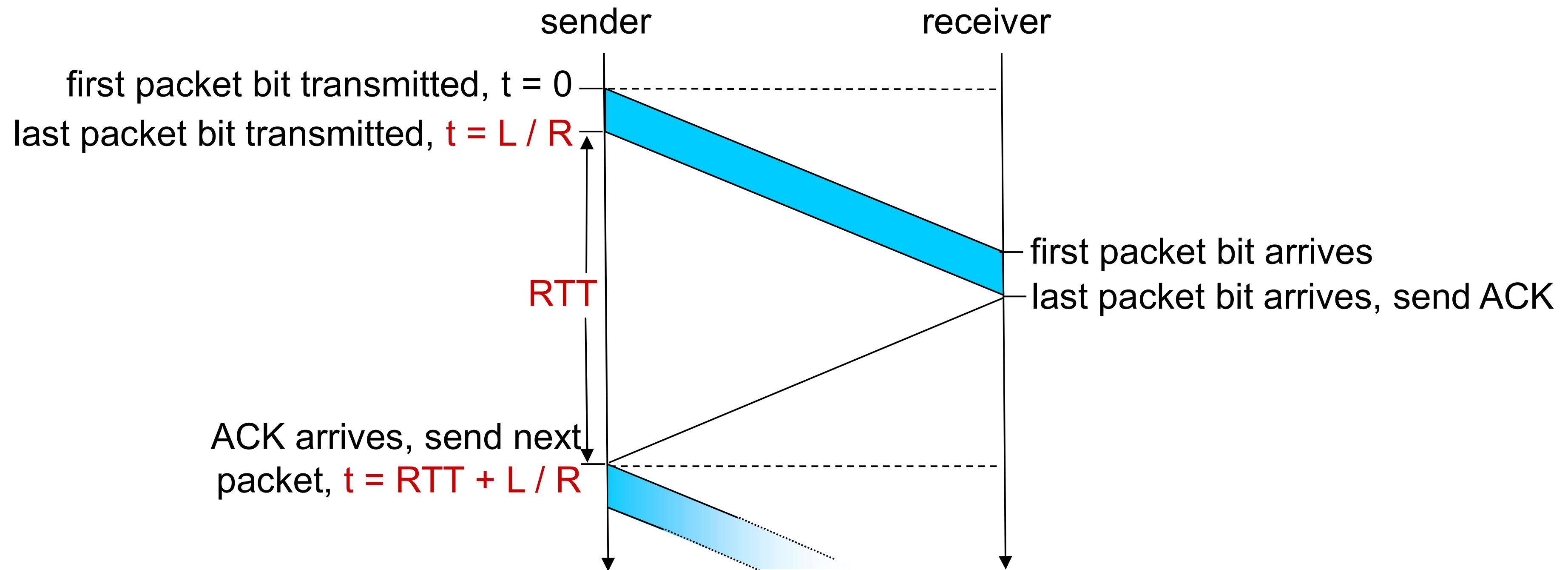
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - $U_{sender}$: *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 throughput over 1 Gbps link
- network protocol limits use of physical resources!

# rdt3.0: stop-and-wait operation

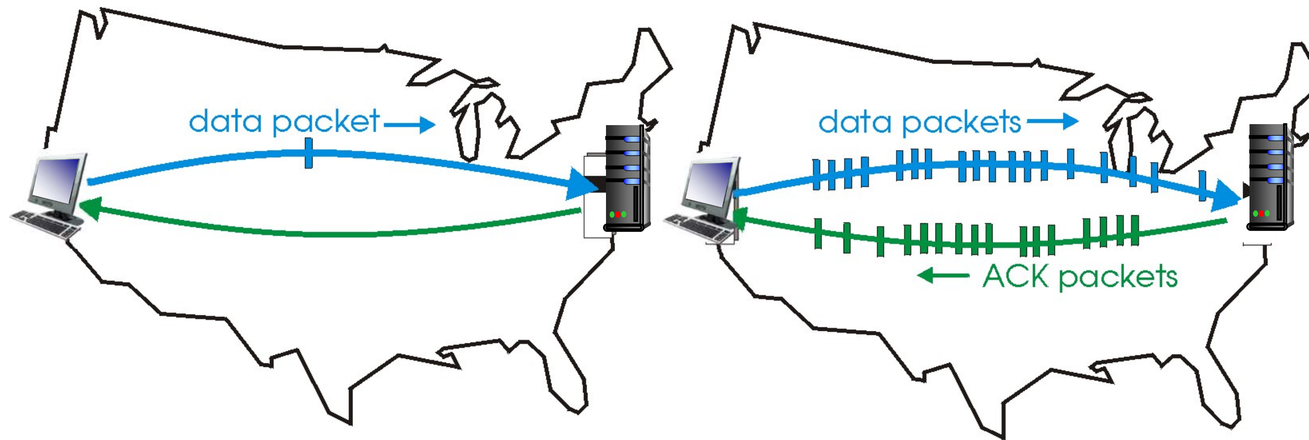



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# Pipelined protocols

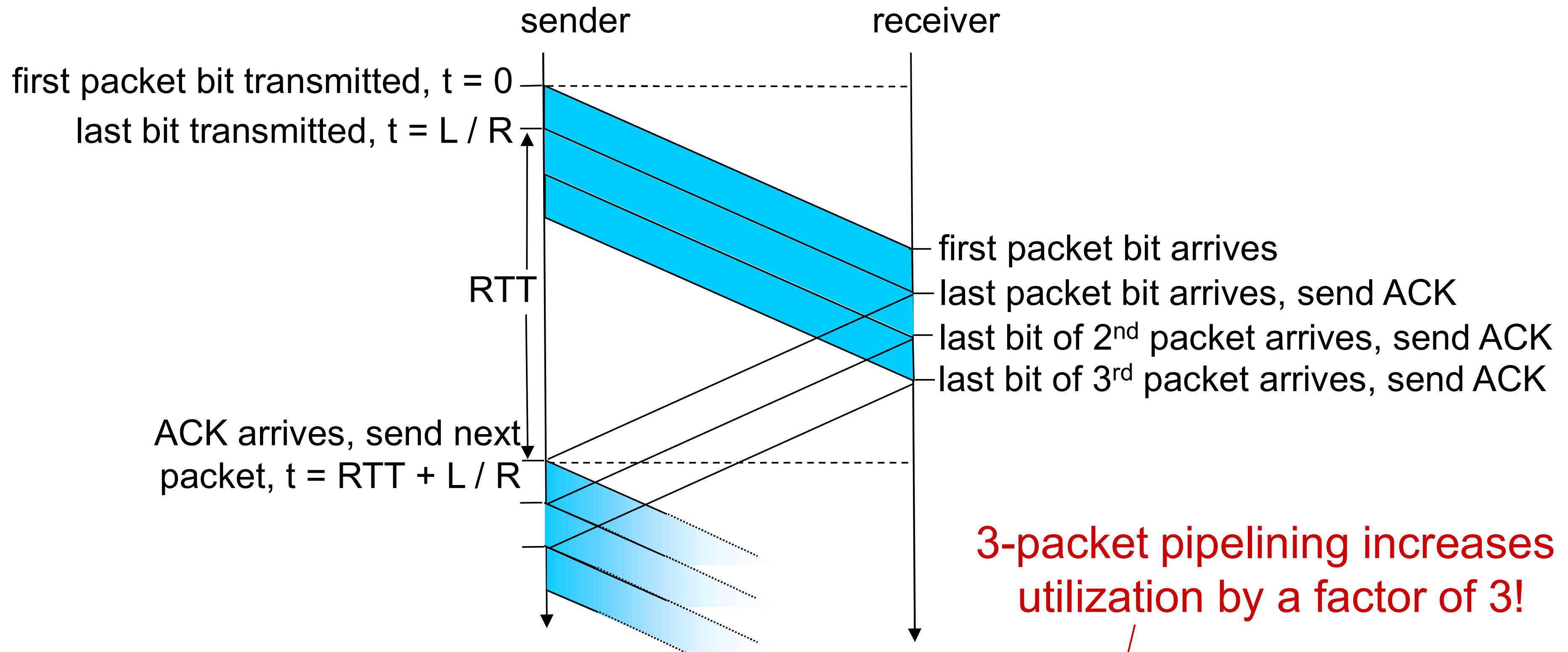
pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

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




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

# Pipelining: increased utilization




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

# Pipelined protocols: overview

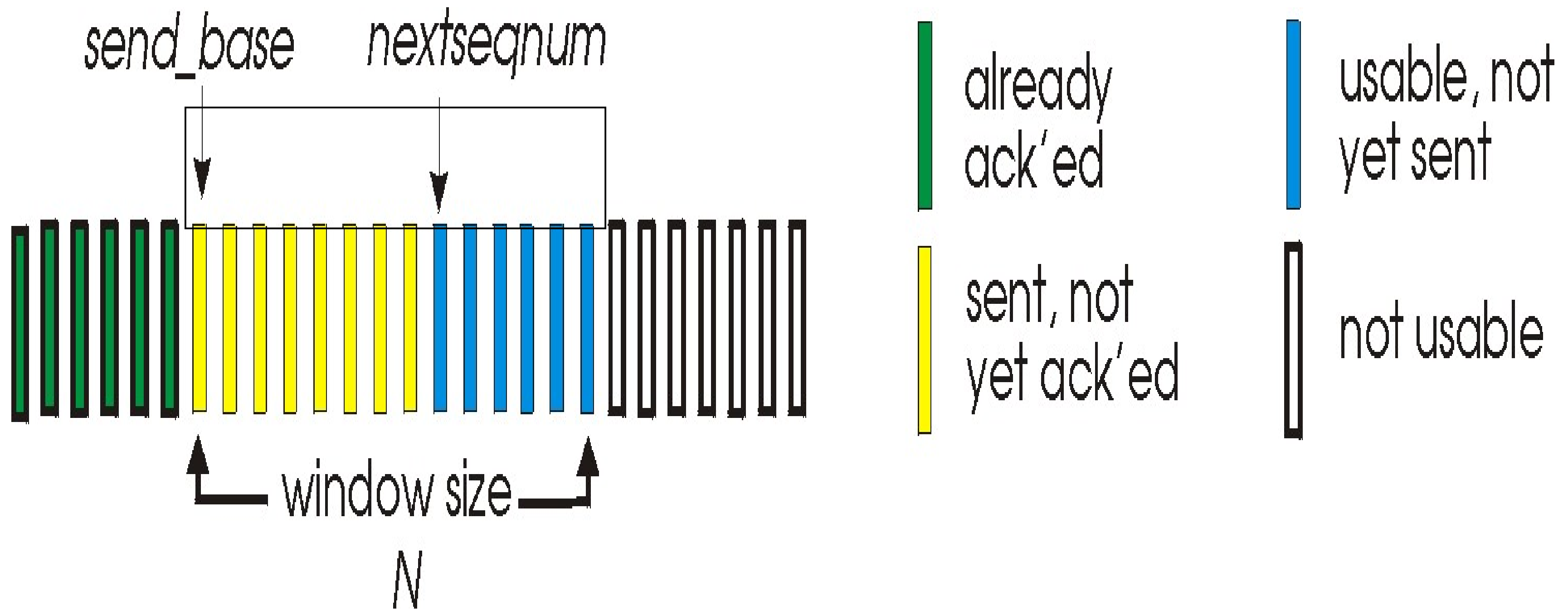
## Go-back-N:

- 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

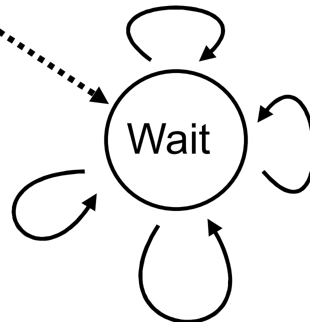
## Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- 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's view of seq. no in GBN



# 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)

$\Lambda$

---

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)**

---

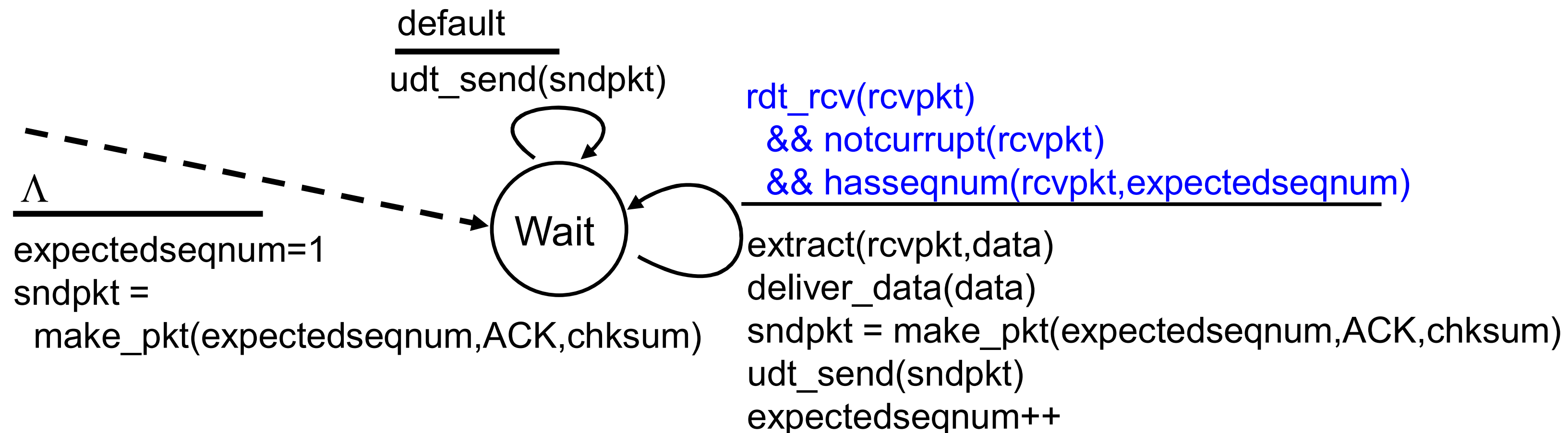
$\Lambda$

**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`**

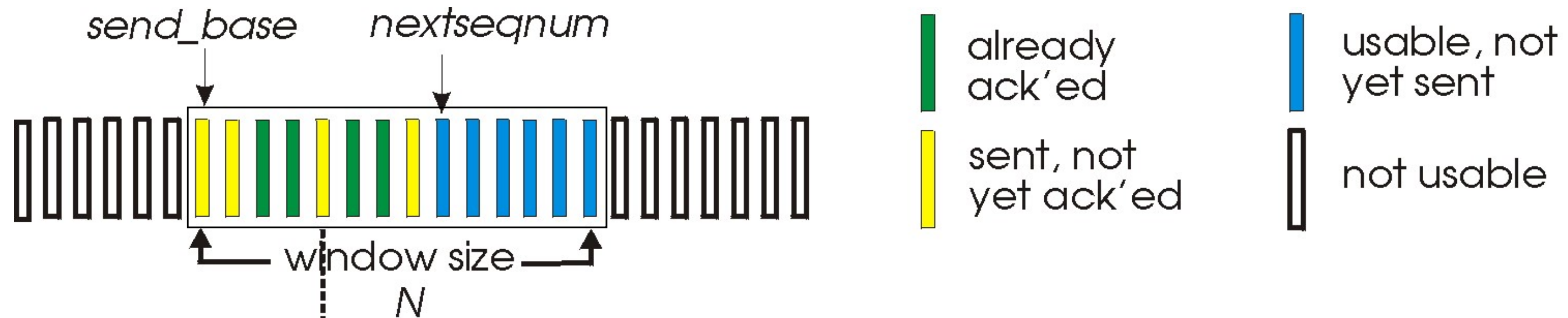
- out-of-order pkt:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

# GBN in action

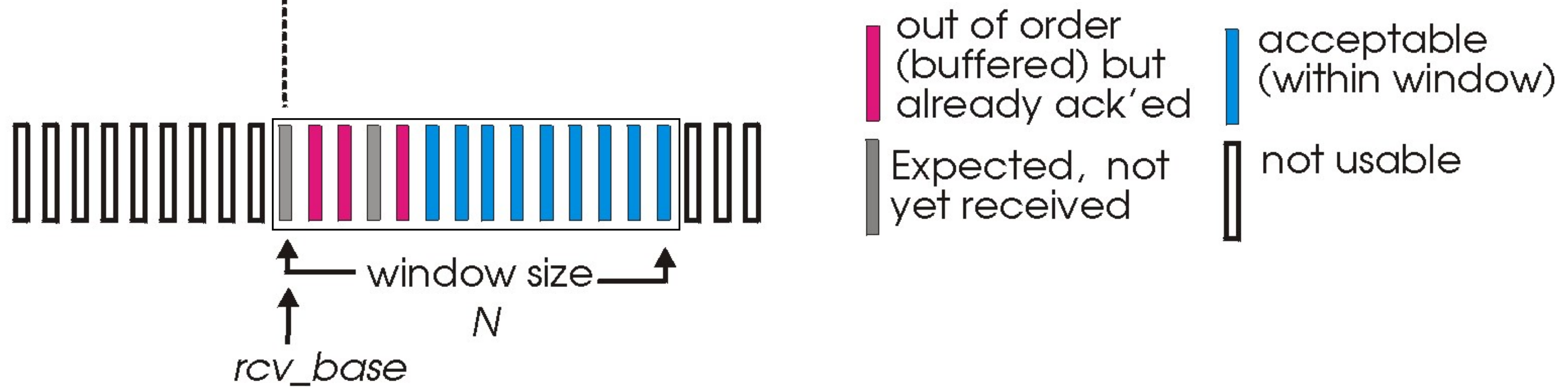
# Selective repeat

- 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

# 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 #

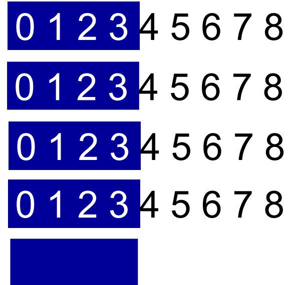
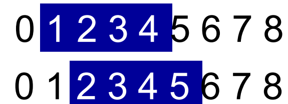
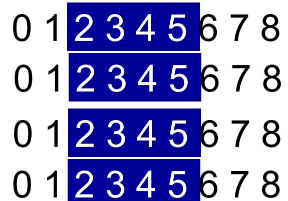
## receiver

**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

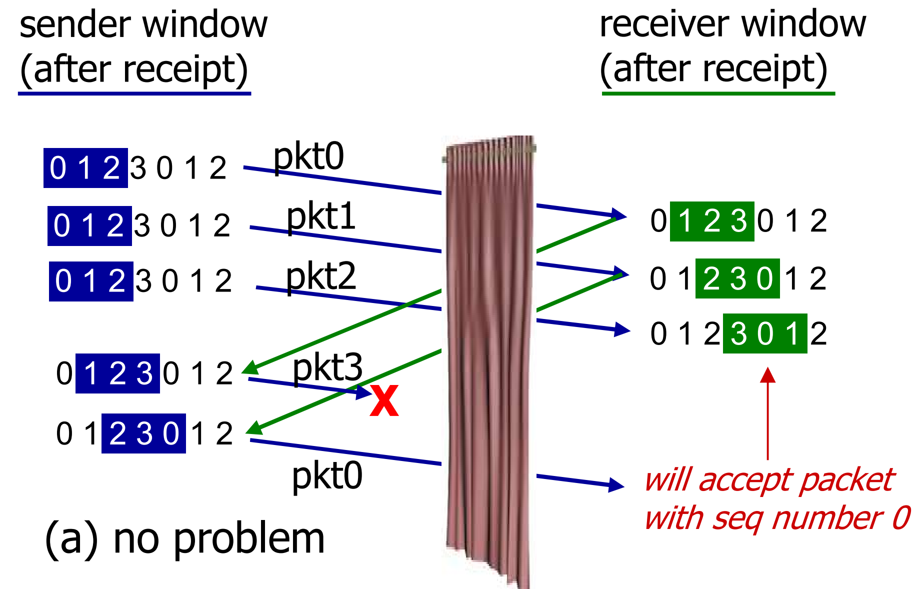
*Q: what happens when ack2 does not arrive?*

# Selective repeat: dilemma

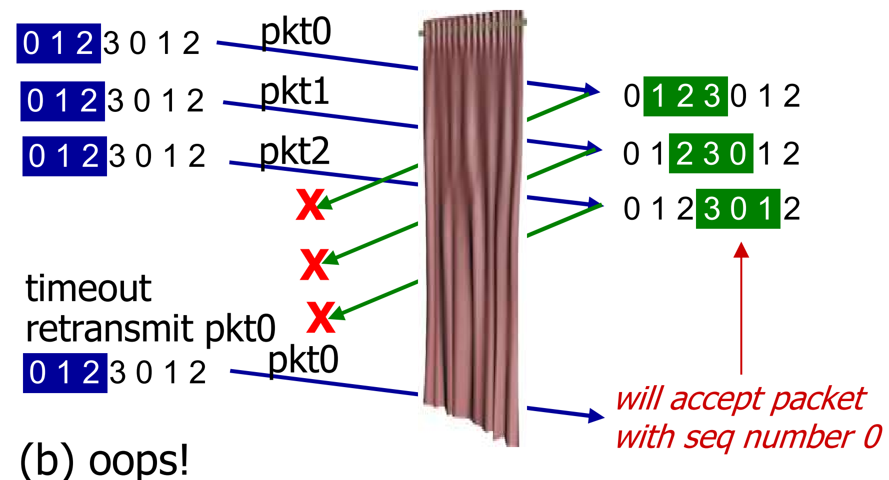
example:

- 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)?

A: window size <= ½(seq# size)

# Chapter 2 outline

2.1 transport-layer services

2.2 multiplexing and demultiplexing

2.3 connectionless transport: UDP

2.4 principles of reliable data transfer

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

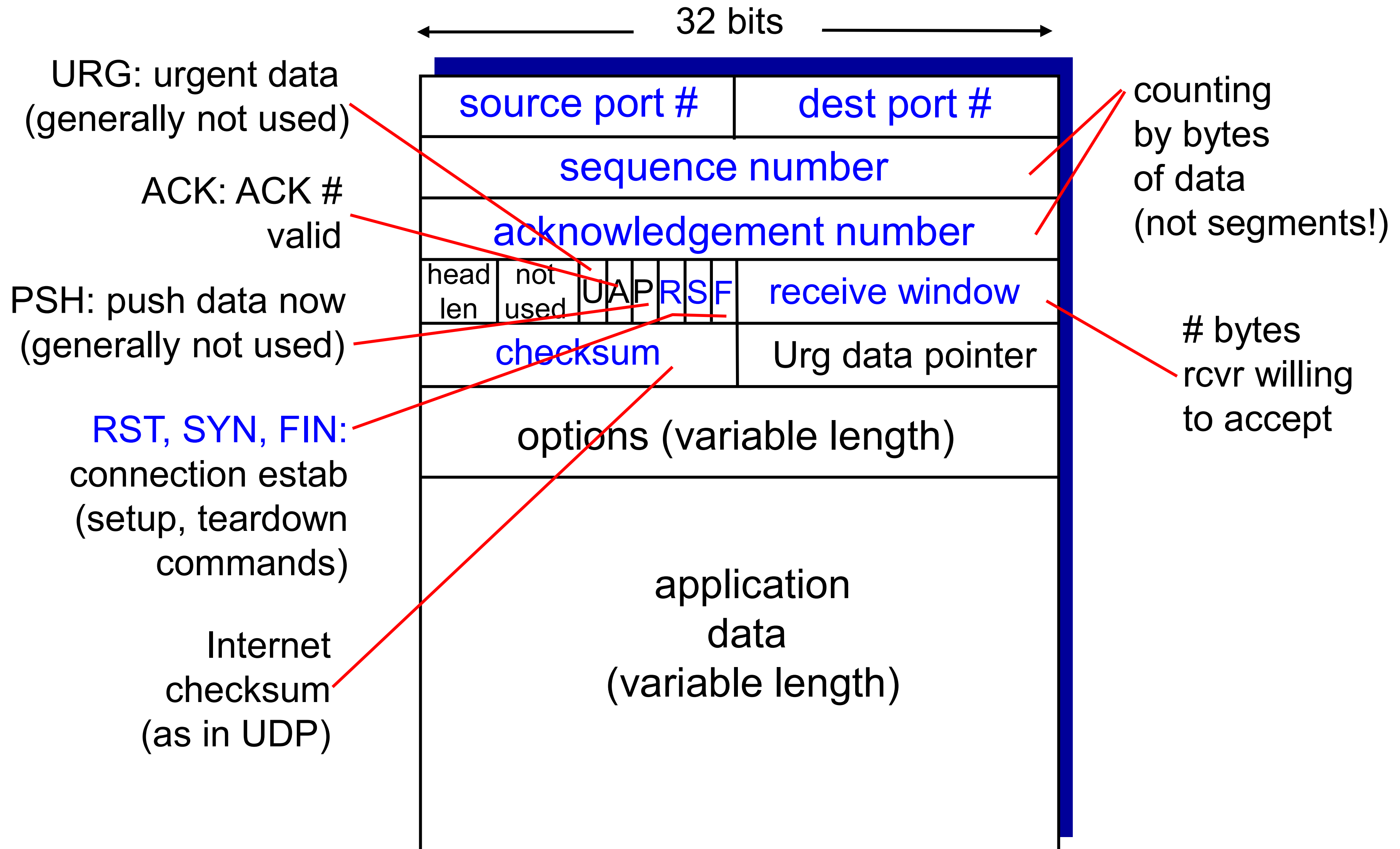
2.6 principles of congestion control

2.7 TCP congestion control

# 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



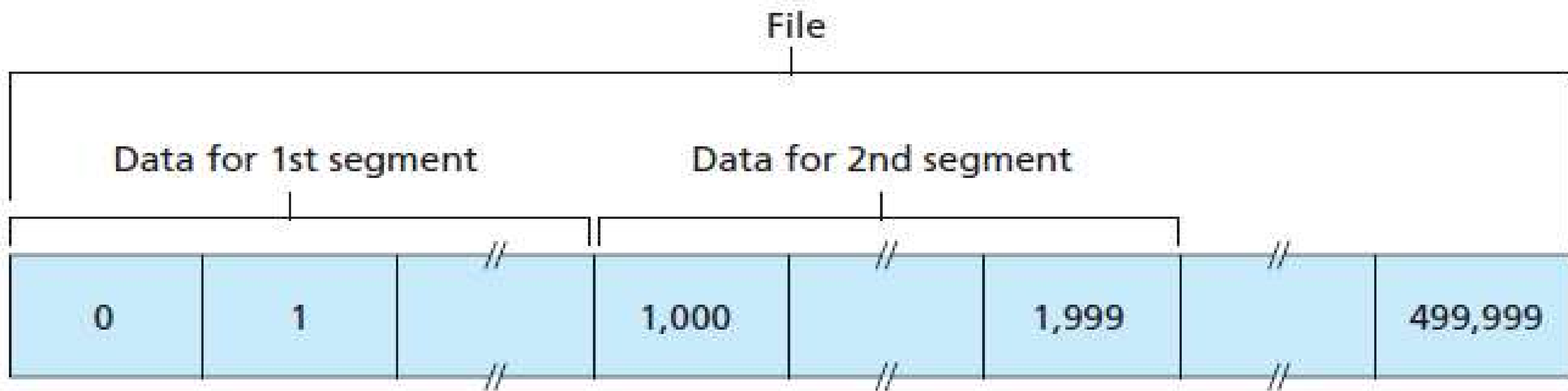
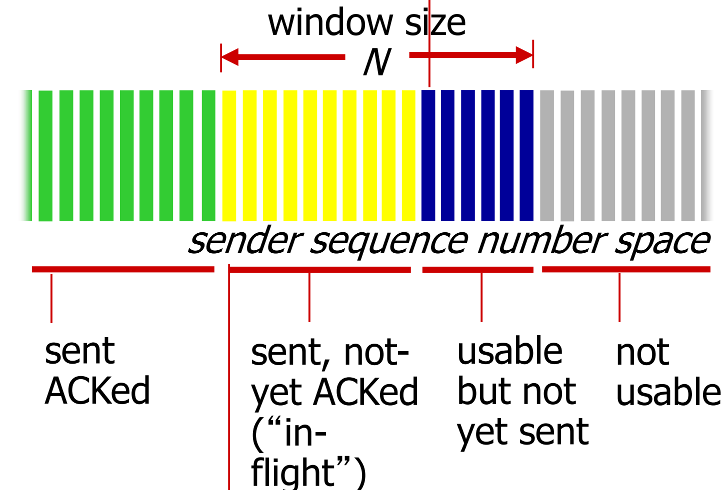


Figure 3.30 ♦ Dividing file data into TCP segments

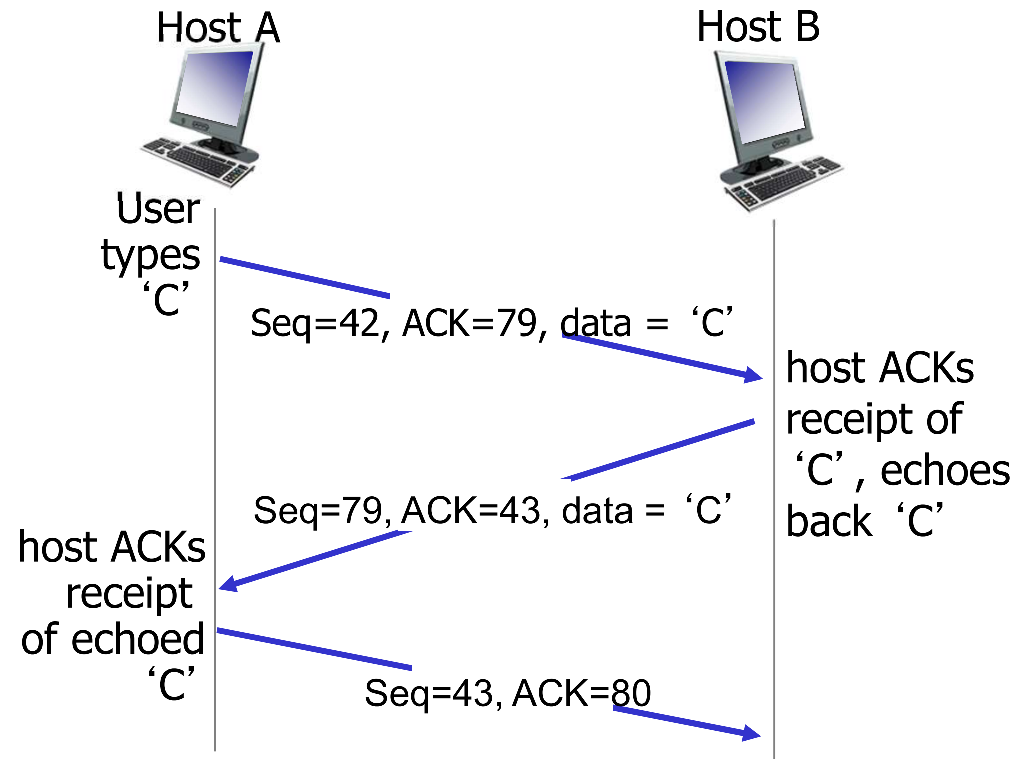
# TCP seq. numbers, ACKs

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

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



# TCP seq. numbers, ACKs




simple telnet scenario

# TCP reliable data transfer

IP does not

1. guarantee datagram delivery,
2. guarantee in-order delivery of datagrams,
3. guarantee the integrity of the data in the datagrams.

With IP service,

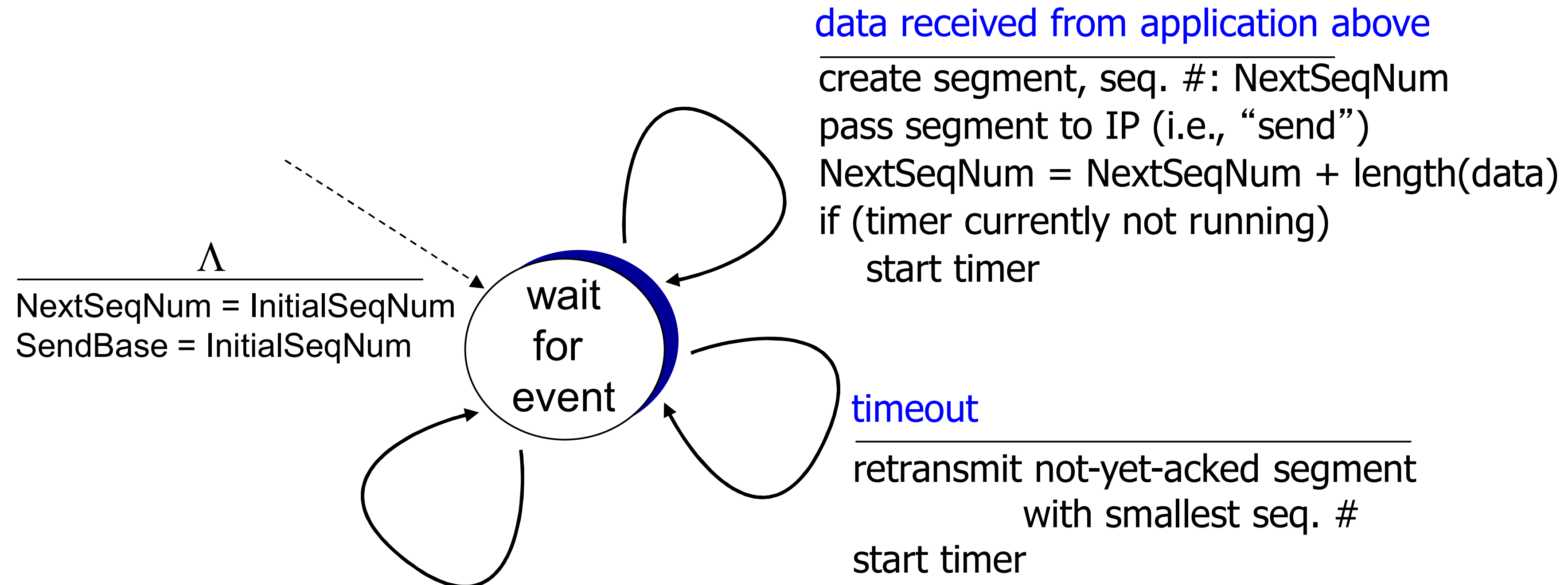
- datagrams can overflow router buffers and never reach their destination,
- It can arrive out of order
- bits in the datagram can get corrupted

# TCP sender events:

**Assumption:**

- sender is not constrained by
  - TCP flow control
  - congestion control
- data from above is less than MSS in size
- data transfer is in one direction only

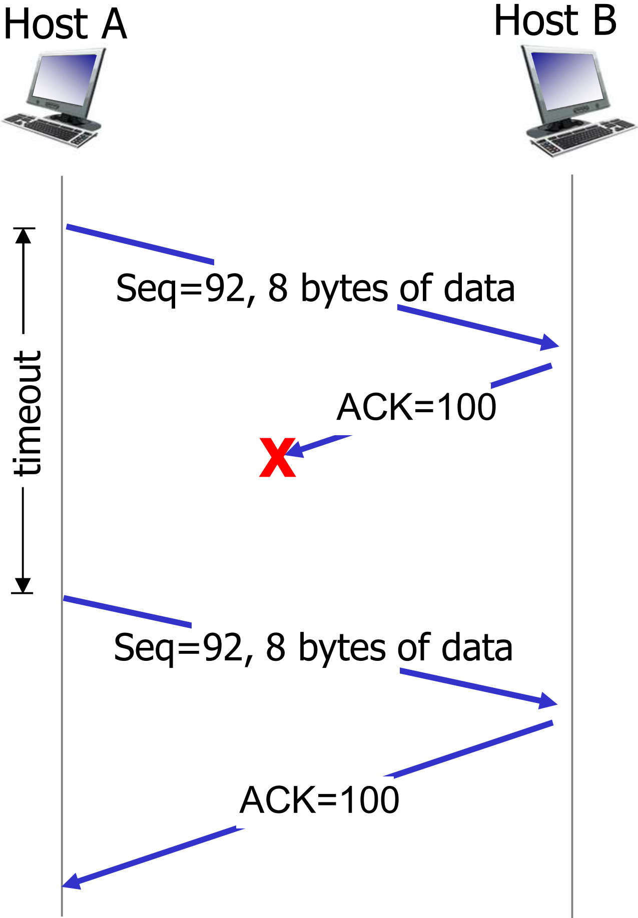
# TCP sender (simplified)



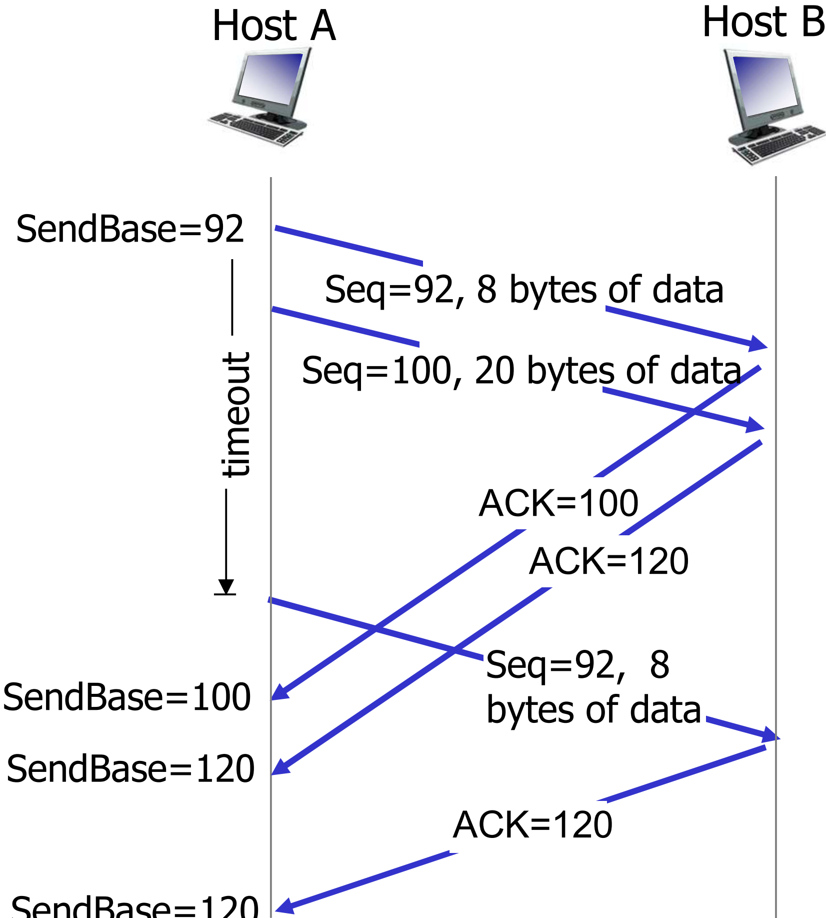

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
 }
```

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

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

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

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

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

# TCP: retransmission scenarios

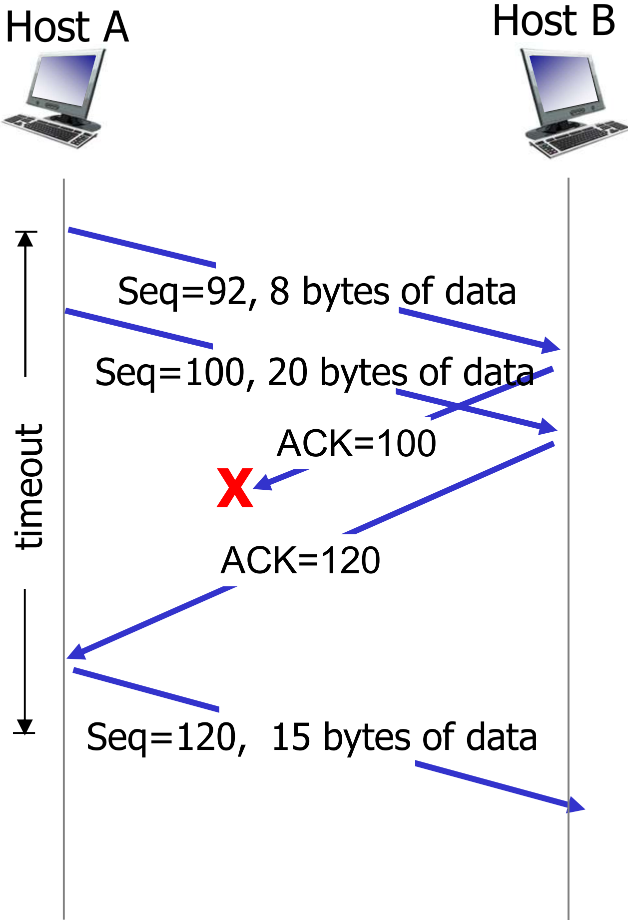



lost ACK scenario




premature timeout

# TCP: retransmission scenarios




cumulative ACK

# TCP fast retransmit

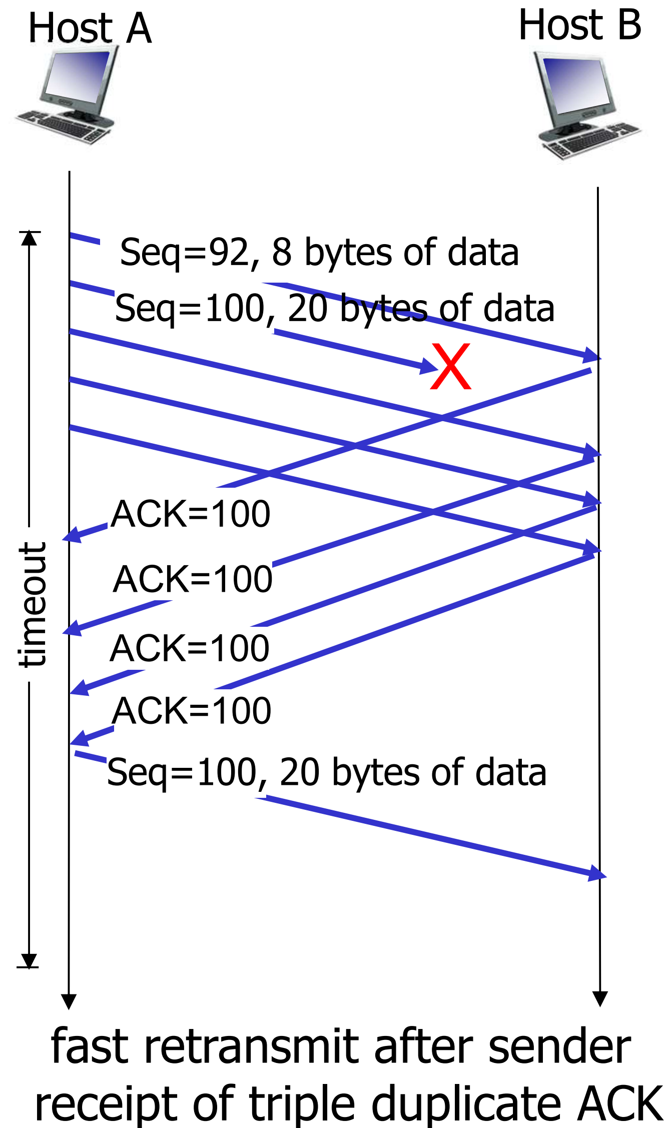
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

*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



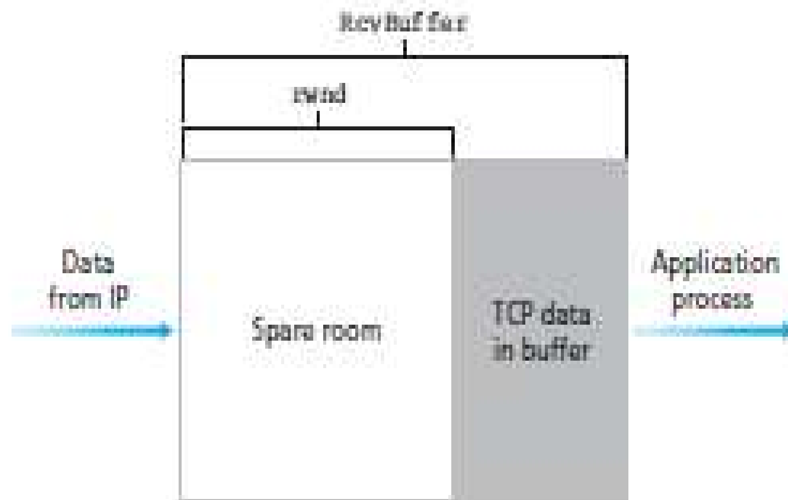

fast retransmit after sender receipt of triple duplicate ACK

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
                SendBase=y
                if (there are currently any not yet
                             acknowledged segments)
                   start timer
                }
          else { /* a duplicate ACK for already ACKed
                segment */
             increment number of duplicate ACKs
                received for y
             if (number of duplicate ACKS received
                for y==3)
                /* TCP fast retransmit */
                resend segment with sequence number y
             }
          break;
```

# TCP ACK generation

| *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 |

# Flow control




ure 3.38 • The receive window (rwnd) and the receive buffer (RcvBuffer)

- TCP provides a **flow-control service** to its applications to eliminate the possibility of the sender overflowing the receiver's buffer.
- Flow control is thus a speed-matching service—matching the rate at which the sender is sending against the rate at which the receiving application is reading.
- TCP provides flow control by having the *sender* maintain a variable called the **receive window**.

- the receive window is used to give the sender an idea of how much free buffer space is available at the receiver. Because TCP is full-duplex, the sender at each side of the connection maintains a distinct receive window.

- Host Ais sending a large file to Host B over a TCP connection. Host B allocates a receive buffer to this connection; denote its size by RcvBuffer.

- From time to time, the application process in Host B reads from the buffer. Define the following variables:
  - LastByteRead: the number of the last byte in the data stream read from the buffer by the application process in B
  - LastByteRcvd: the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B

Because TCP is not permitted to overflow the allocated buffer, we must have

LastByteRcvd – LastByteRead ≤ RcvBuffer

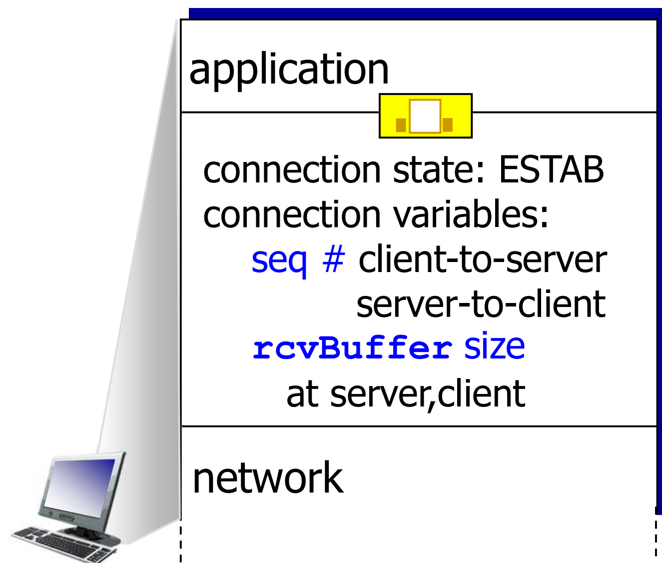
The receive window, denoted rwnd is set to the amount of spare room in the buffer:

rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

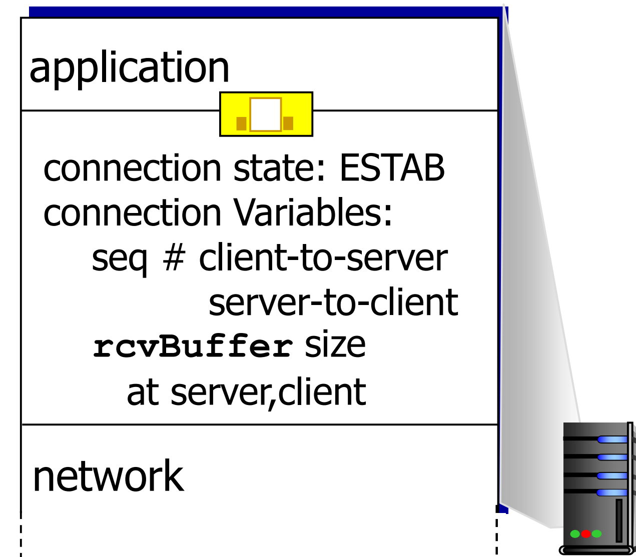
# Connection Management

before exchanging data, sender/receiver "handshake":

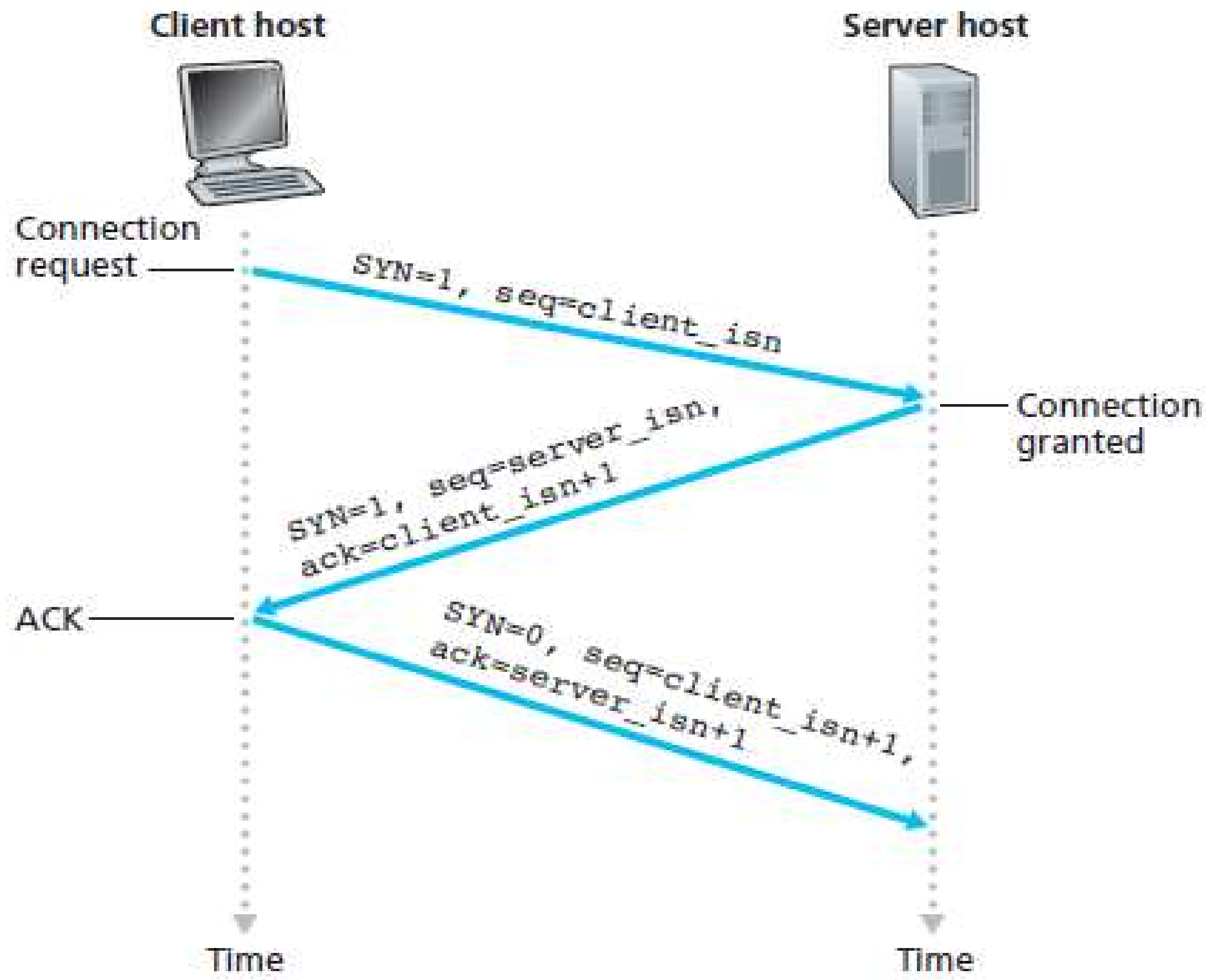
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters




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

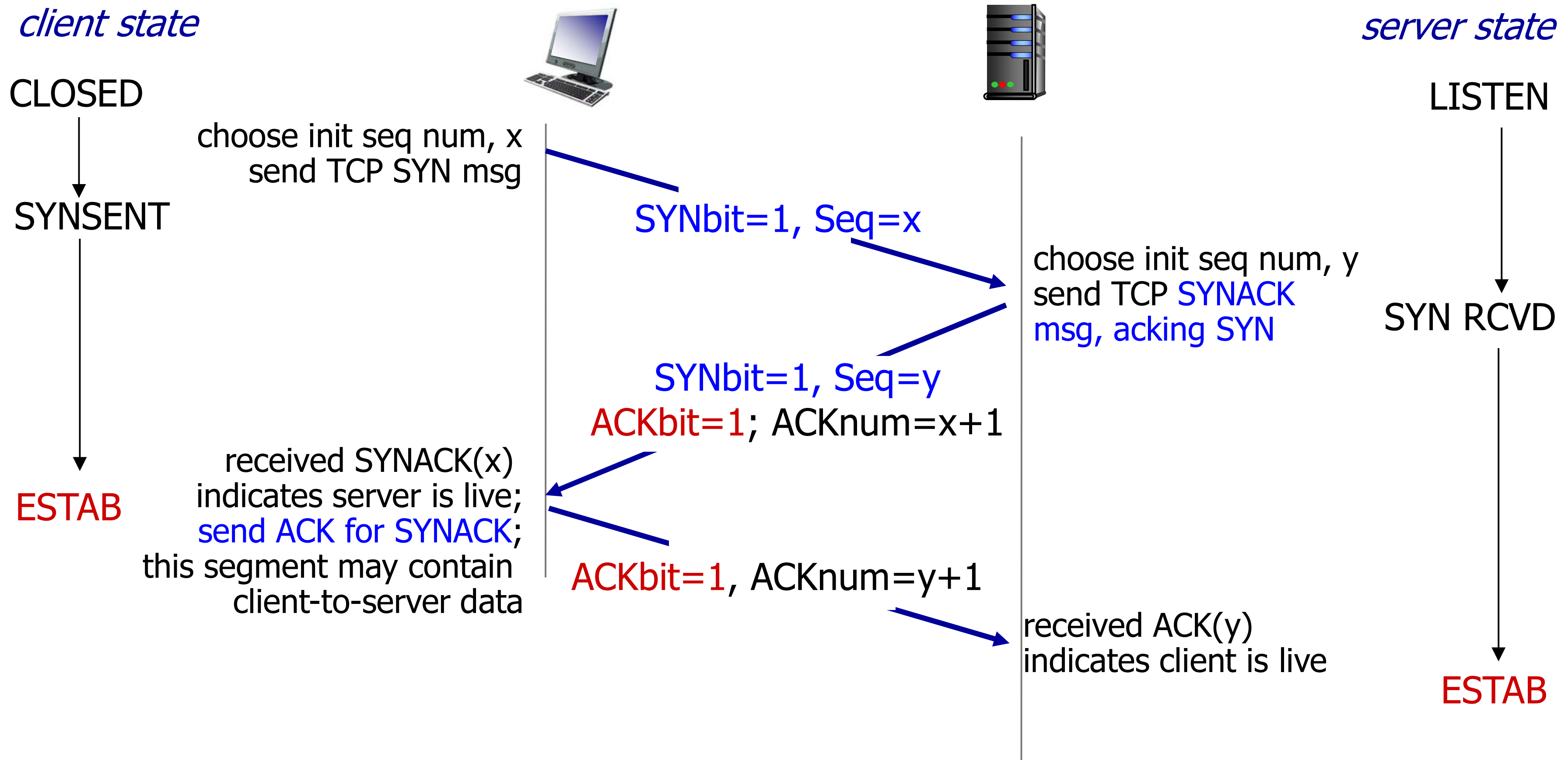



```
Socket connectionSocket =
  welcomeSocket.accept();
```

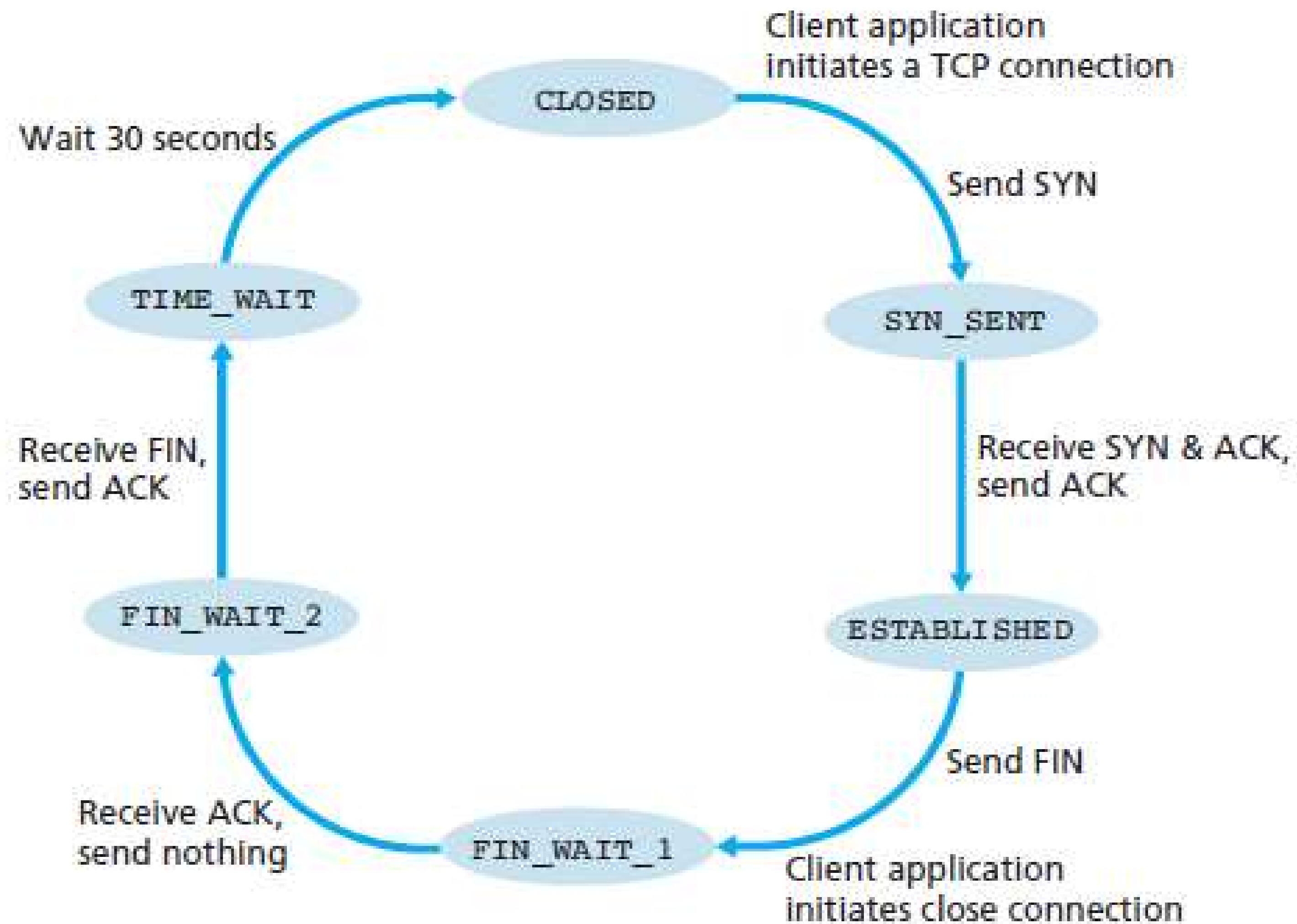


Client host
Server host
Connection
request
SYN=1, seq=client_isn
Connection
granted
SYN=1, seq=server_isn,
ack=client_isn+1
ACK
SYN=0, seq=client_isn+1,
ack=server_isn+1
Time
Time

# TCP 3-way handshake



# TCP 3-way handshake: FSM



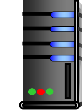
# TCP: closing a connection

- 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

*client state*

*server state*



ESTAB

ESTAB

```
clientSocket.close()
```

FIN_WAIT_1 — can no longer send but can receive data

CLOSE_WAIT

FIN_WAIT_2 — wait for server close

can still send data

LAST_ACK

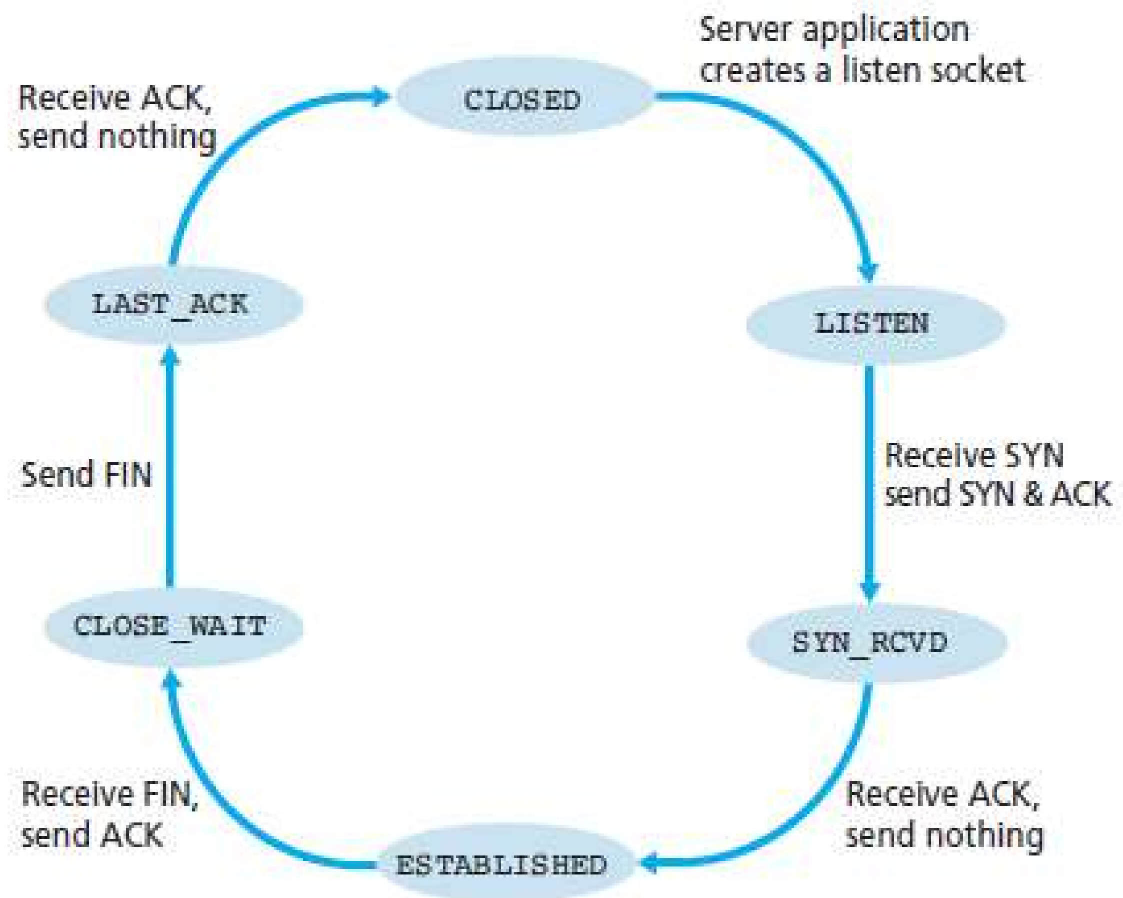
TIMED_WAIT

can no longer send data

timed wait for 2*max segment lifetime

CLOSED

CLOSED


Server application
creates a listen socket
Receive ACK,
send nothing
CLOSED
LAST_ACK
LISTEN
Send FIN
Receive SYN
send SYN & ACK
CLOSE_WAIT
SYN_RCVD
Receive FIN,
send ACK
ESTABLISHED
Receive ACK,
send nothing

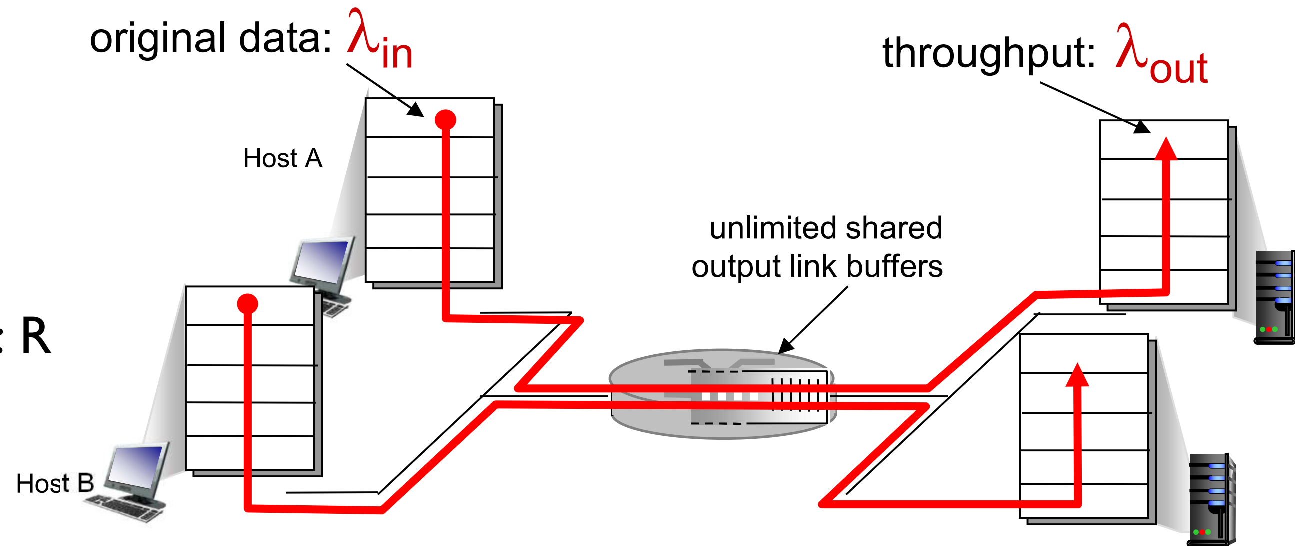
# Principles of congestion control

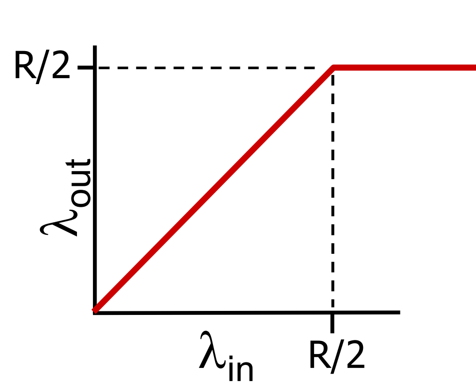
*congestion:*

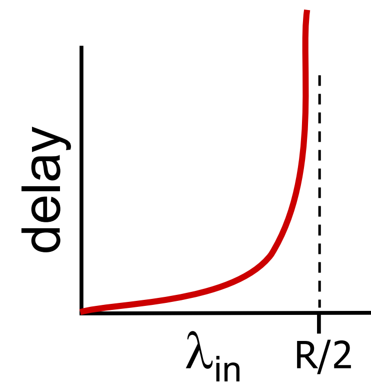
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

# Causes/costs of congestion: scenario 1

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



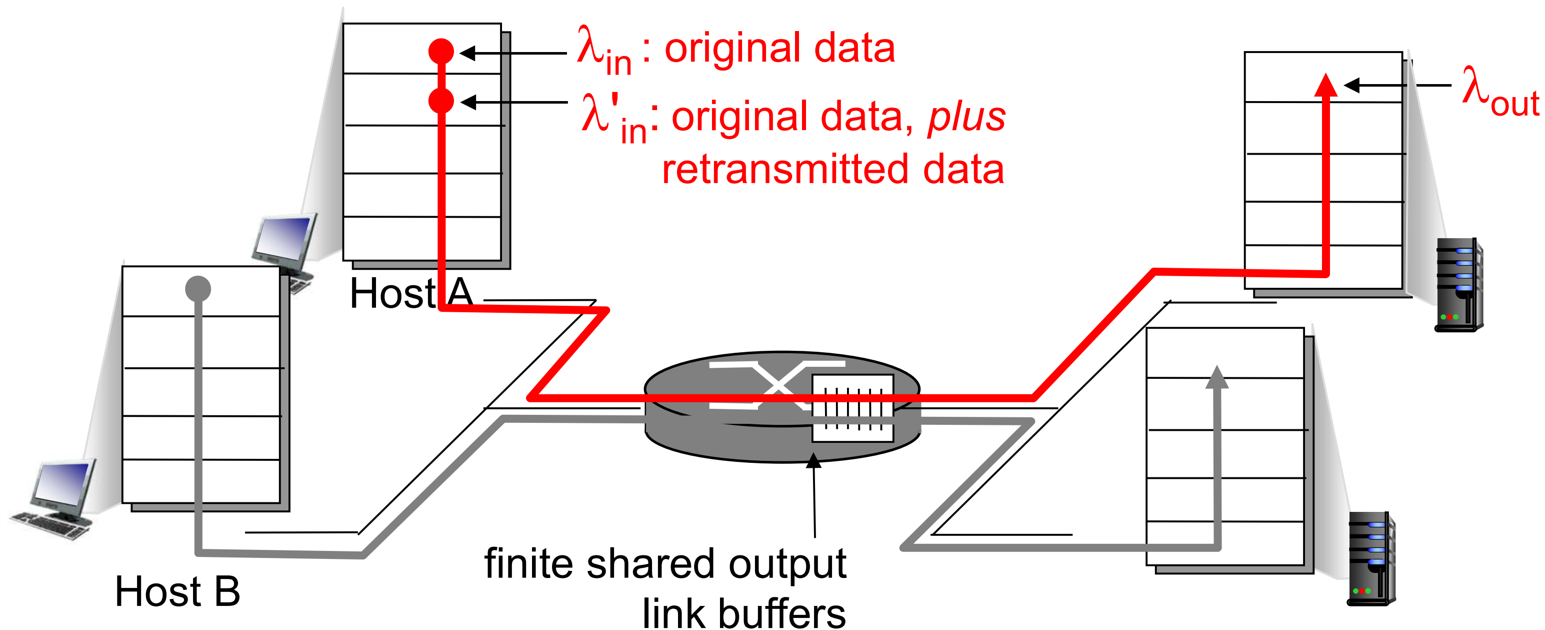




- maximum per-connection throughput: R/2
- large delays as arrival rate, $\lambda_{in}$, approaches capacity

# Causes/costs of congestion: scenario 2

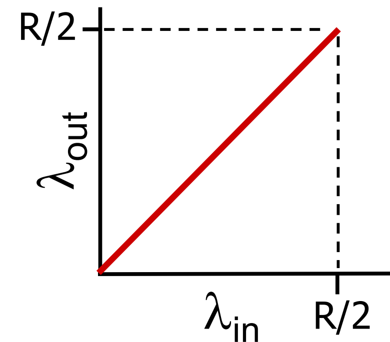
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

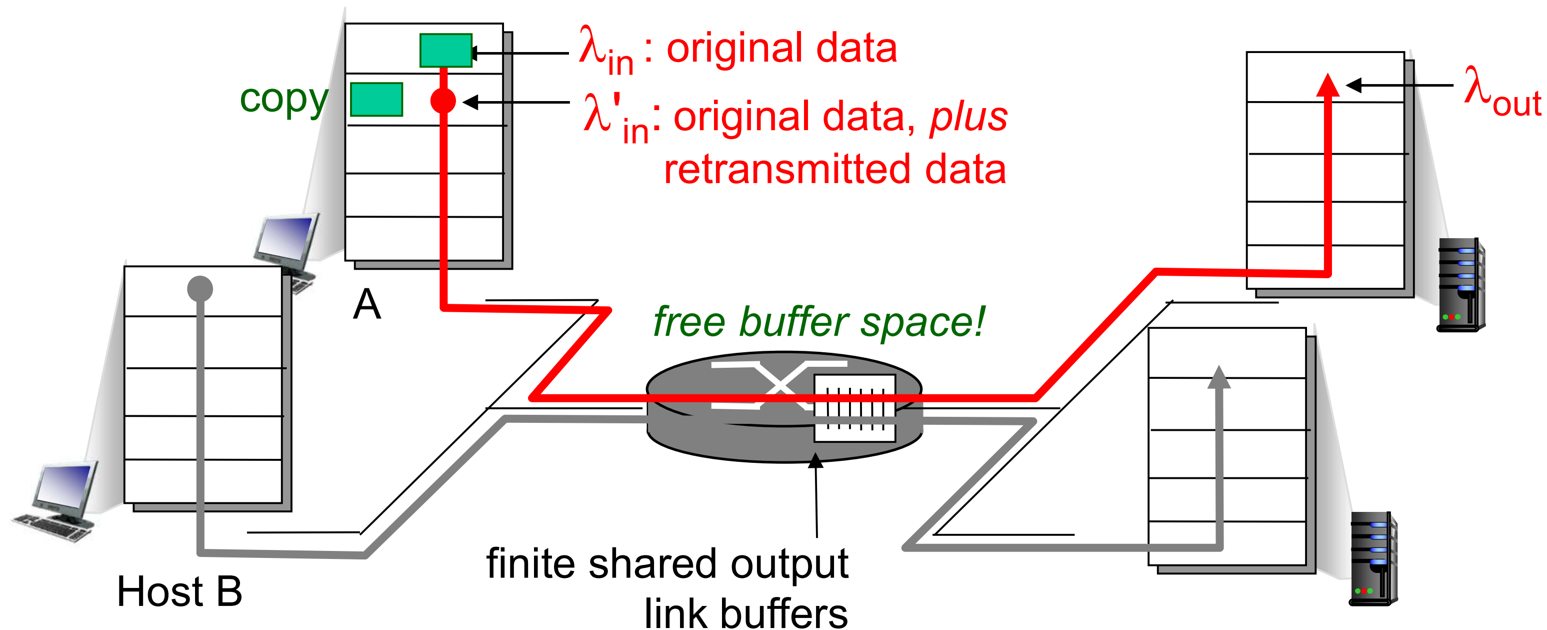


# Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

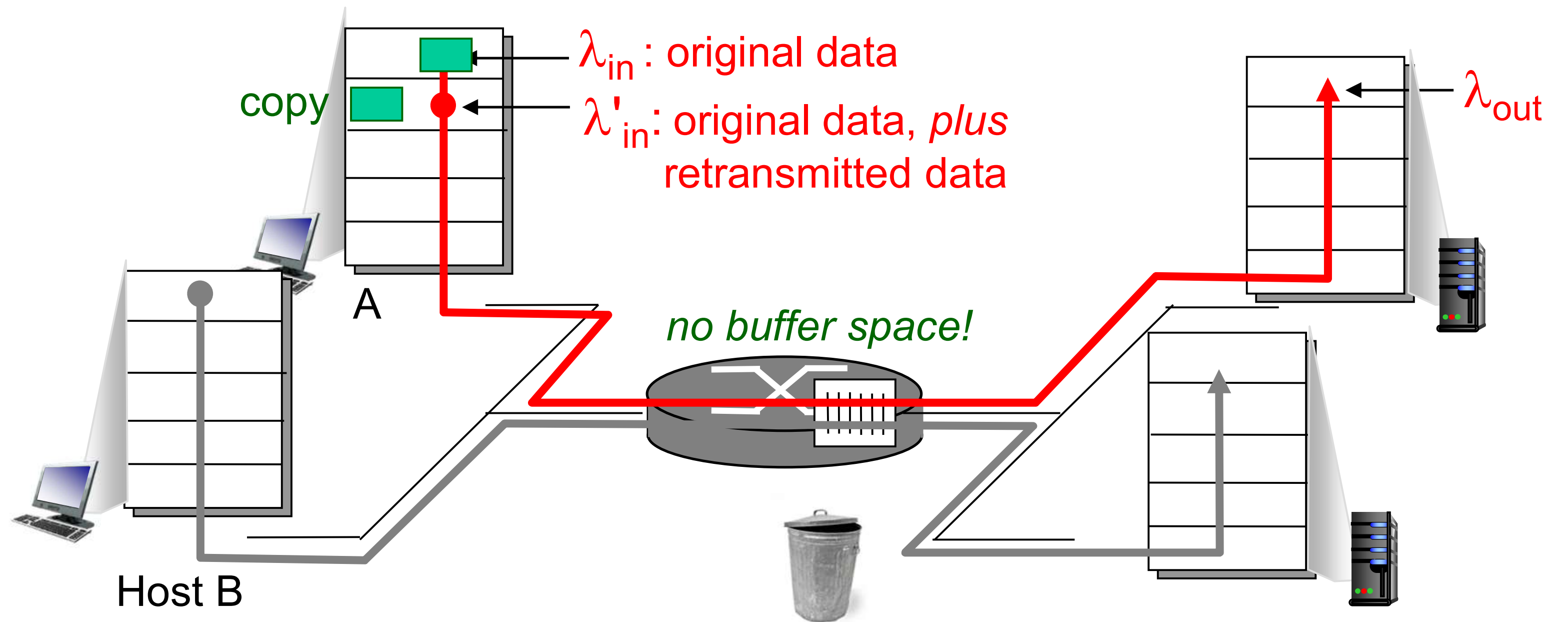


# Causes/costs of congestion: scenario 2

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

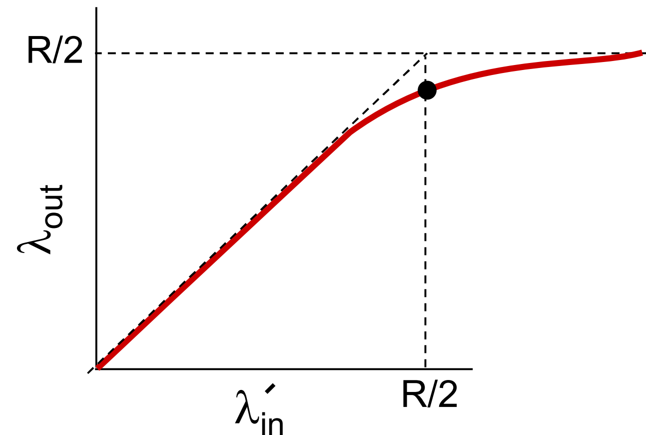
- sender only resends if packet *known* to be lost

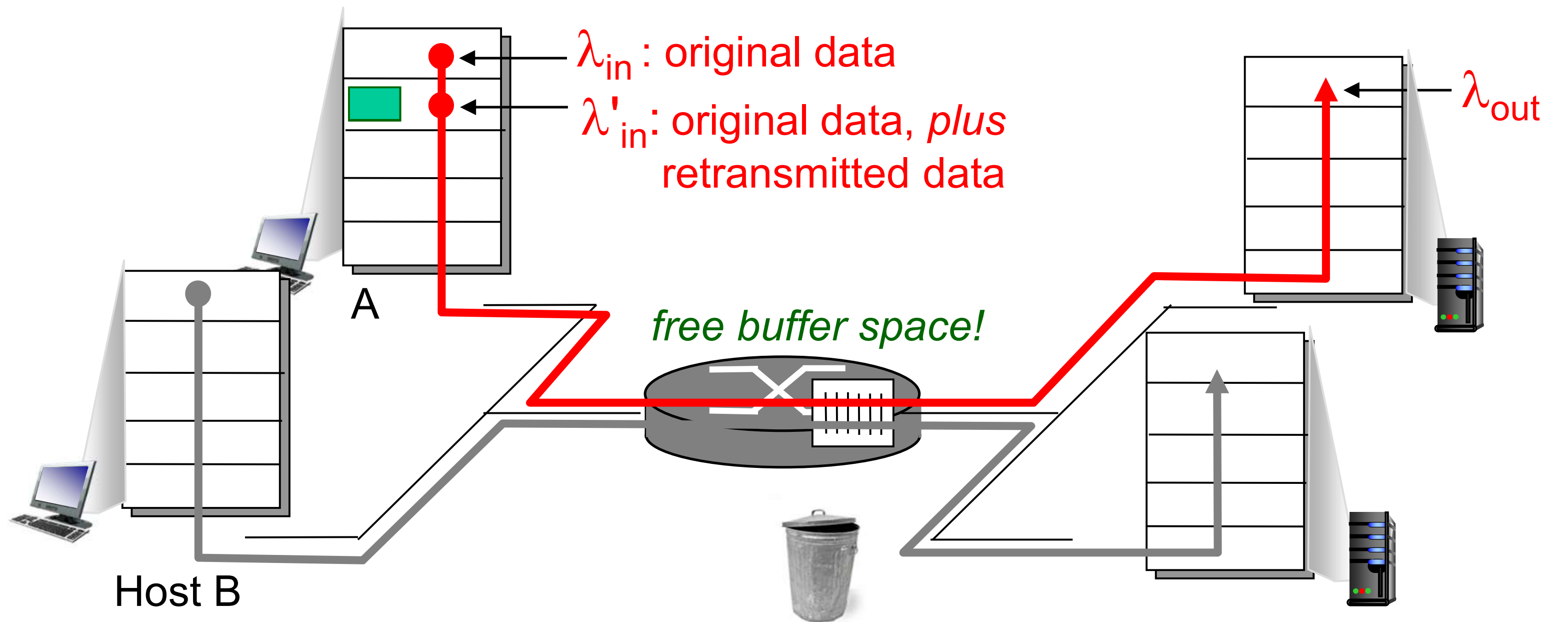


# Causes/costs of congestion: scenario 2

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

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

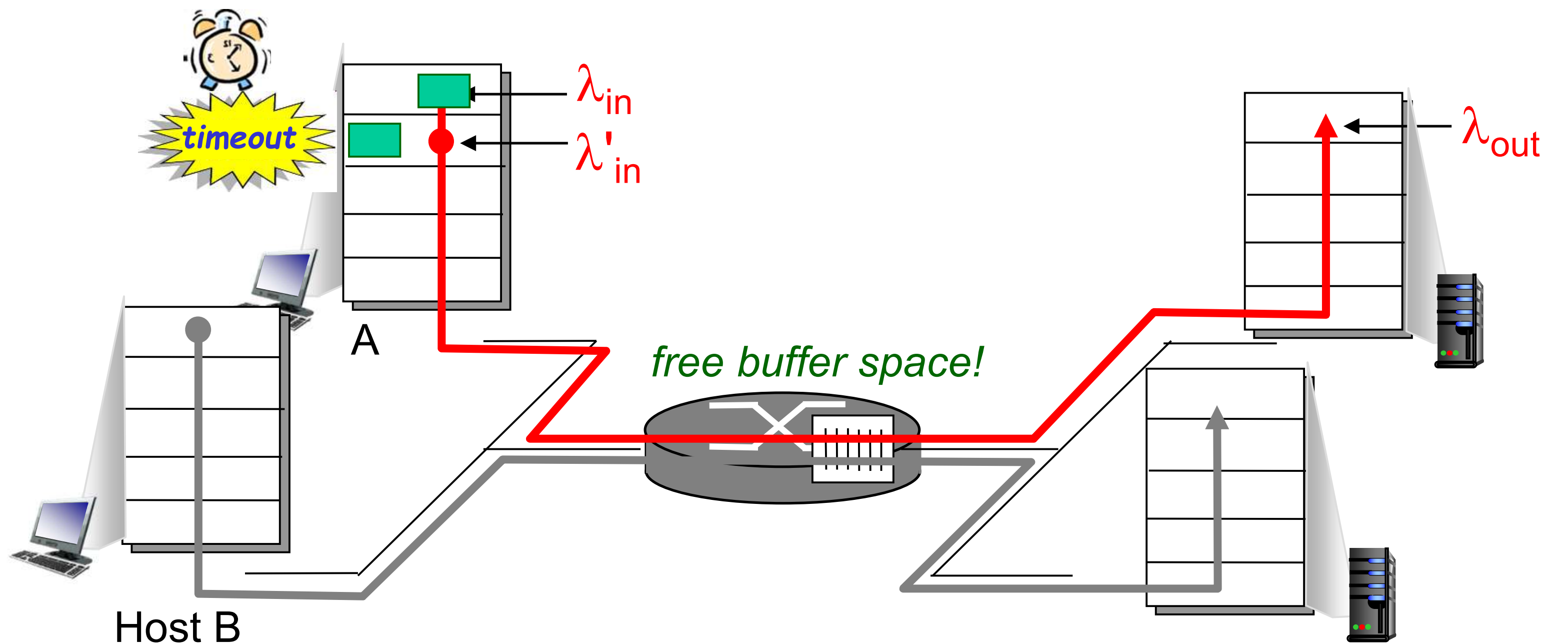


# Causes/costs of congestion: scenario 2

## *Realistic: duplicates*

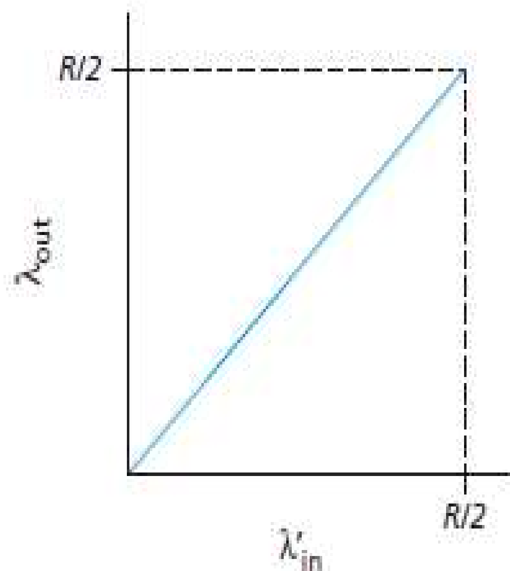
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

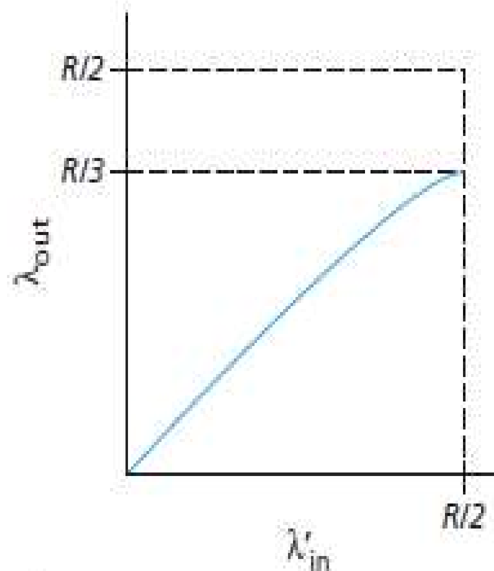


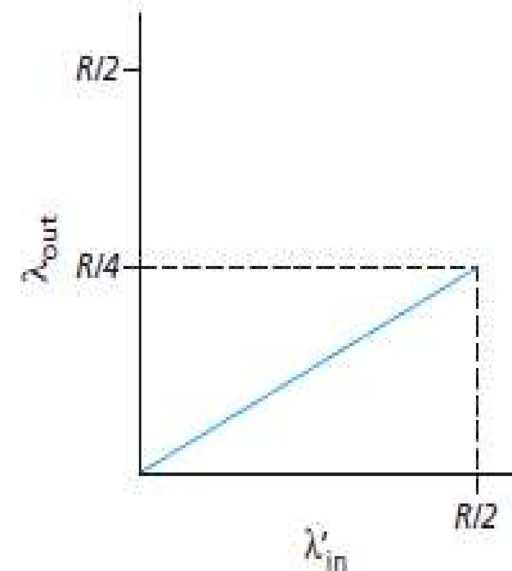
# Causes/costs of congestion: scenario 2

"costs" of congestion:

- more work (retransmit)
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing good throughput

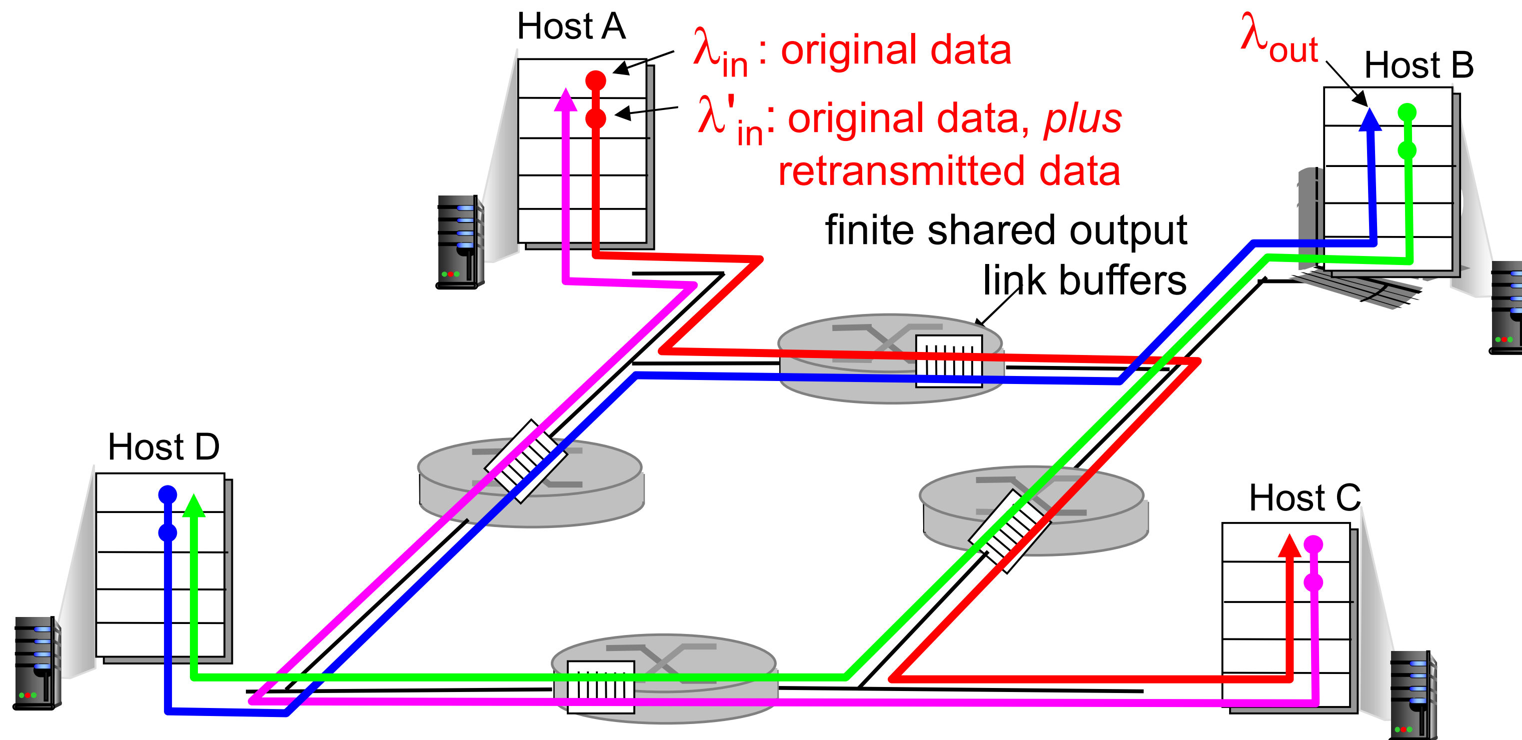


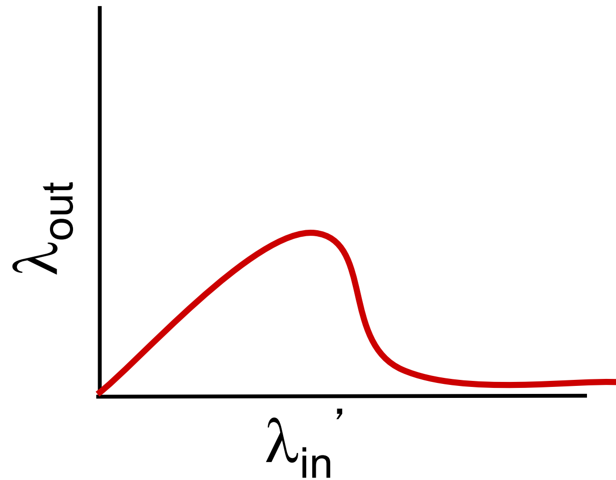



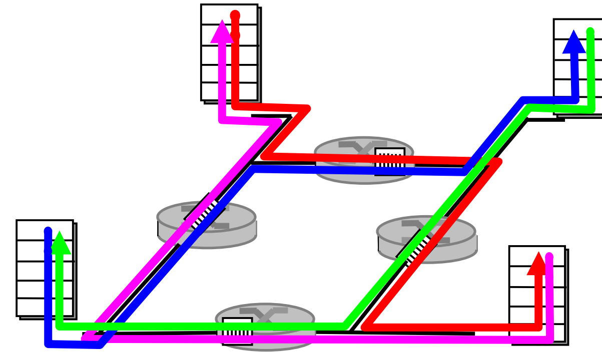
# Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit



# 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/switches provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Explicit feedback

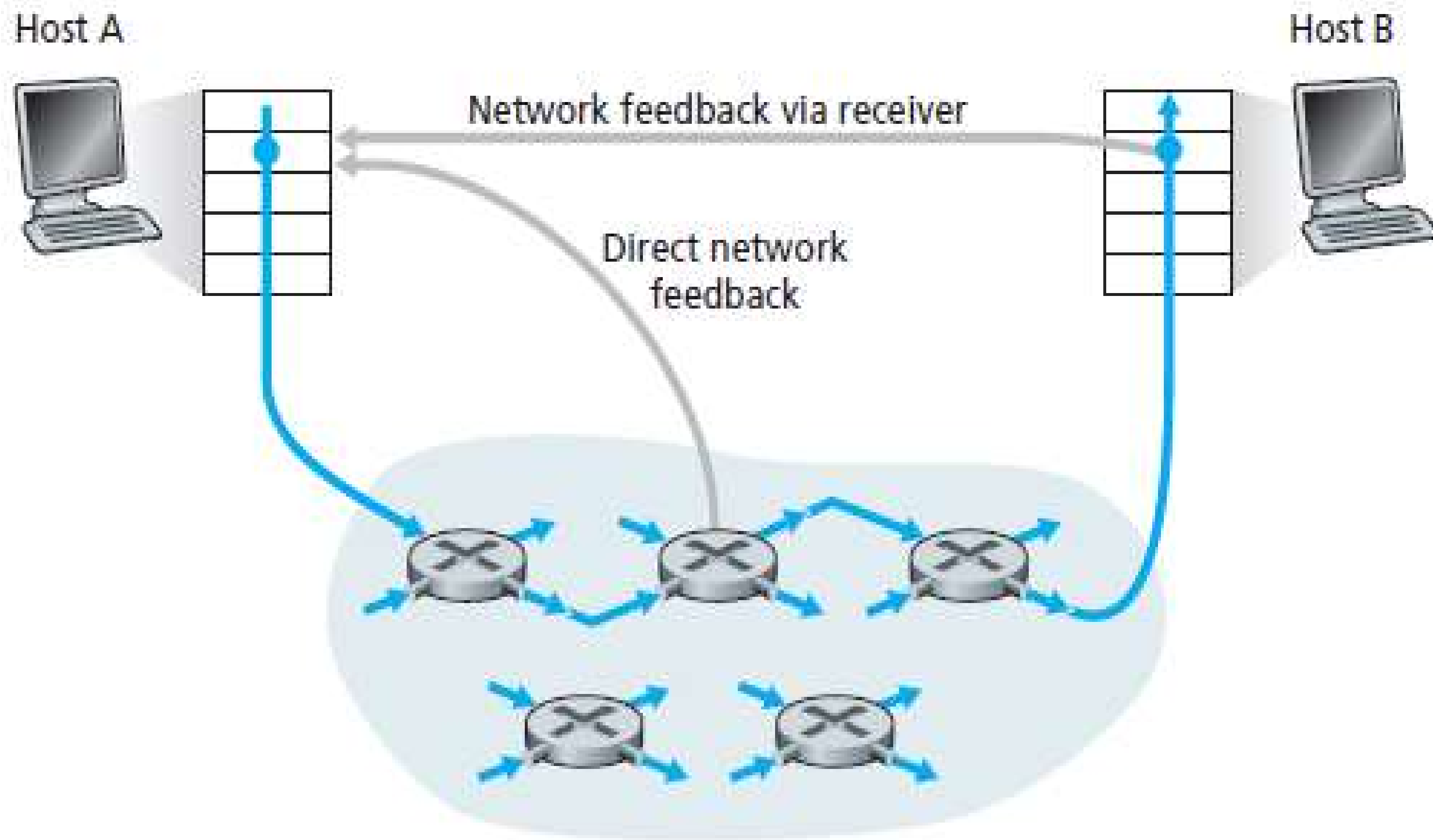



**Figure 3.49** ♦ Two feedback pathways for network-indicated congestion information

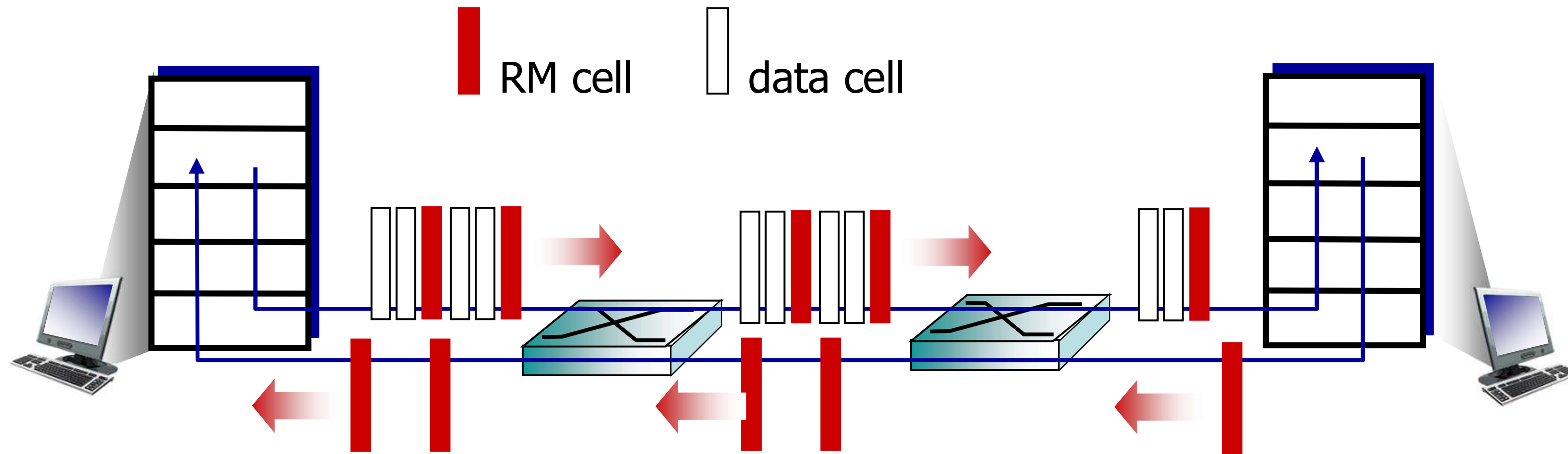
# Case study: ATM ABR congestion control

ABR: available bit rate:
in ATM

RM (resource management) cells:

- sent by sender, interspersed with data cells ( RM cell)
- 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,

# Case study: ATM ABR congestion control

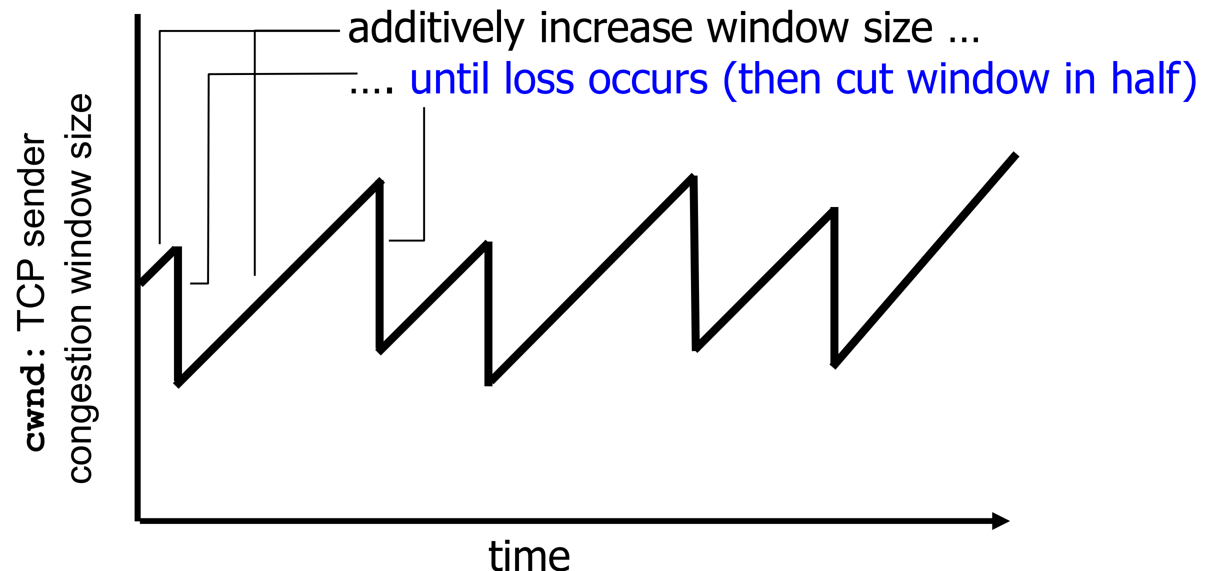



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - Set to min supportable rate of all switches on source to destination path
- EFCI (Explicit Forward CI) bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

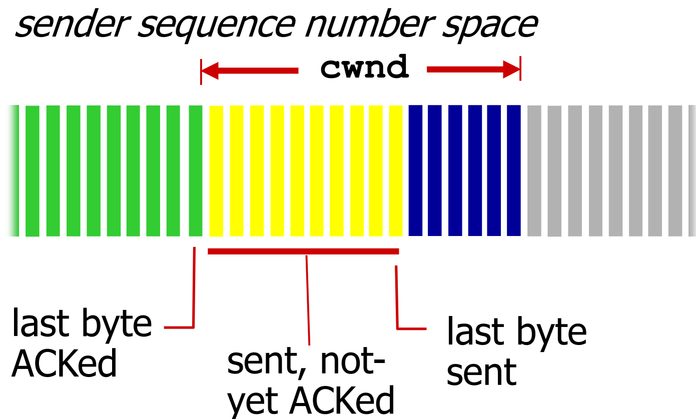
# TCP congestion control: additive increase multiplicative decrease

- *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:

```
LastByteSent-
  LastByteAcked   ≤  cwnd
```

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

*TCP sending rate:*

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

## congestion window

- In TCP, the **congestion window** is one of the factors that determines the number of bytes that can be outstanding at any time.
- The congestion window is maintained by the sender.
- The congestion window is a means of stopping a link between the sender and the receiver from becoming overloaded with too much traffic.
- It is calculated by estimating how much congestion there is on the link.

1. A lost segment implies congestion, and hence, the TCP sender’s rate should be decreased when a segment is lost.

2. An acknowledged segment indicates that the network is delivering the sender’s segments to the receiver, and hence, the sender’s rate can be increased when an ACK arrives for a previously unacknowledged segment.

3. Bandwidth probing

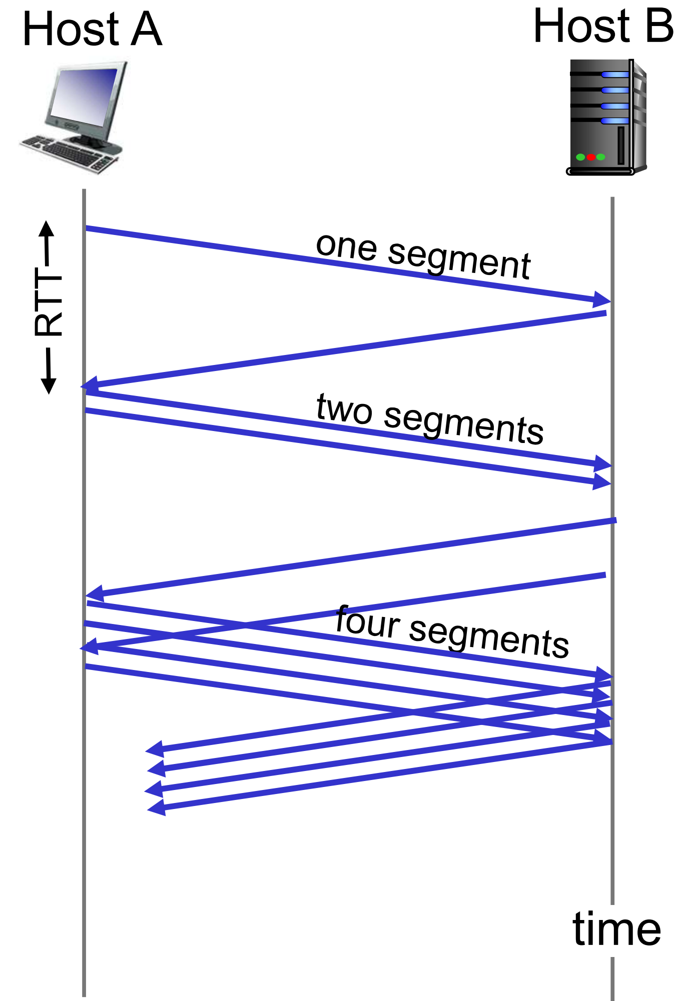
**TCP congestion-control algorithm**

The algorithm has three major components:

(1) slow start,
(2) congestion avoidance, and
(3) fast recovery.

# 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

1. if there is a loss event (i.e., congestion) indicated by a timeout, the TCP sender sets the value of cwnd to 1 and begins the slow start process anew.

   -It also sets the value of a second state variable, ssthresh (shorthand for "slow start threshold") to cwnd/2

2. slow start may end is directly tied to the value of ssthresh.

   -when the value of cwnd equals ssthresh, slow start ends and TCP transitions into congestion avoidance mode.

3. slow start can end is if three duplicate ACKs are detected, in which case TCP performs a fast retransmit and enters the fast recovery state,

# Fast Recovery

- **TCP Vegas**
- TCP Vegas was deployed as the default congestion control method.
- that emphasizes packet delay, rather than packet loss, as a signal to help determine the rate at which to send packets.
- TCP Vegas uses additive increases in the congestion window.

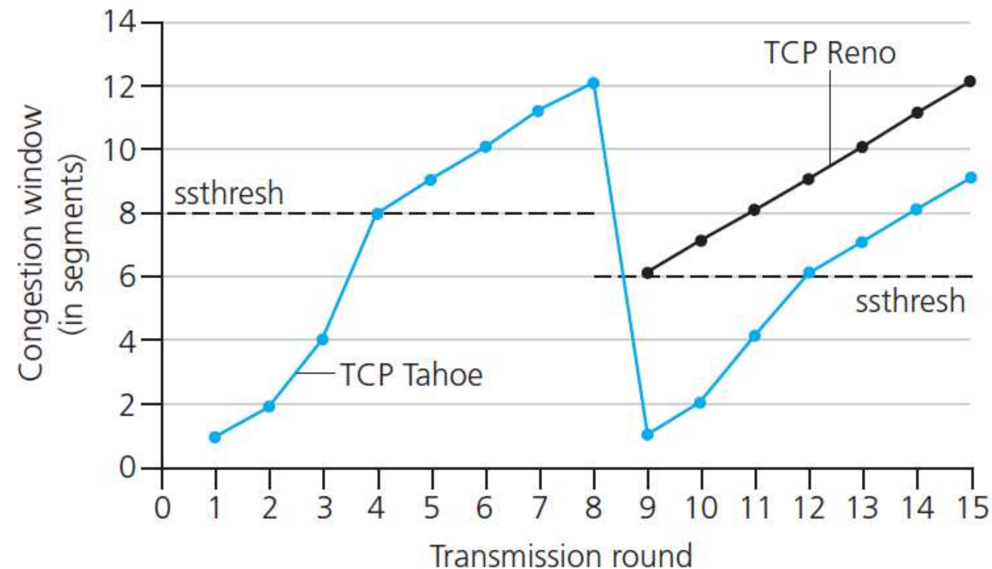
- **TCP Tahoe**
- When a loss occurs, fast retransmit is sent
- half of the current CWND is saved as *ssthresh*
- slow start begins again from its initial CWND.
- Once the CWND reaches *ssthresh*,
- TCP changes to congestion avoidance algorithm where *each new ACK* increases the CWND by *MSS / CWND*.
- This results in a linear increase of the CWND.

- **TCP Reno**
- A fast retransmit is sent,
- half of the current CWND is saved as *ssthresh*and as new CWND,
- thus skipping slow start and going directly to the congestion avoidance algorithm.
- The overall algorithm here is called fast recovery.

# TCP: switching from slow start to CA

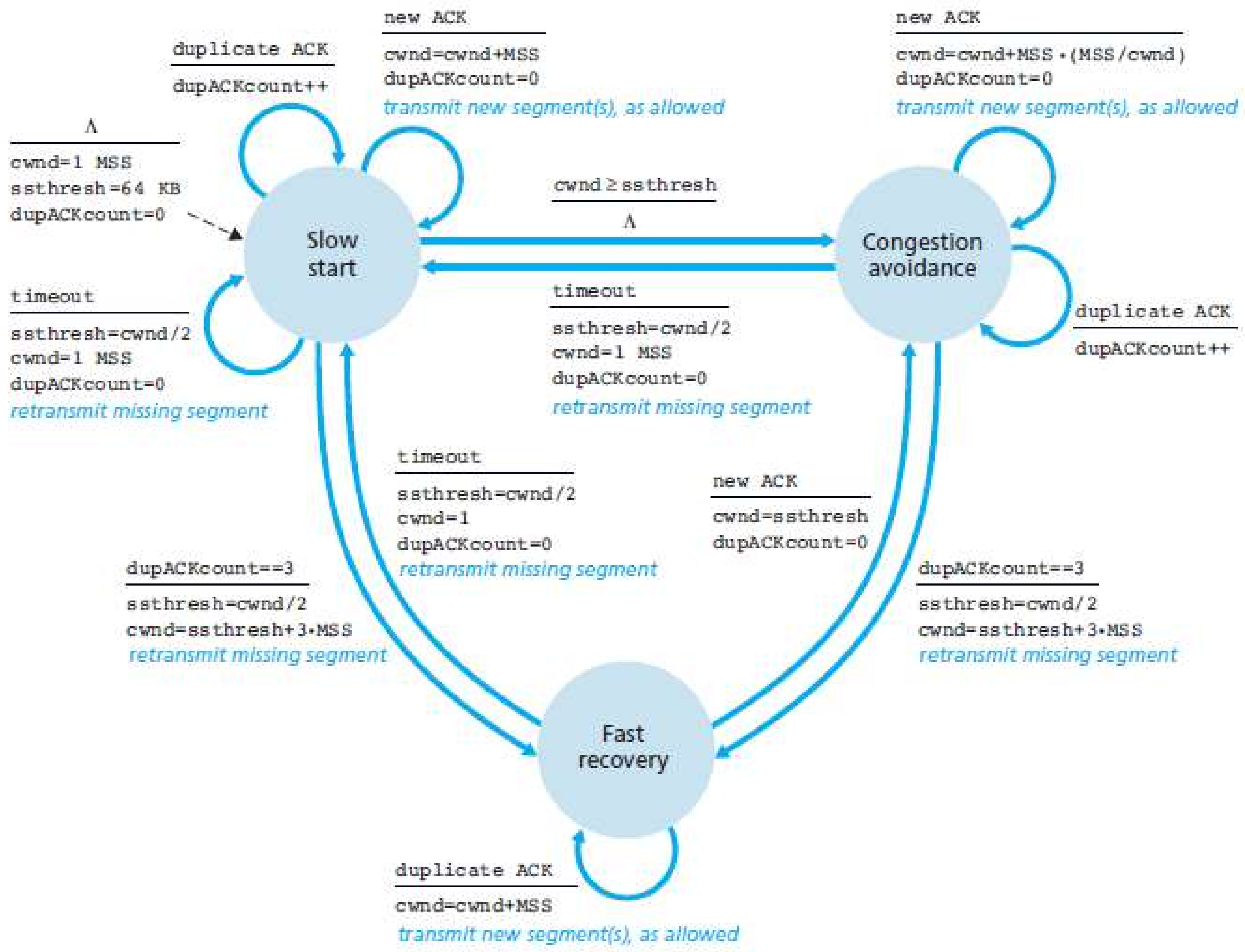
Q: when should the exponential increase switch to linear?

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

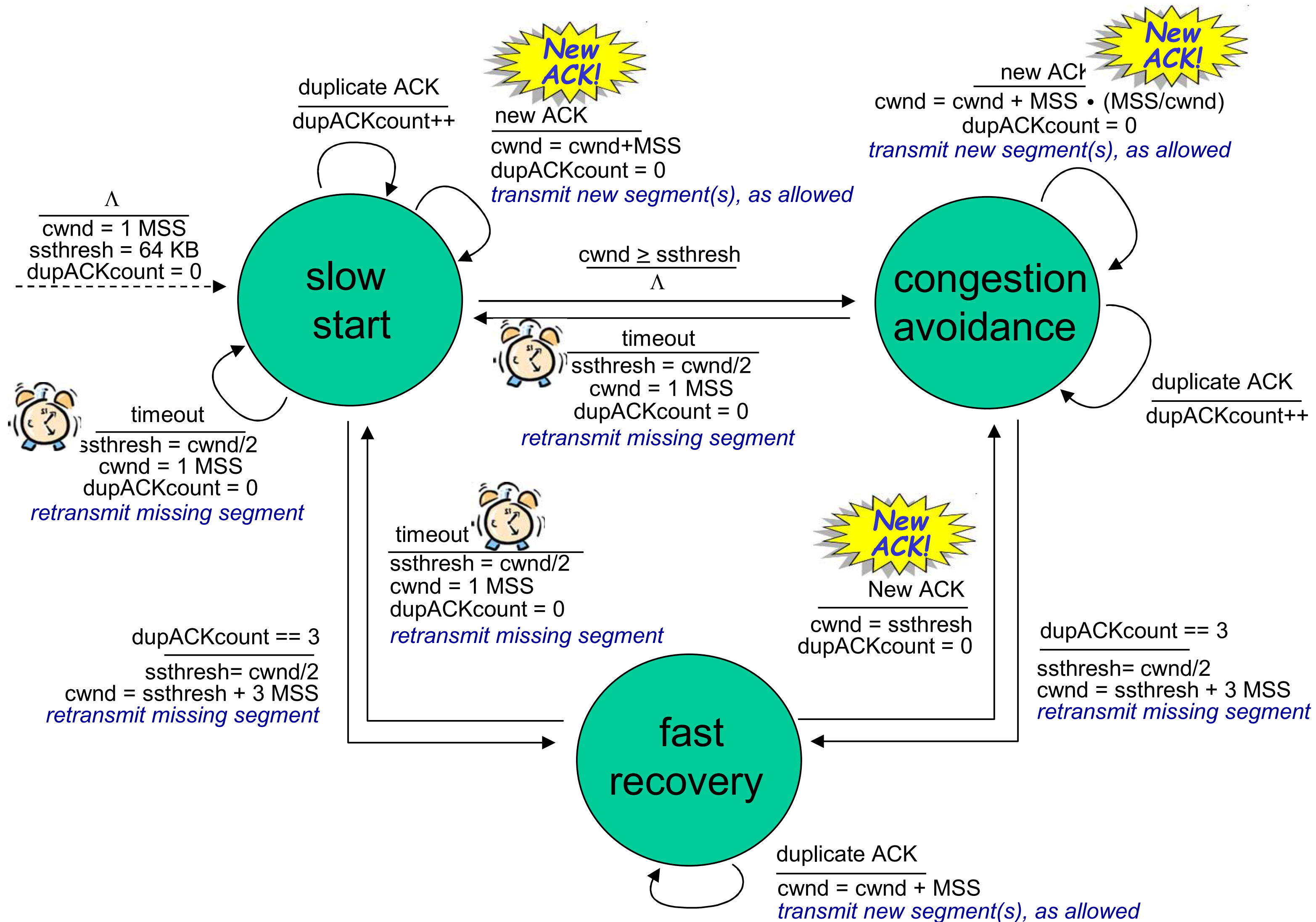



## Implementation:

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


duplicate ACK
dupACKcount++
new ACK
cwnd=cwnd+MSS
dupACKcount=0
transmit new segment(s), as allowed
new ACK
cwnd=cwnd+MSS•(MSS/cwnd)
dupACKcount=0
transmit new segment(s), as allowed
Λ
cwnd=1 MSS
ssthresh=64 KB
dupACKcount=0
Slow start
cwnd ≥ ssthresh
Λ
Congestion avoidance
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
duplicate ACK
dupACKcount++
timeout
ssthresh=cwnd/2
cwnd=1
dupACKcount=0
retransmit missing segment
new ACK
cwnd=ssthresh
dupACKcount=0
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
Fast recovery
duplicate ACK
cwnd=cwnd+MSS
transmit new segment(s), as allowed

# Summary: TCP Congestion Control



xgraph

Close Hdcpy About

Congestion Window graph for Reno

congestion window size

16cs088

23.0000
22.0000
21.0000
20.0000
19.0000
18.0000
17.0000
16.0000
15.0000
14.0000
13.0000
12.0000
11.0000
10.0000
9.0000
8.0000
7.0000
6.0000
5.0000
4.0000
3.0000
2.0000
1.0000
0.0000

0.0000 1.0000 2.0000 3.0000 4.0000 5.0000

Time(sec)

xgraph

Close Hdcpy About

Congestion Window graph for Reno

congestion window size

16cs088

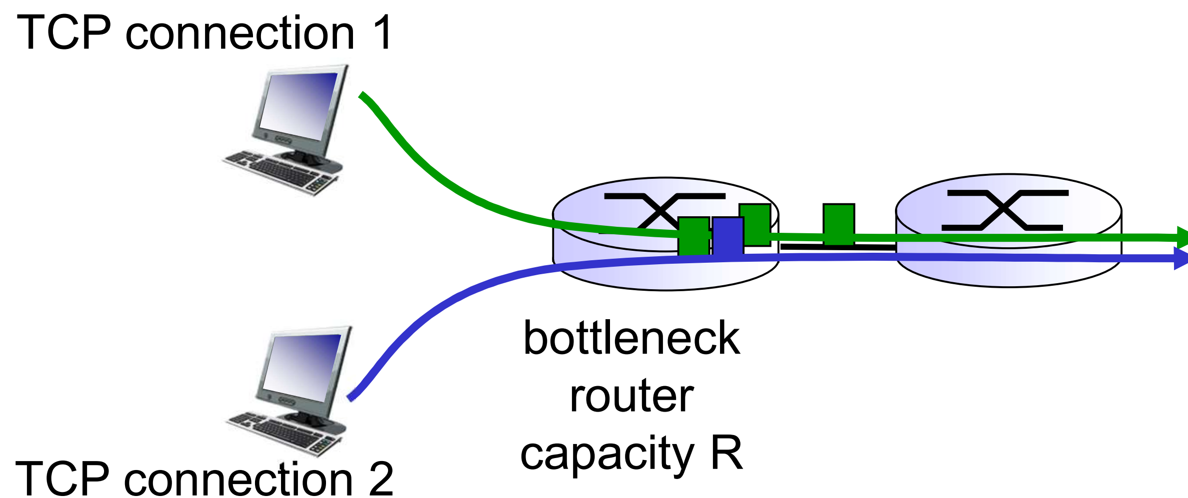
32.0000
30.0000
28.0000
26.0000
24.0000
22.0000
20.0000
18.0000
16.0000
14.0000
12.0000
10.0000
8.0000
6.0000
4.0000
2.0000
0.0000

0.0000 1.0000 2.0000 3.0000 4.0000 5.0000

Time(sec

# TCP Fairness

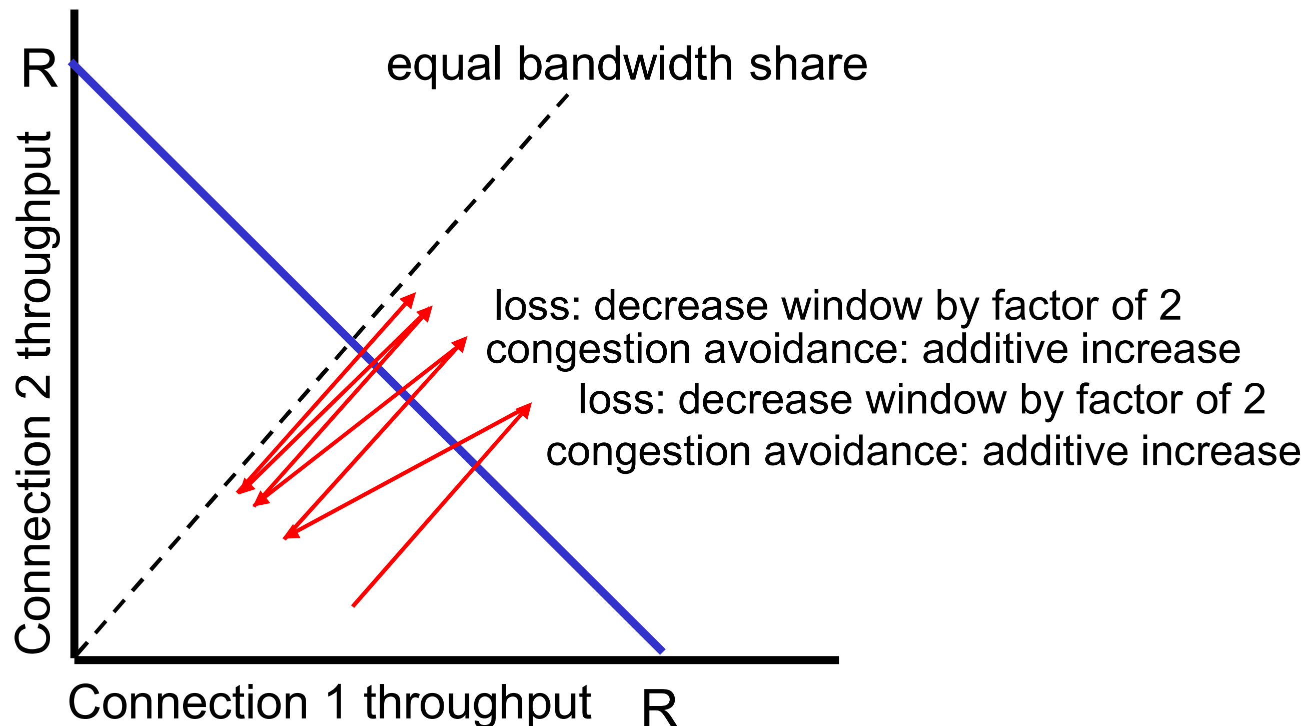
*fairness goal:* if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



# 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

# Chapter 2: summary

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

next:

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