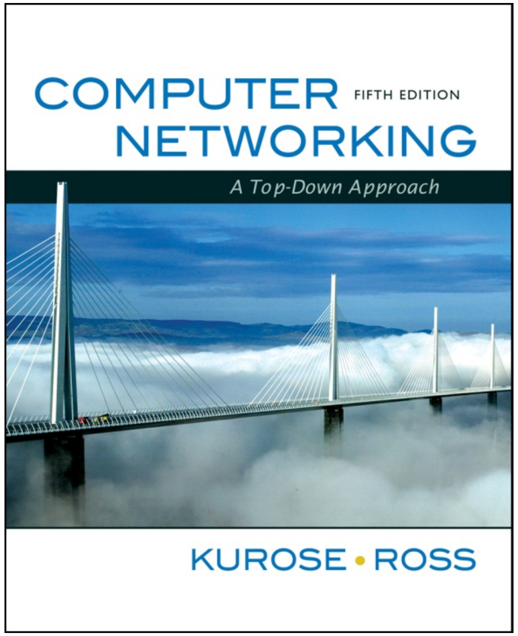
# Chapter 3
# Transport Layer




## A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR



*Computer Networking: A Top Down Approach ,*
6th edition.
Jim Kurose, Keith Ross
Addison-Wesley, March 2012.

# Chapter 3: Transport Layer

Our goals:

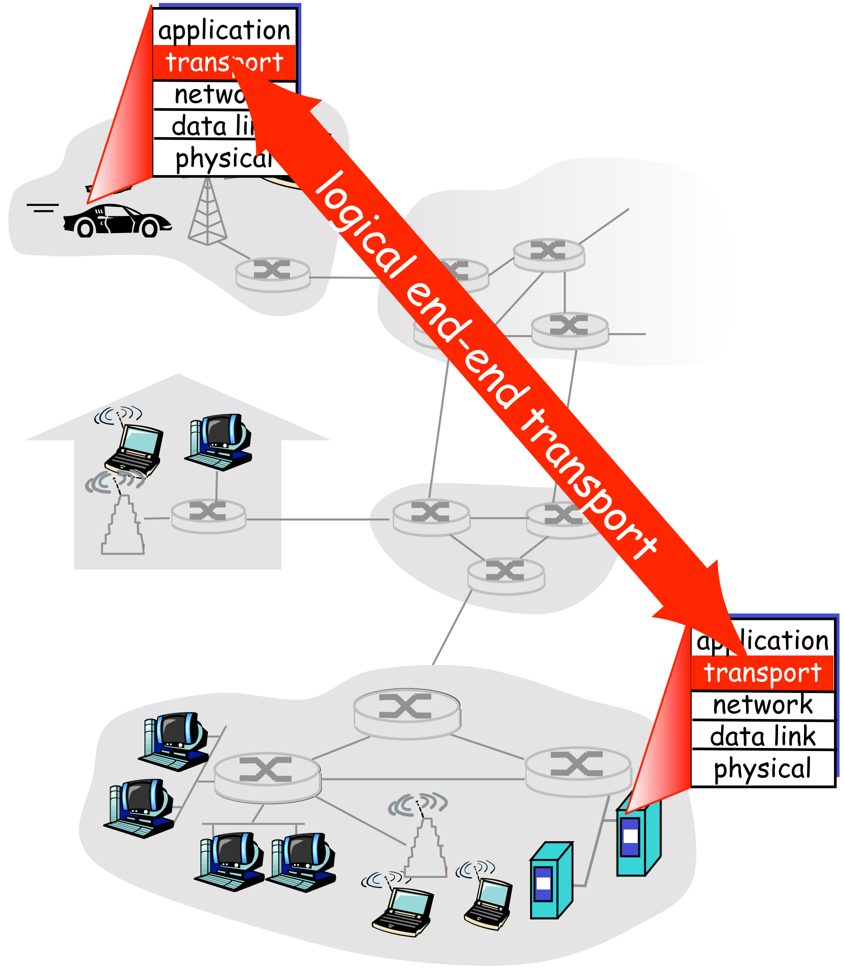
- understand principles behind transport layer services:
  - multiplexing/ demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

# Chapter 3 outline

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

# Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - 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 computers
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

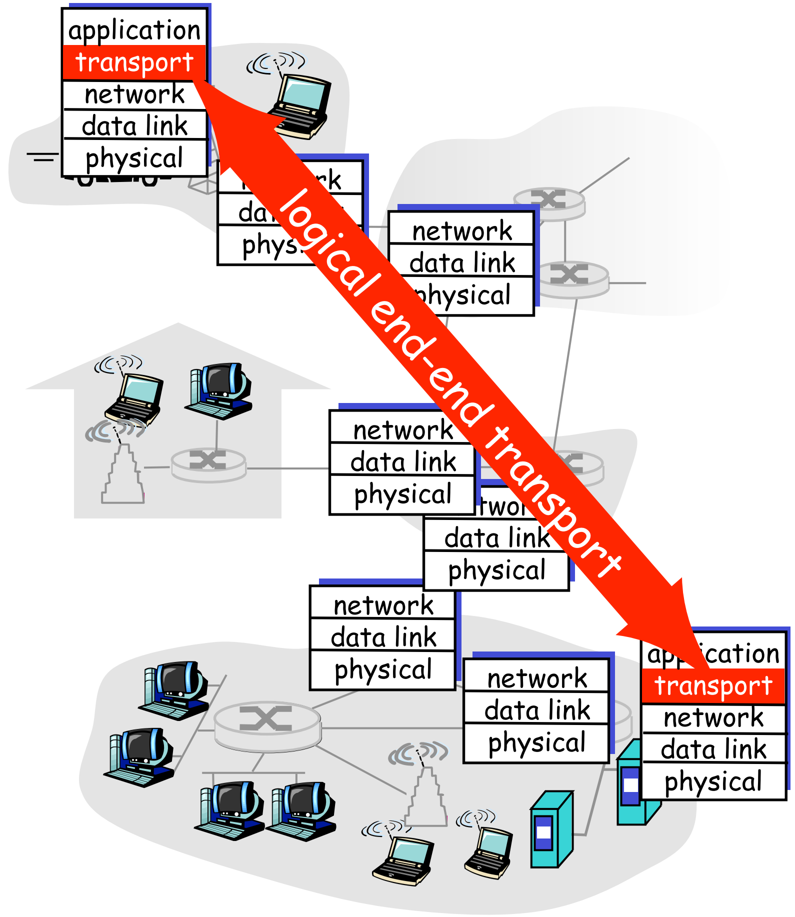
## Household analogy:

*12 kids sending letters to 12 kids*

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill (parents)
- 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



# Chapter 3 outline

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

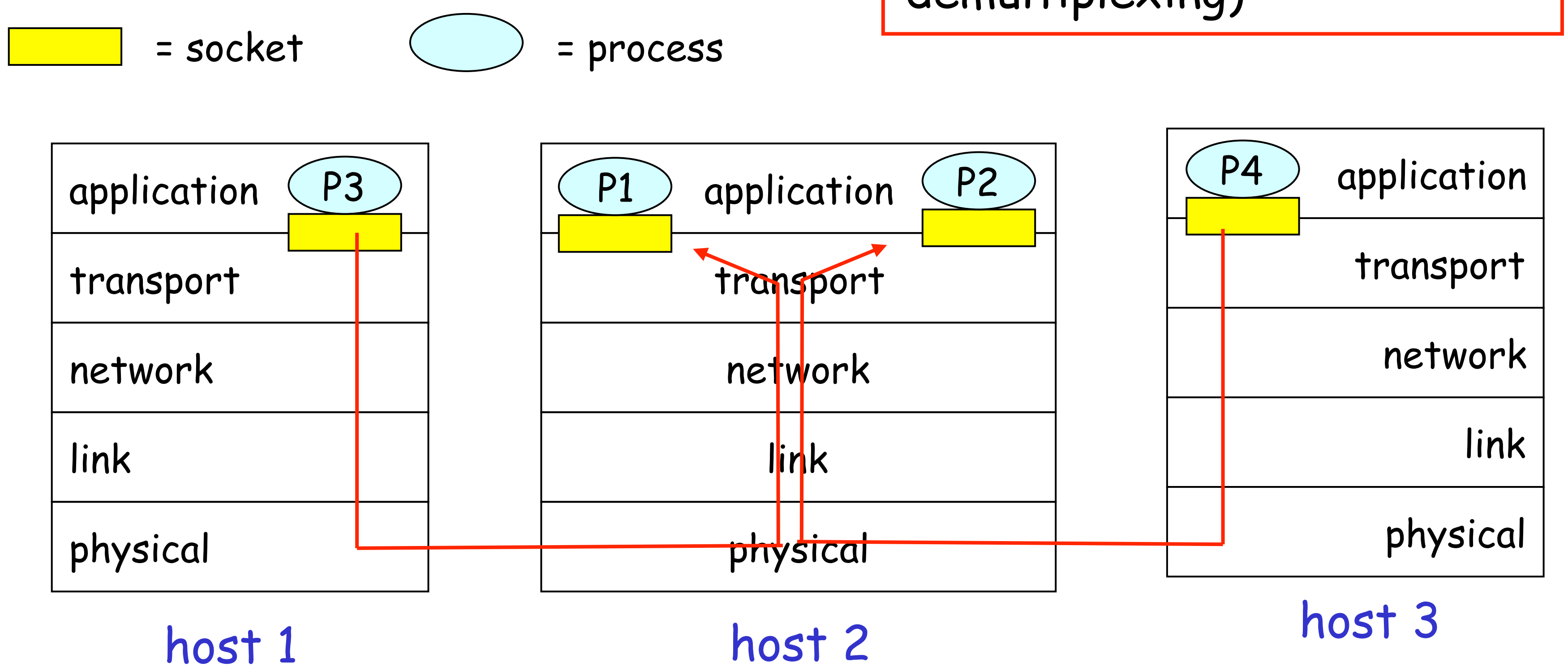
# Multiplexing/demultiplexing

**Demultiplexing at rcv host:**

delivering received segments to correct socket

**Multiplexing at send host:**

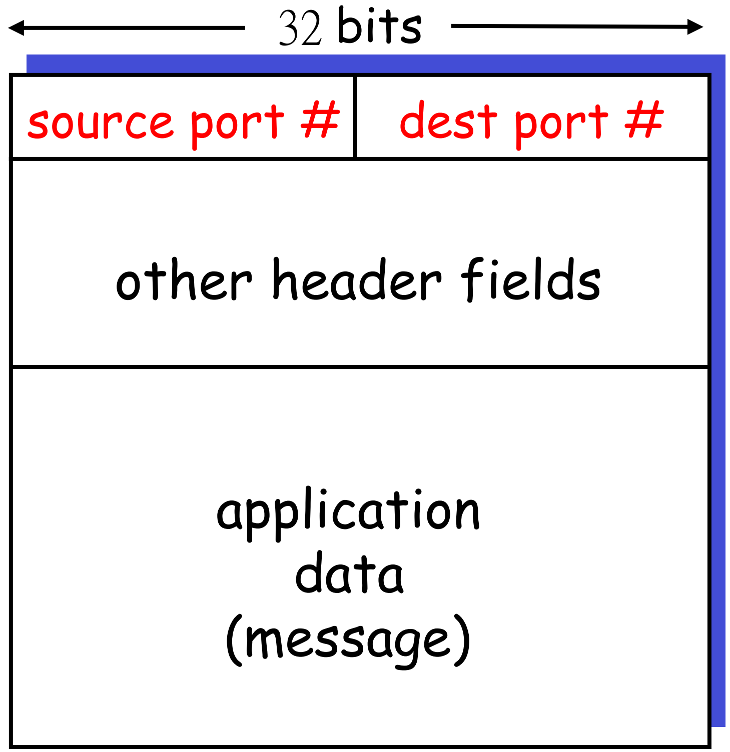
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



# Quiz Time!

# How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket




TCP/UDP segment format

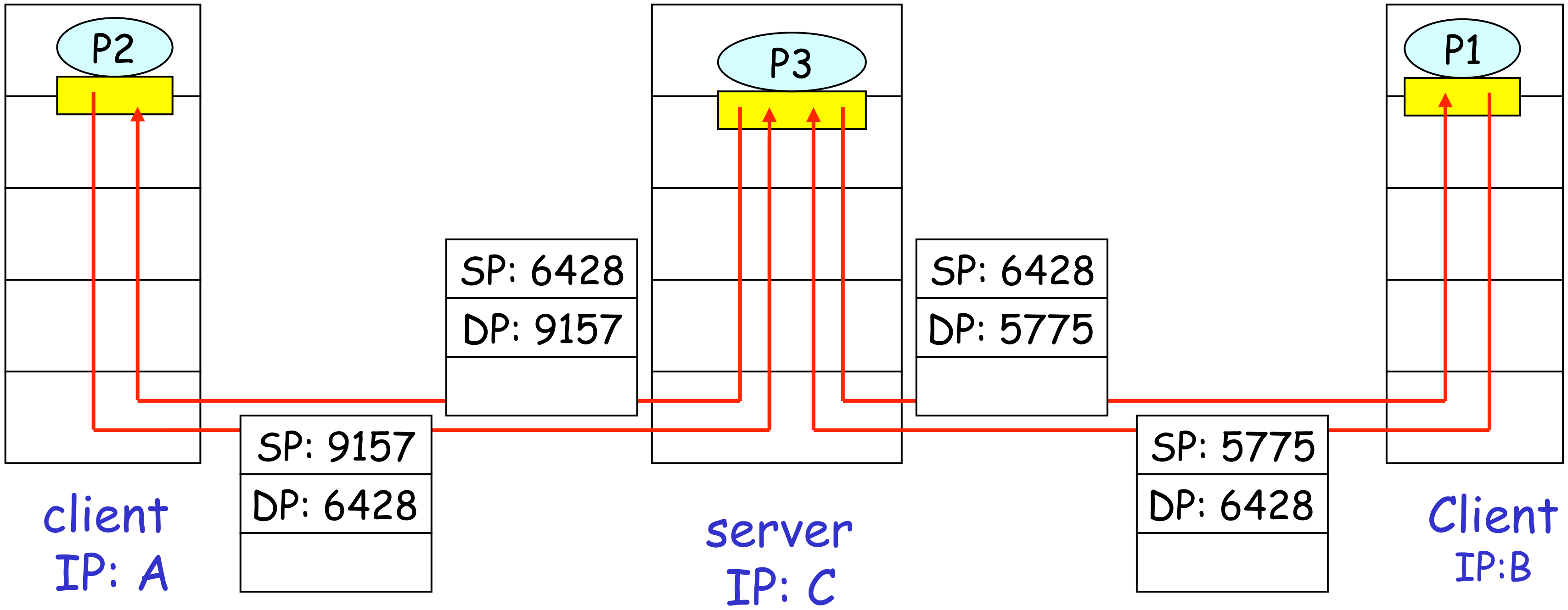
# Connectionless demultiplexing

- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

# Connectionless demux (cont)



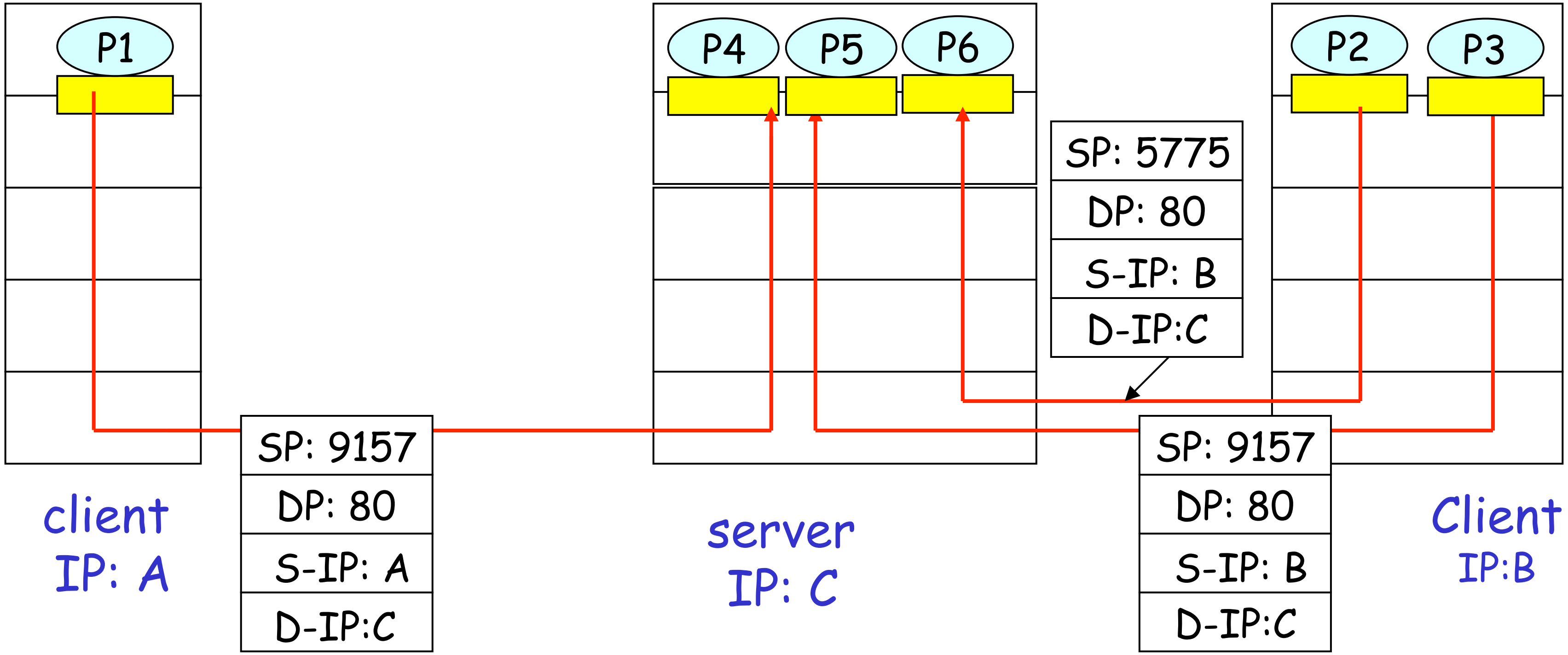

SP provides "return address"

# Connection-oriented demux

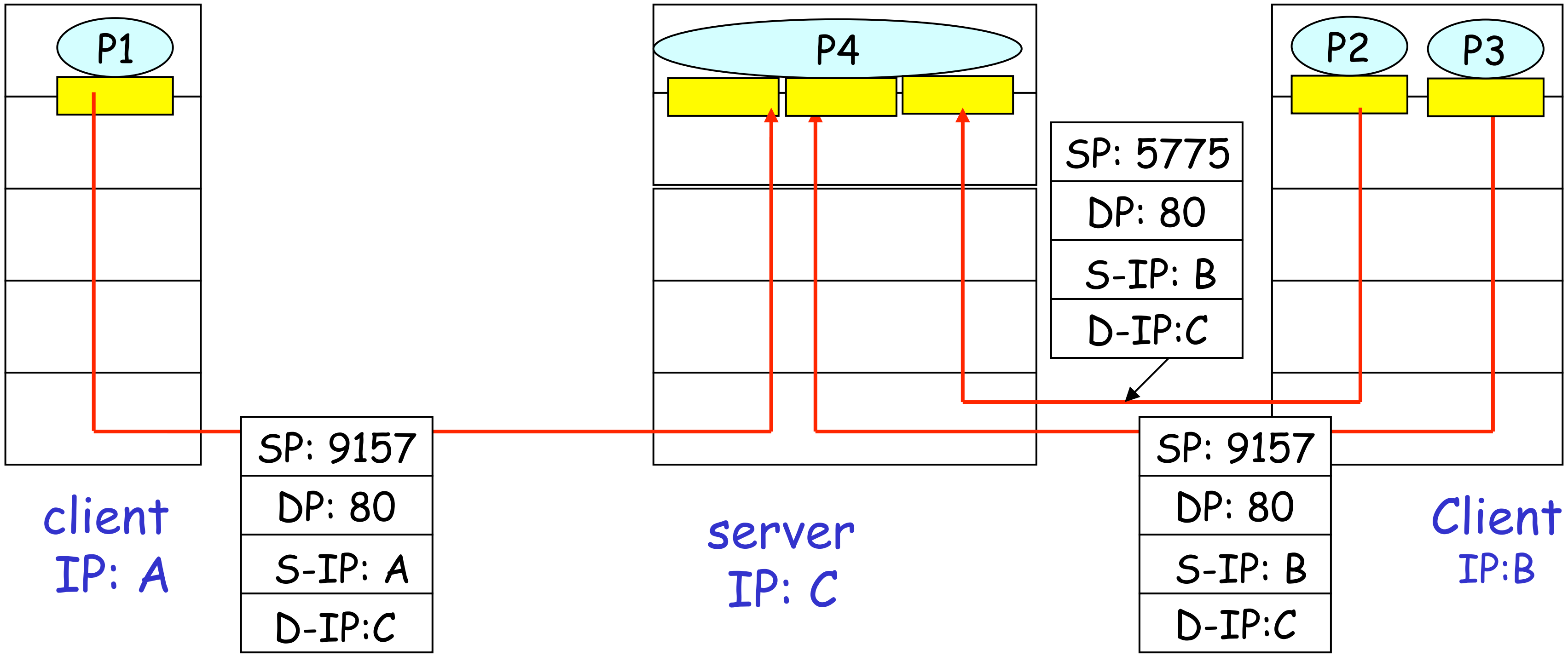
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket

- 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 (cont)



# Connection-oriented demux: Threaded Web Server



# Chapter 3 outline

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

# UDP: User Datagram Protocol [RFC 768]

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

**Why is there a UDP?**

# UDP: User Datagram Protocol [RFC 768]

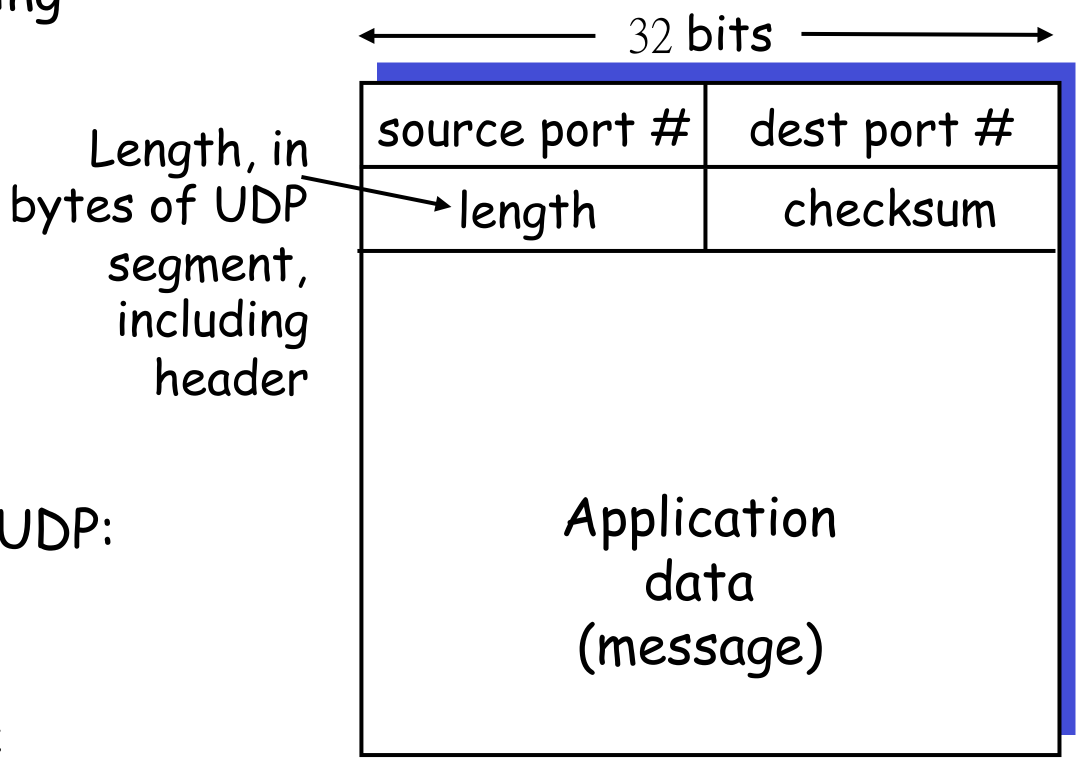
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

**Why is there a UDP?**

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

# UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!




UDP segment format

# UDP checksum

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

**Sender:**

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1'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

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

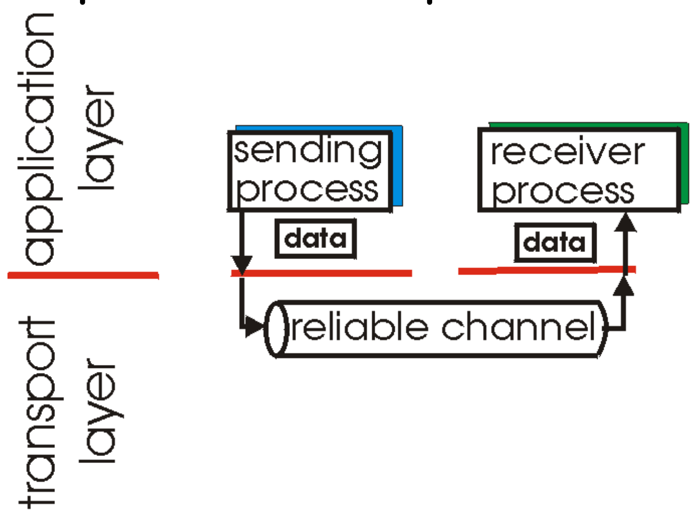
| | | |
|---|---|---|
| | | 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 |
| | | 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 |
| wraparound | 1 | 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 |
| sum | | 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0 |
| checksum | | 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1 |

# Chapter 3 outline

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

# Principles of Reliable data transfer

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

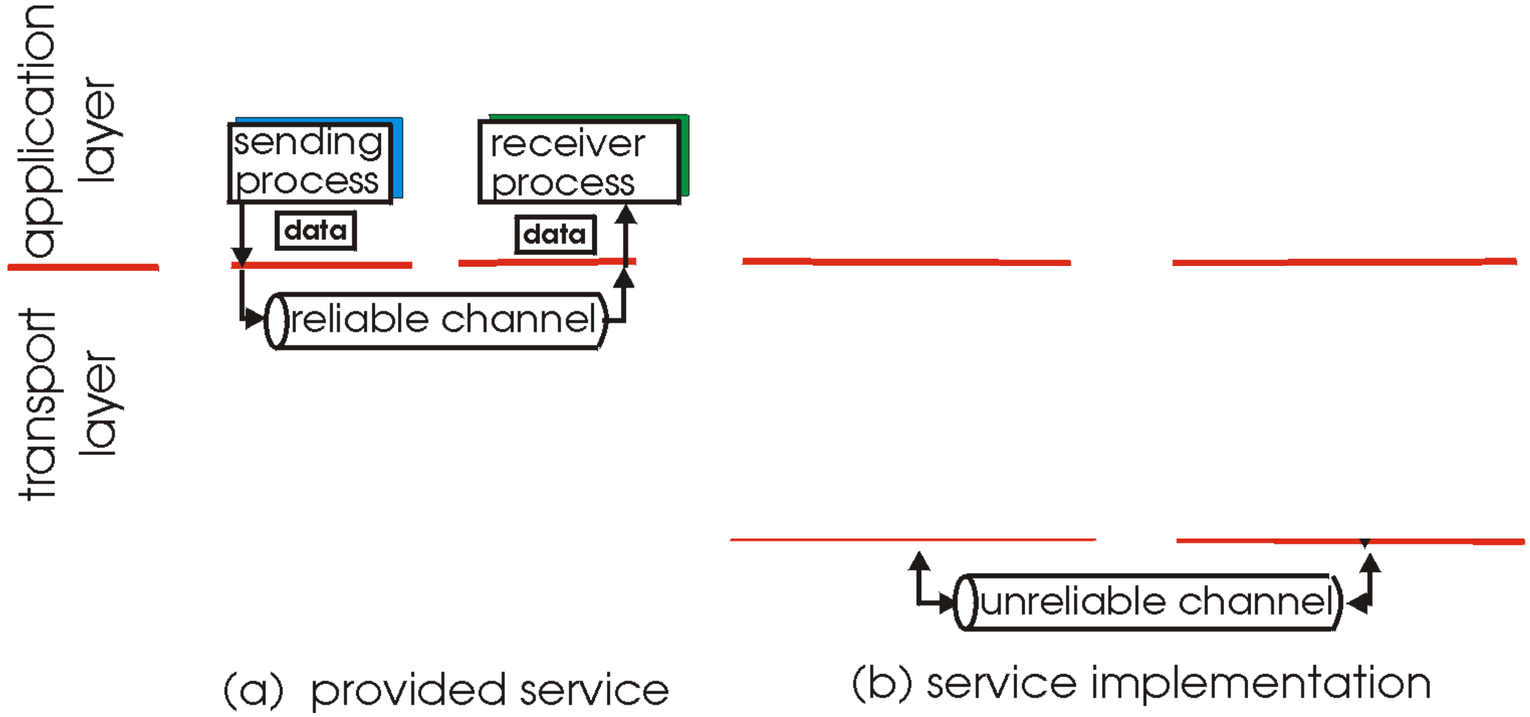



(a) provided service

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

# Principles of Reliable data transfer

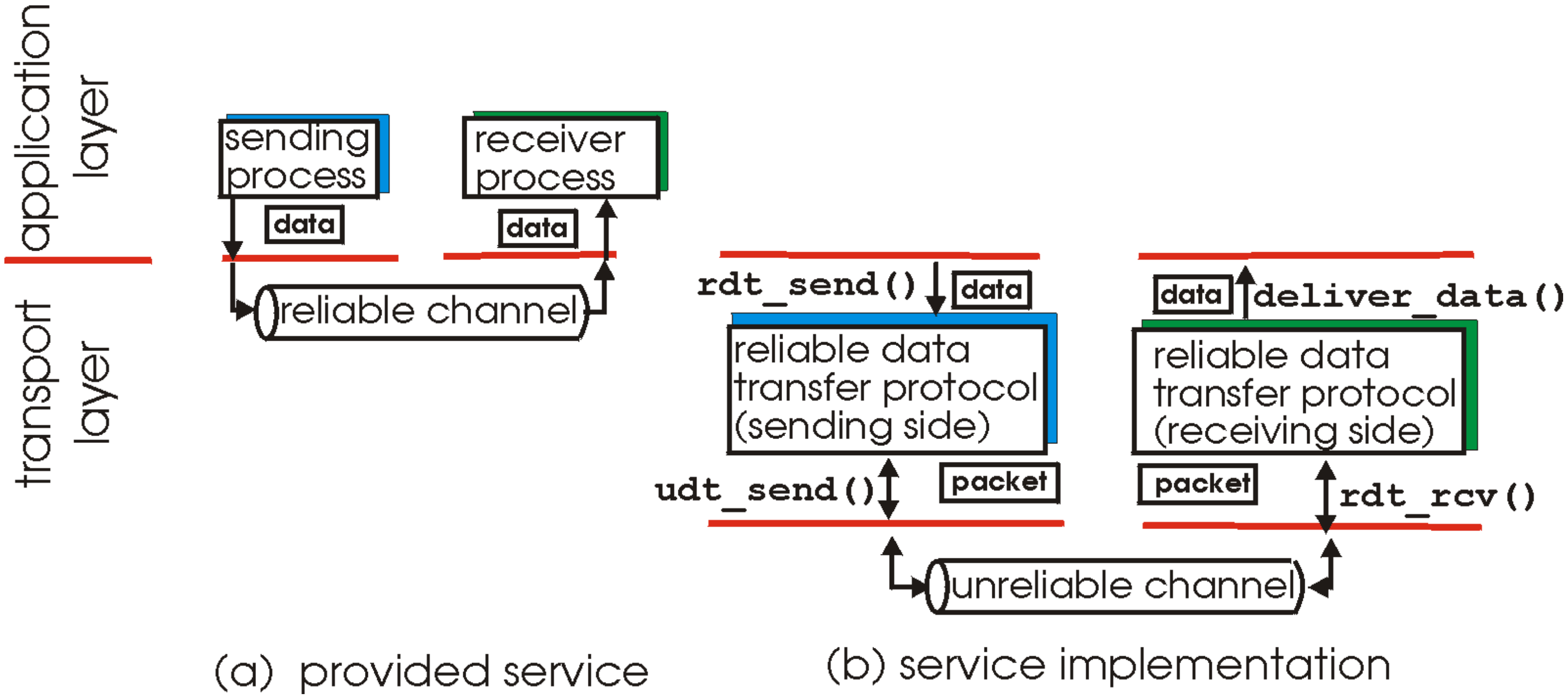
- important in app., transport, link layers
- top-10 list of important networking topics!




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

# Principles of Reliable data transfer

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

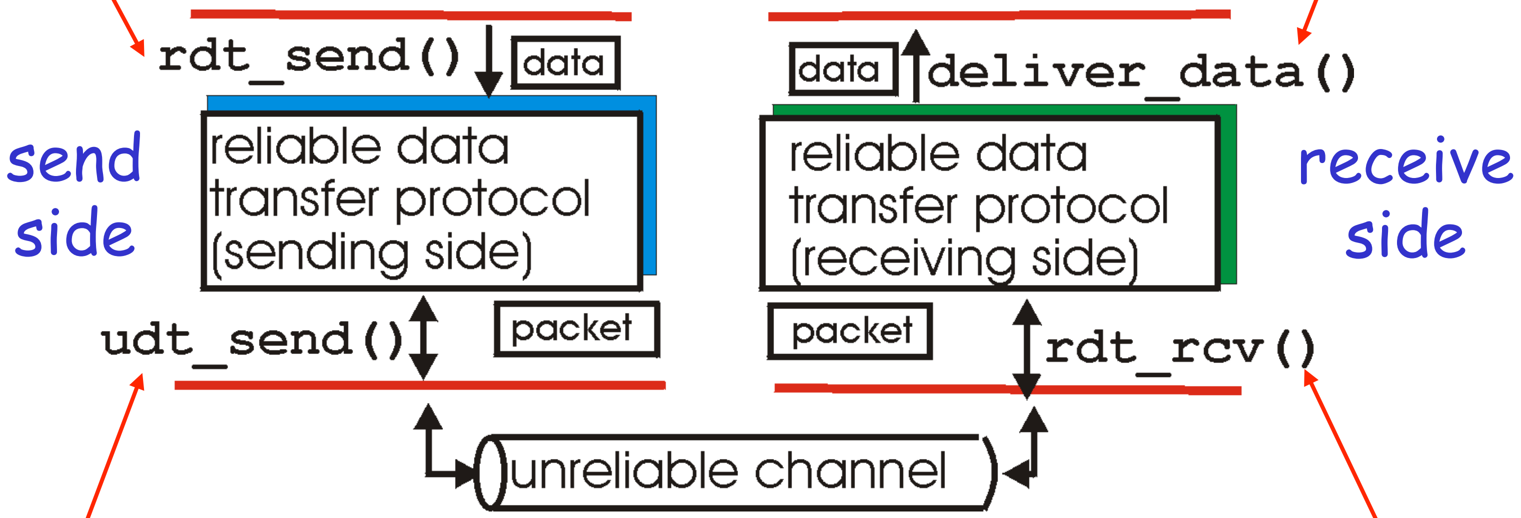



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Reliable data transfer: getting started

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

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



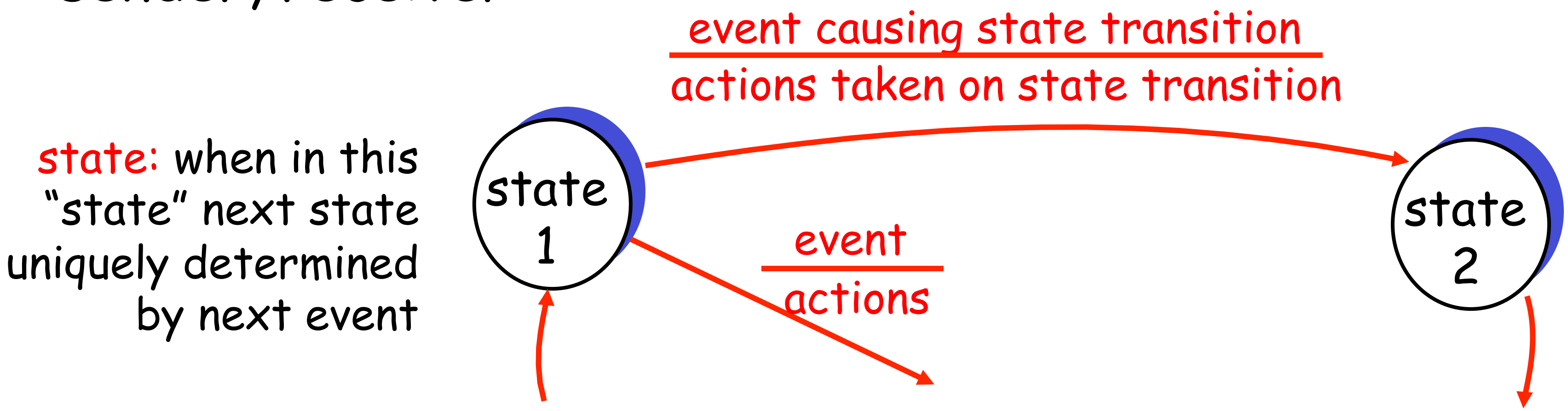

**udt_send():** called by rdt, to transfer packet over unreliable channel to receiver

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

# Reliable data transfer: getting started

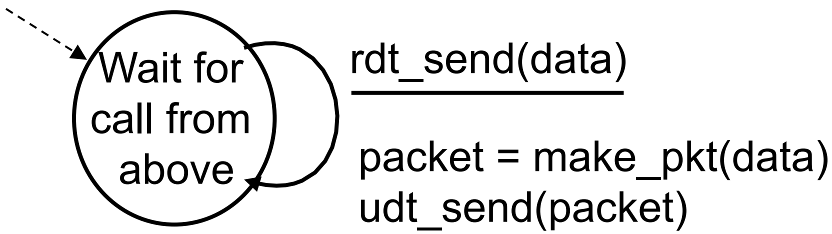
We'll:

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

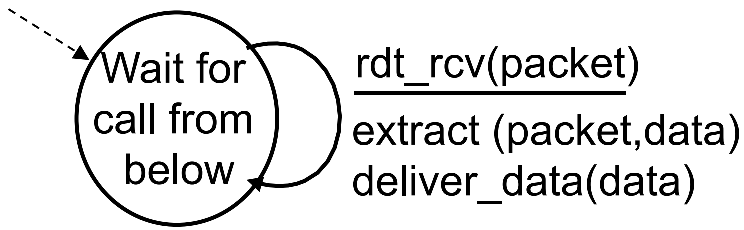


# Rdt1.0: reliable transfer over a reliable channel

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




sender

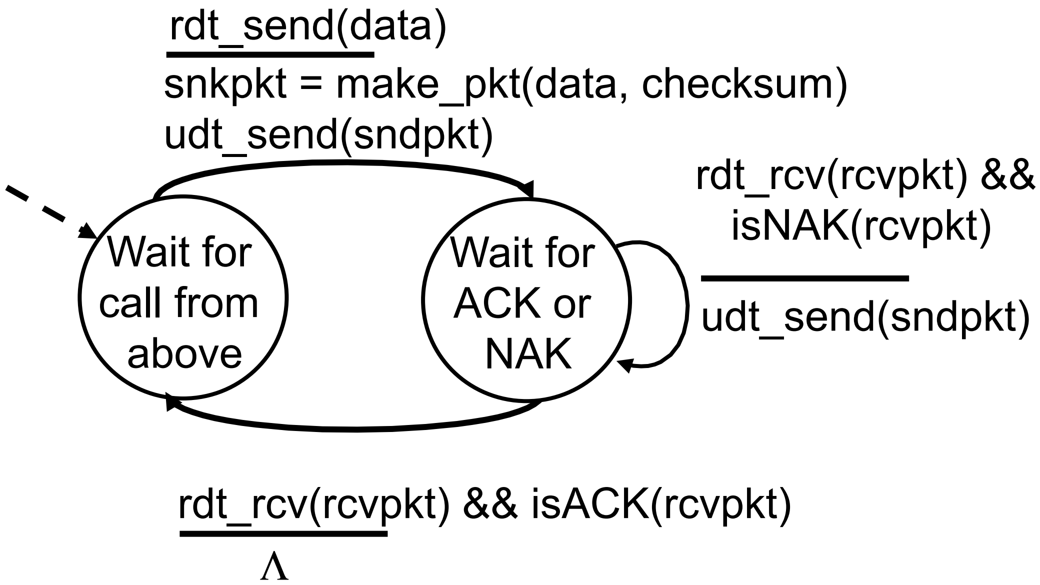



receiver

# Rdt2.0: channel with bit errors

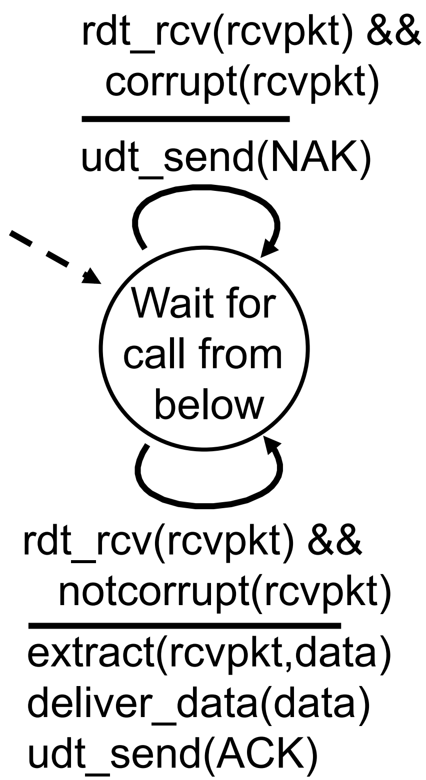
- underlying channel may flip bits in packet
  - Q: how to detect bit errors?
  - recall: UDP checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that packet received OK
  - *negative acknowledgements (NAKs):* receiver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

# rdt2.0: FSM specification

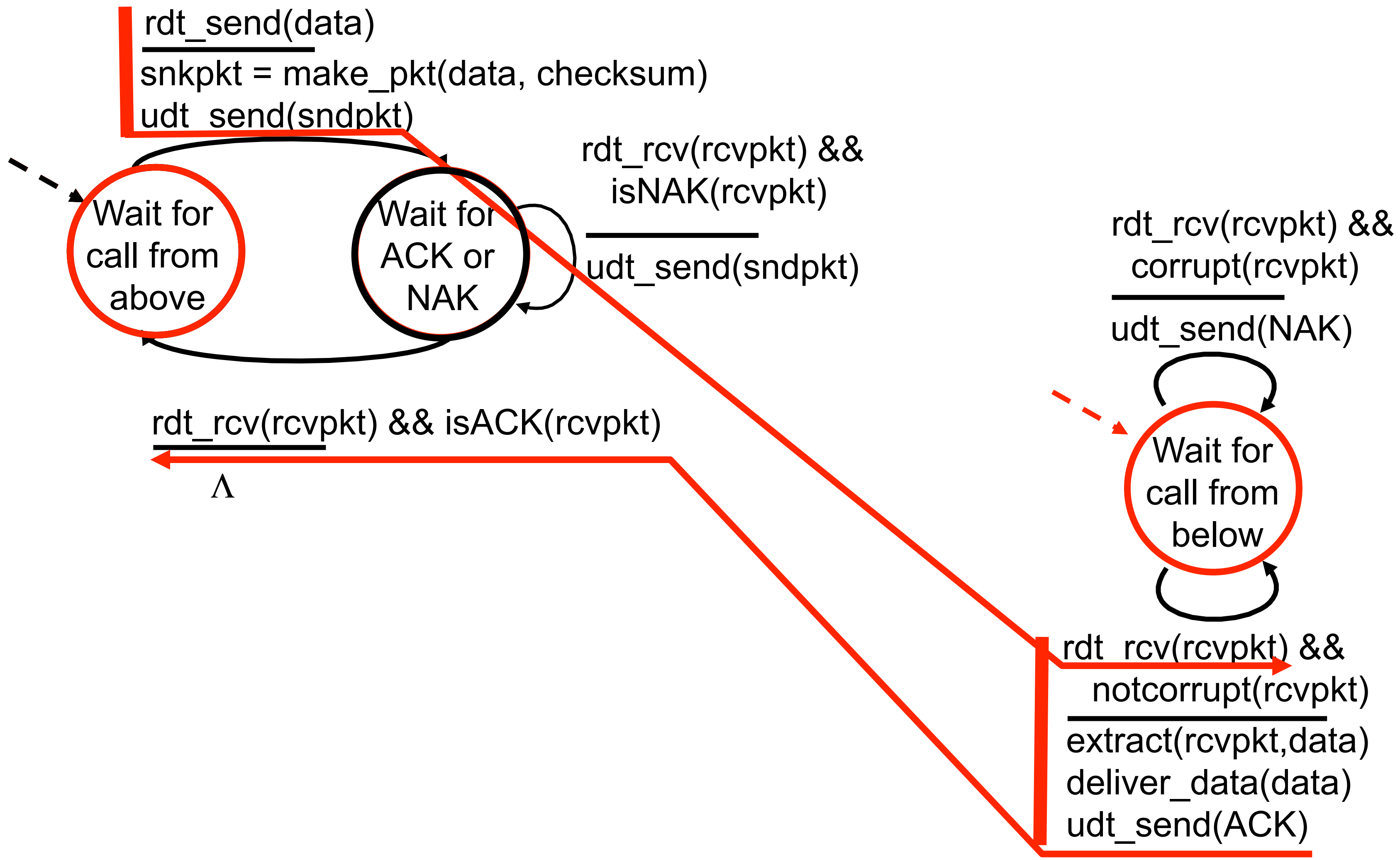



sender

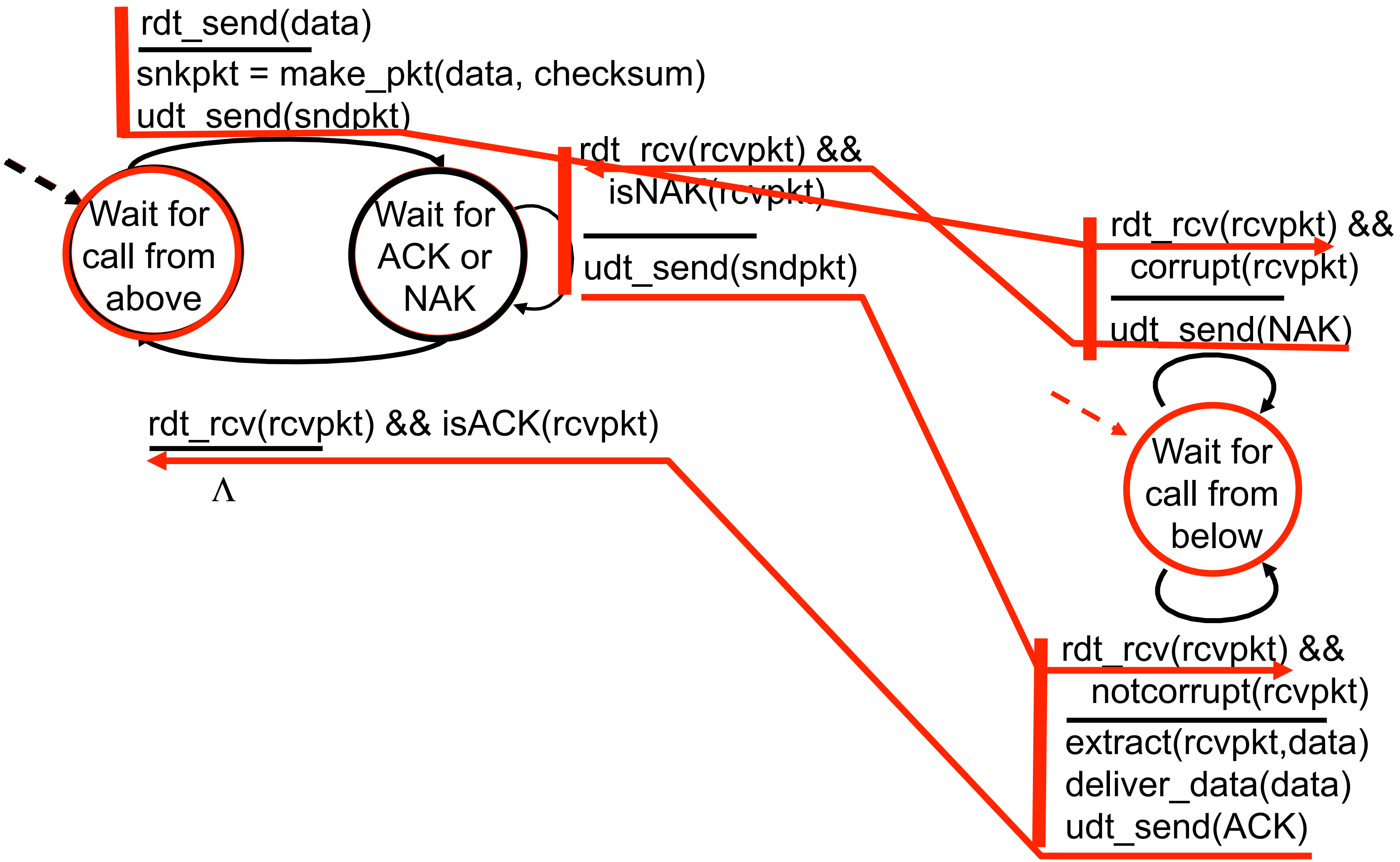
receiver



# 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!

What to do?

- Q?

# rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!

What to do?

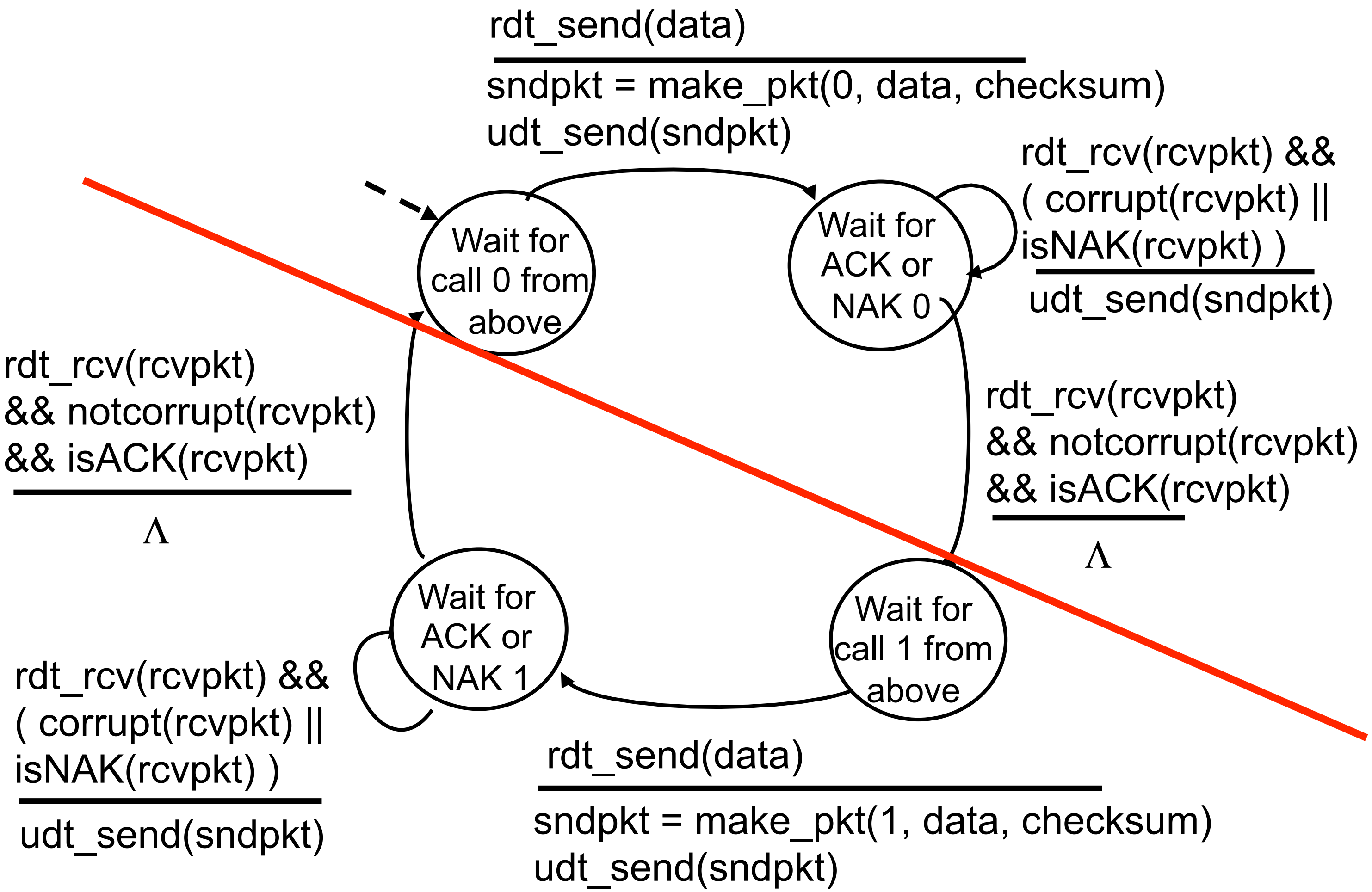
- sender ACKs/NAKs receiver's ACK/NAK? What if this sender ACK/NAK corrupted?... (see an end?)
- retransmit anyway, but this might cause retransmission of correctly received packet!

Handling duplicates:

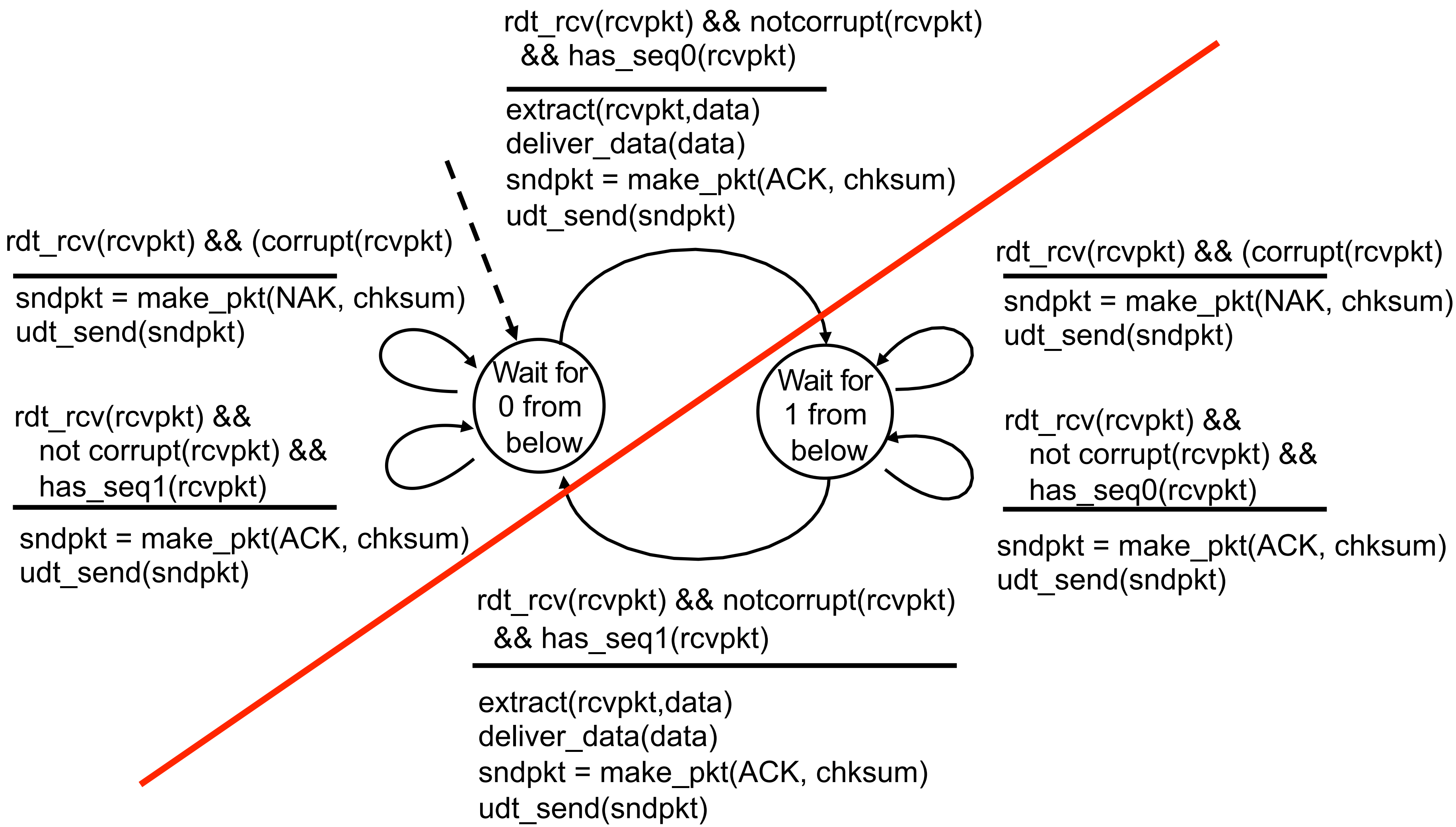
- sender adds *sequence number* to each packet
- sender retransmits current packet if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate packet

stop and wait
Sender sends one packet, then waits for receiver response

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



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



# Quiz Time!

# rdt2.1: discussion

**Sender:**

- sequence # added to packet
- two sequence #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" packet has 0 or 1 sequence #

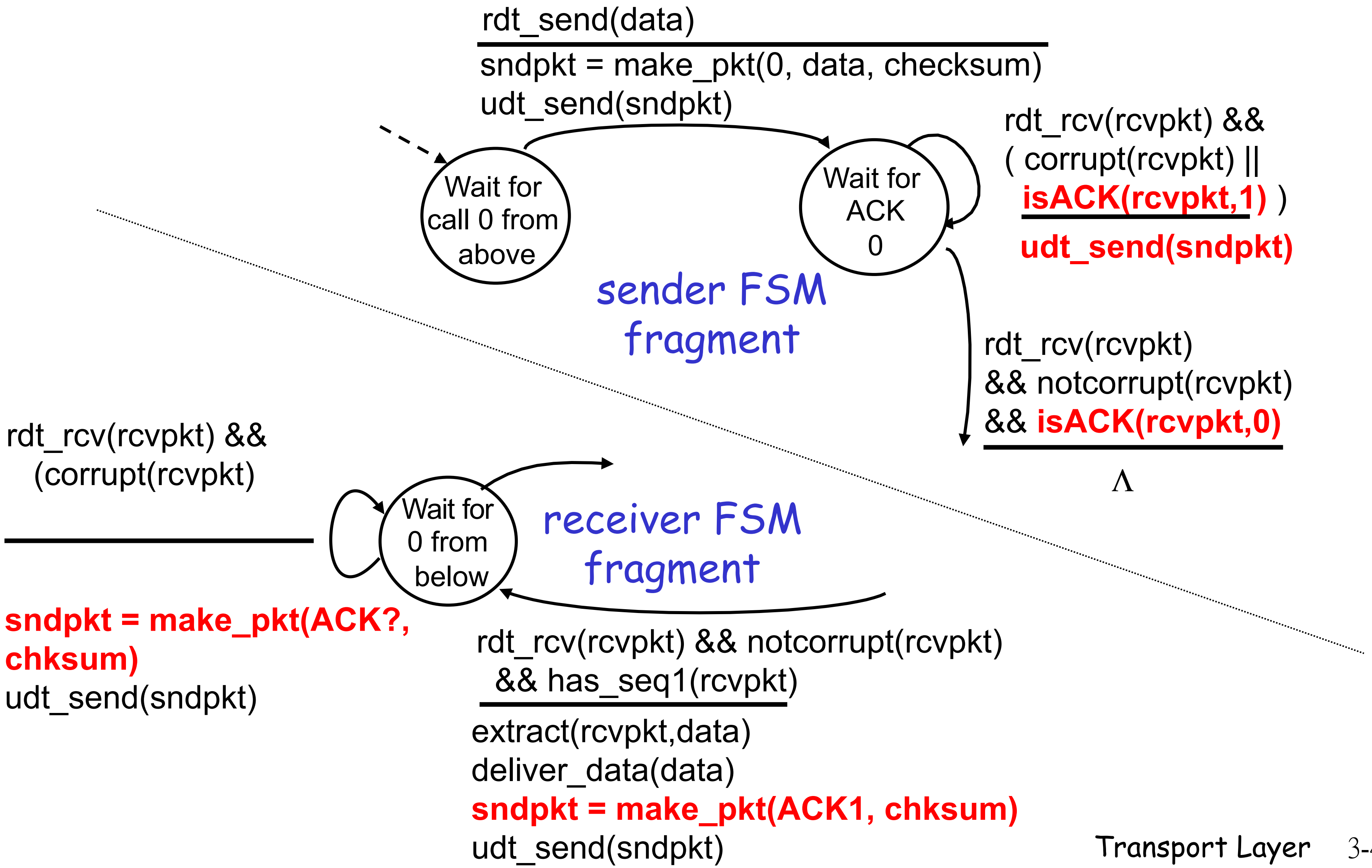
**Receiver:**

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

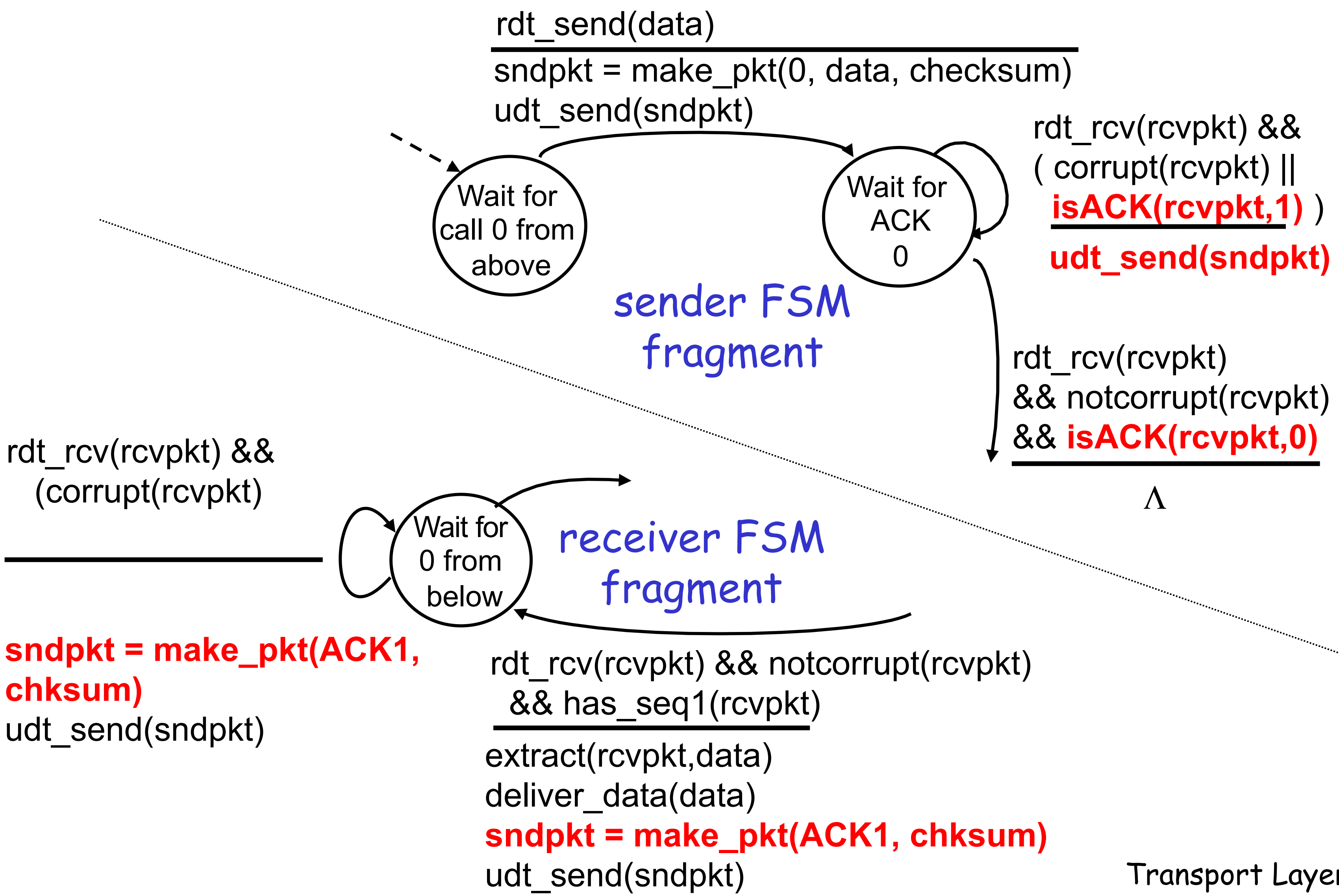
# rdt2.2: a NAK-free protocol

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

# rdt2.2: sender, receiver fragments



# rdt2.2: sender, receiver fragments



# Quiz Time!

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

**New assumption:** underlying channel can also lose packets (data or ACKs)

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

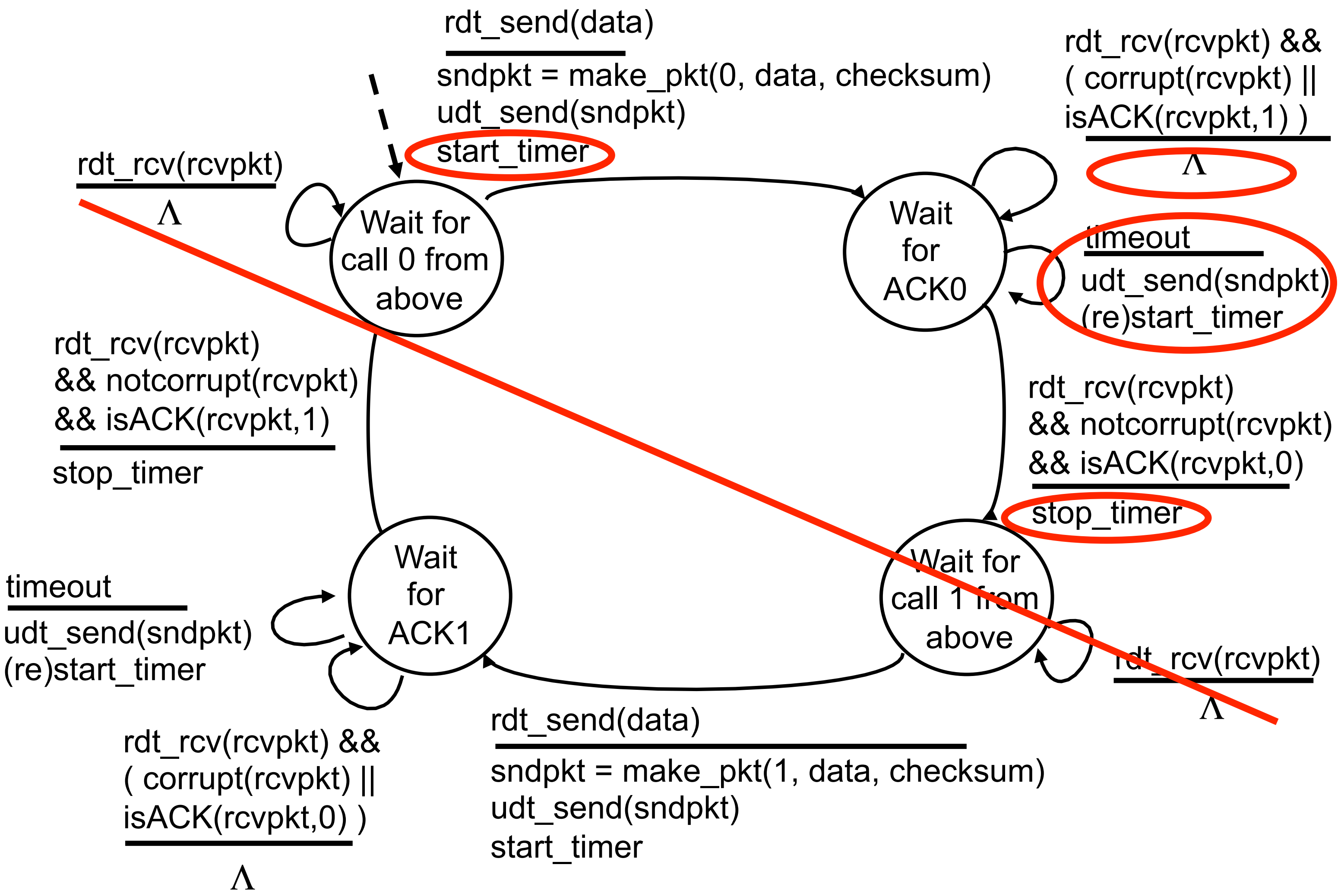
**Q:** how to deal with loss?

- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

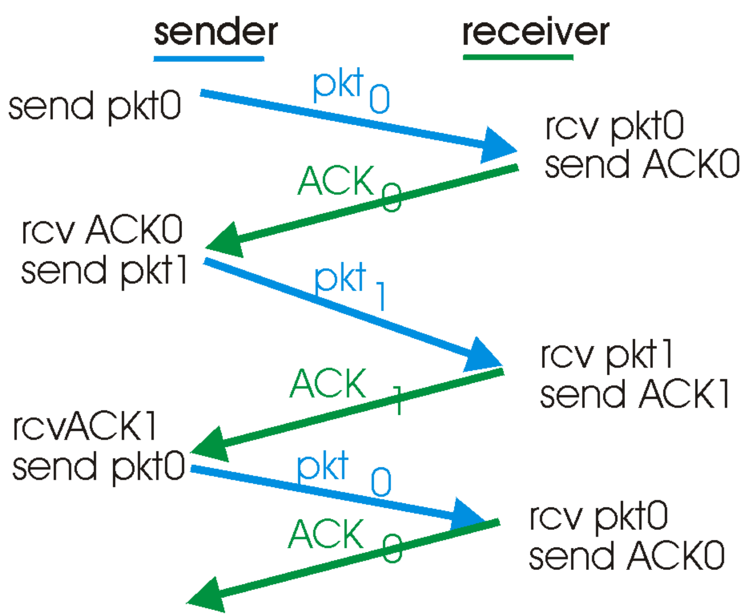
**Approach:** sender waits "reasonable" amount of time for ACK

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

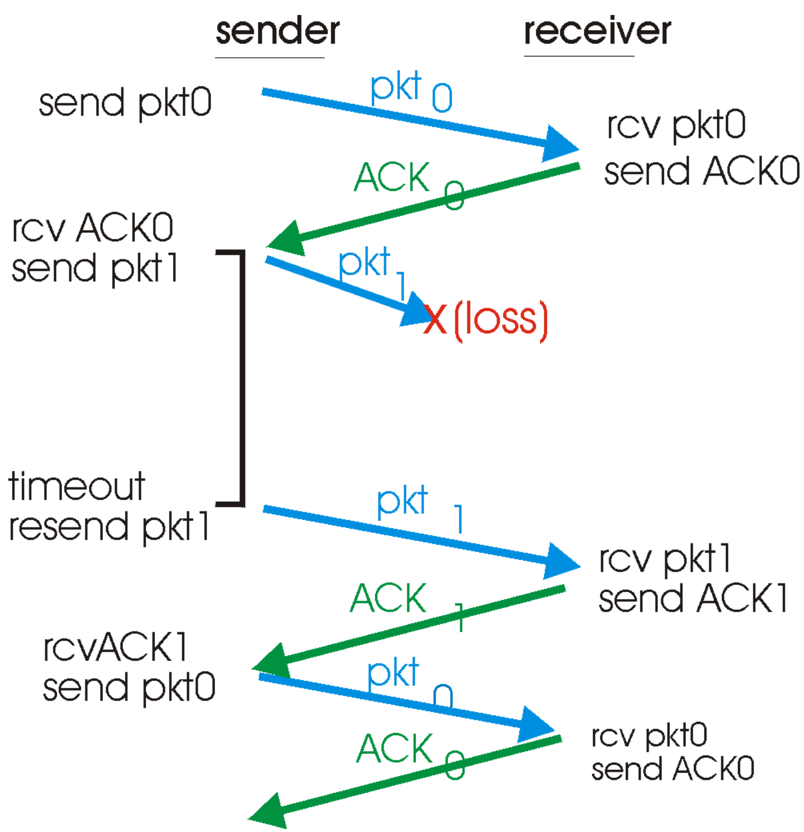
# rdt3.0 sender



# rdt3.0 in action

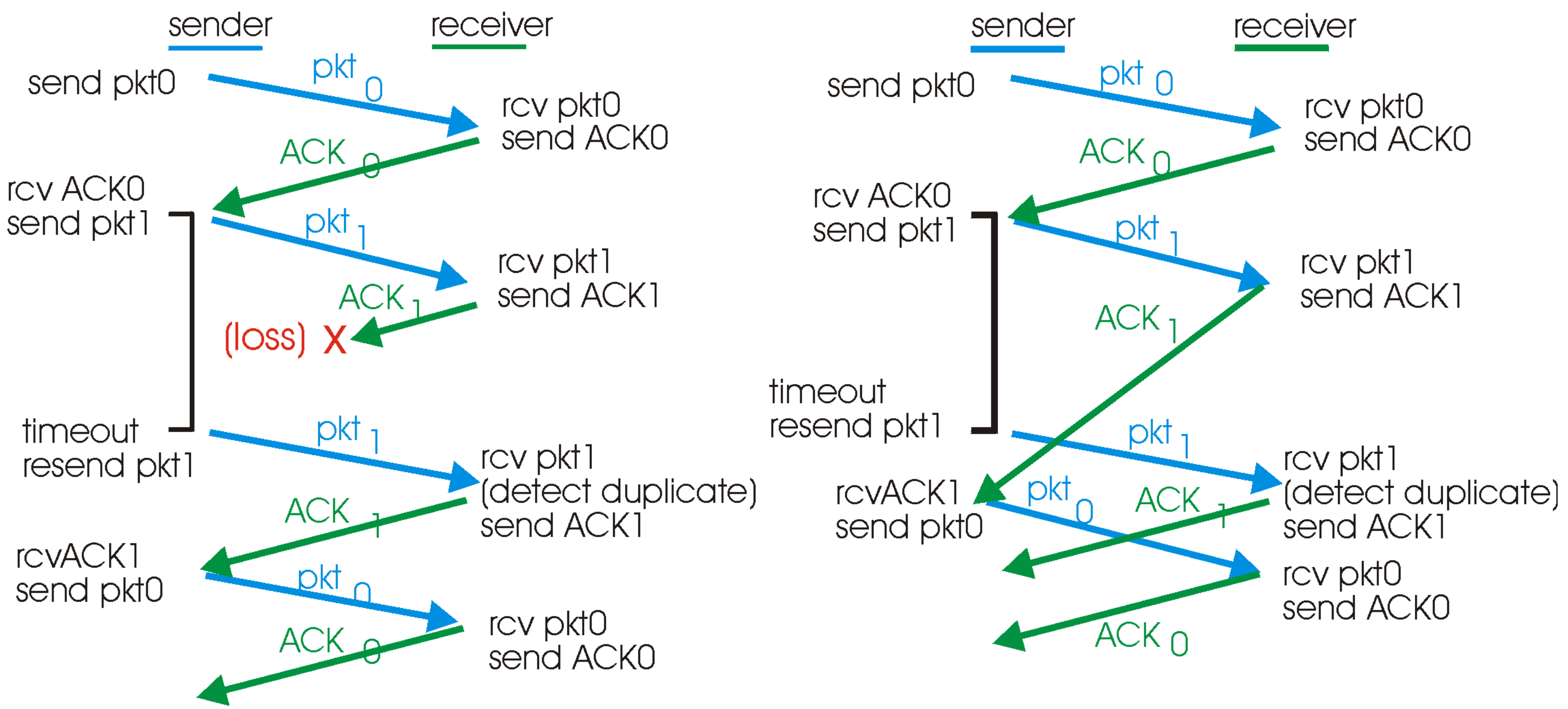



(a) operation with no loss




(b) lost packet

# rdt3.0 in action




(c) lost ACK

(d) premature timeout

# Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

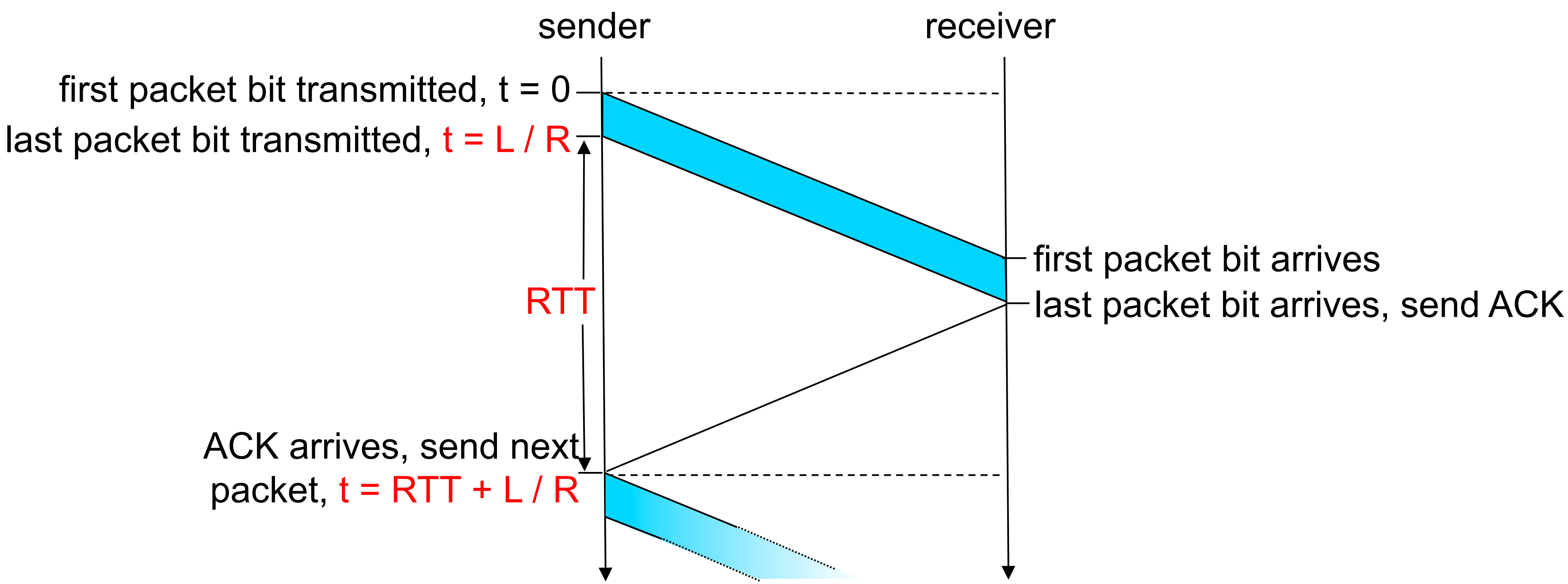
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

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

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

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

# rdt3.0: stop-and-wait operation

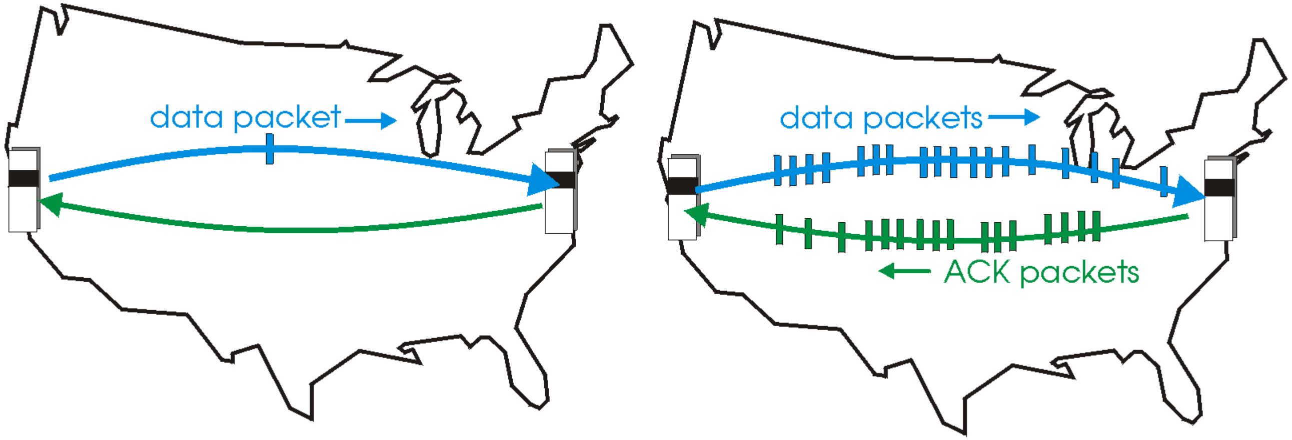



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

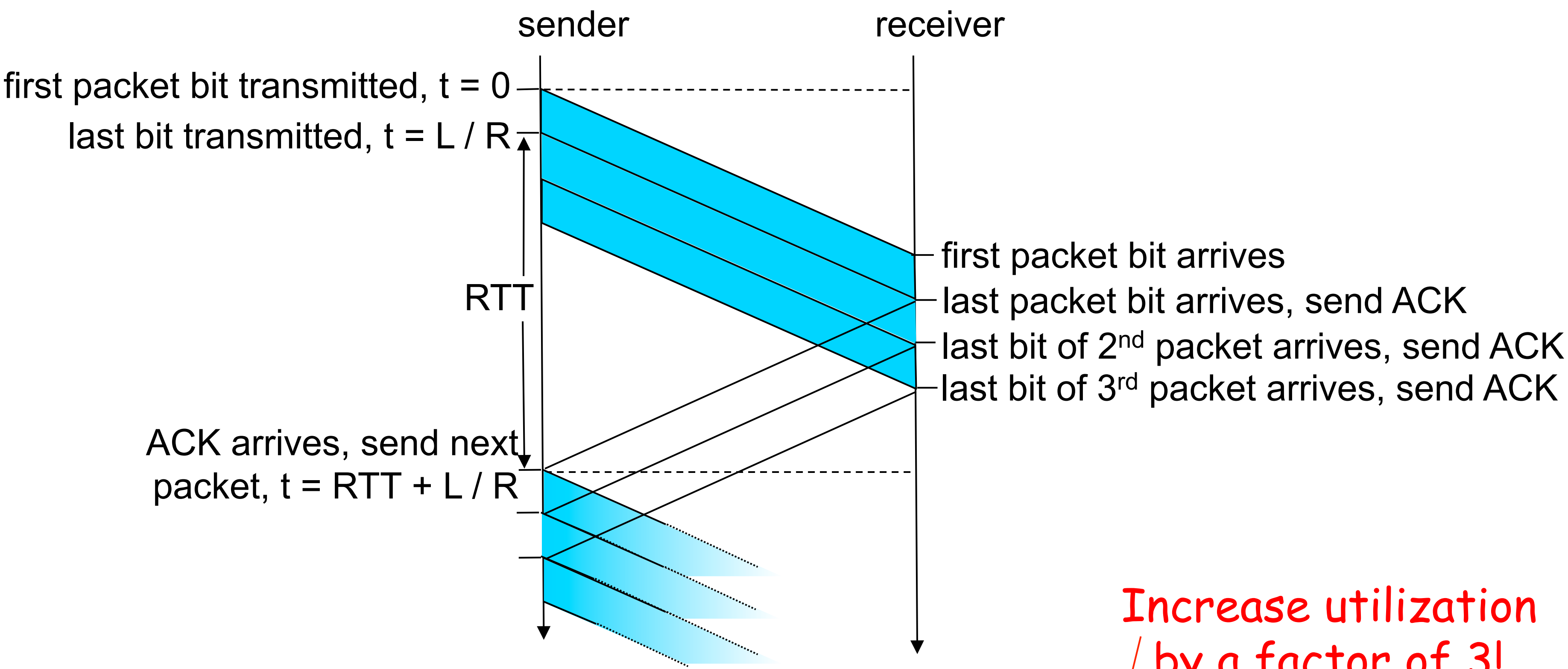
- range of sequence numbers must be increased
- buffering at sender and/or receiver




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

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

# Pipelining: increased utilization



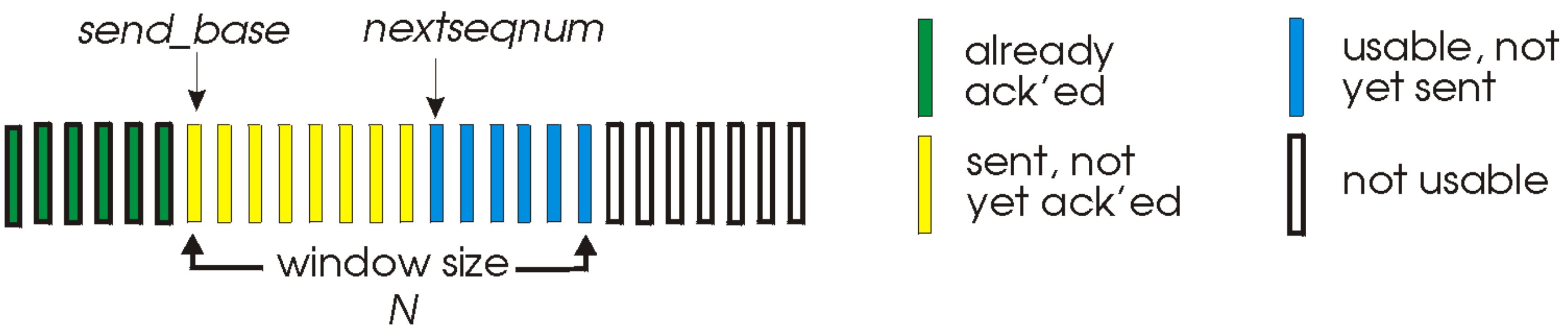

Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

# Go-Back-N

## Sender:

- Sequence # in packet header, k-bit
- "window" of up to N, consecutive unack'ed packets allowed




- ACK(n): ACKs all packets up to, including sequence # n
  - Cumulative ACK
- Timer for each in-flight packet batch (per send_base)
- *timeout(n):* retransmit packet n and all higher sequence # packets in window (send_base to nextseqnum)

# GBN: sender extended FSM

rdt_send(data)

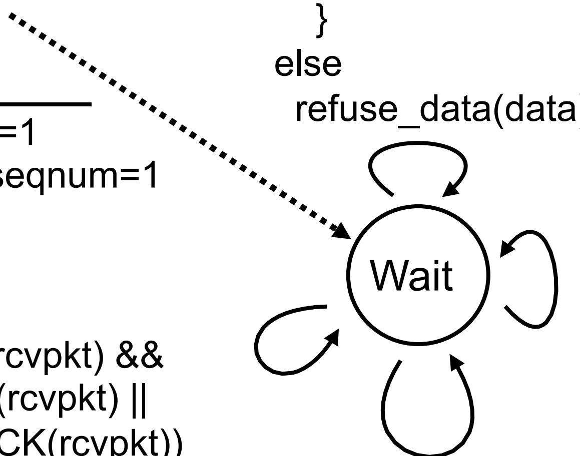
---

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

---

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

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isOLDACK(rcvpkt))

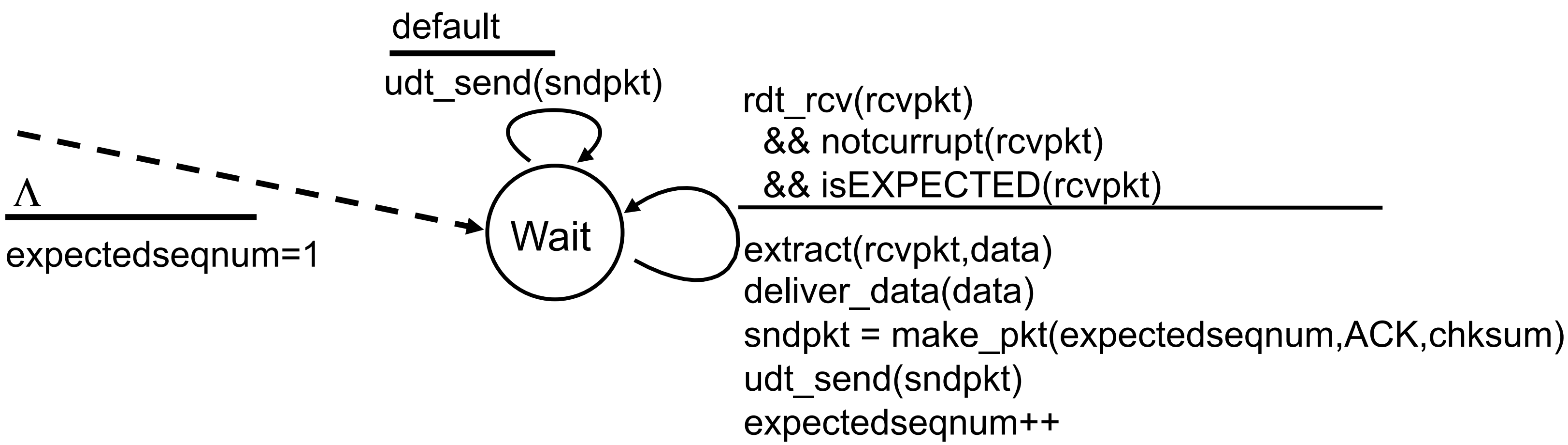
---

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)  && isNEWACK (rcvpkt)

---

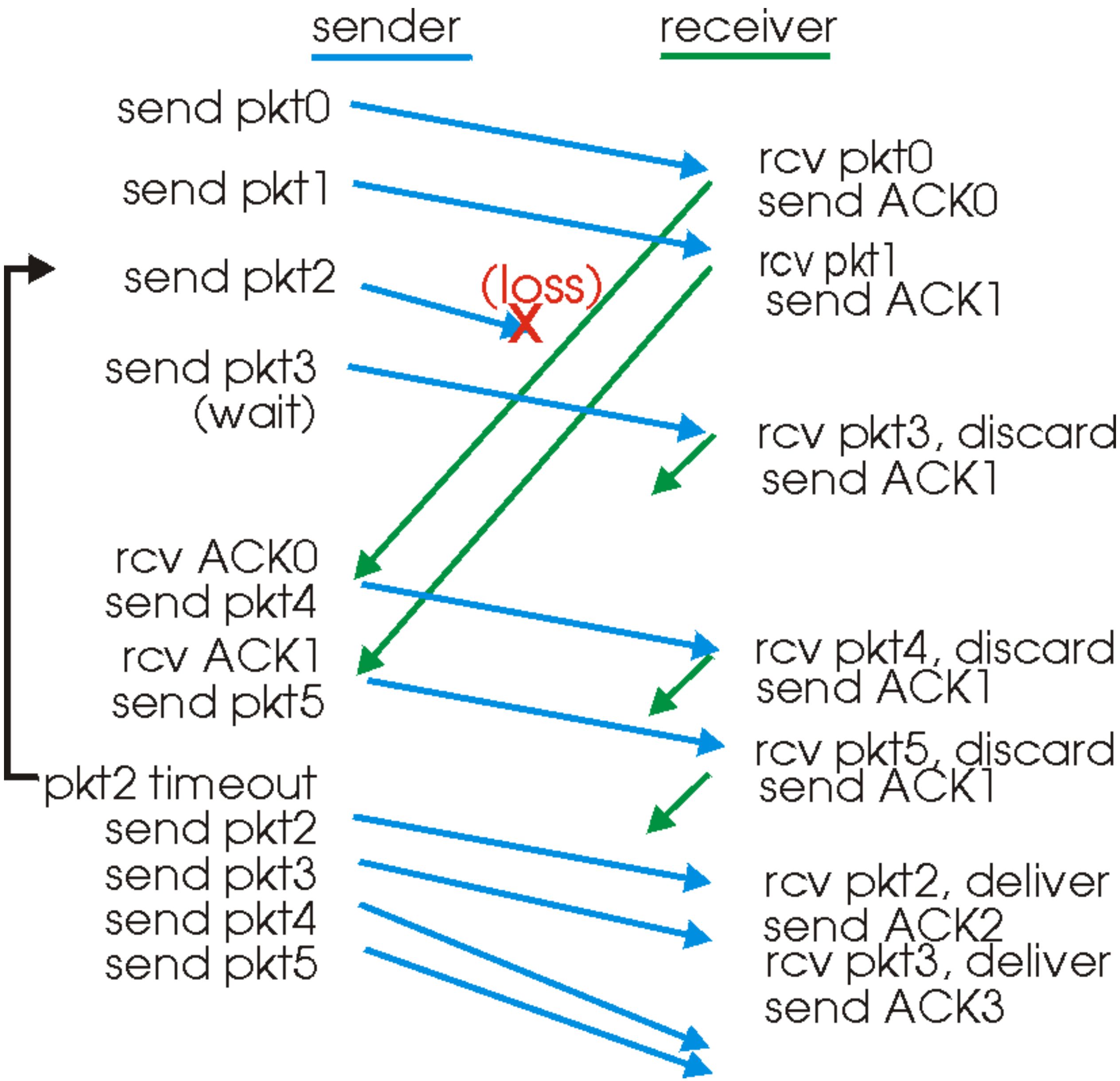
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer **(no pkt outstanding)**
else {
    stop_timer **(for the old base)**
    start_timer **(for the new base)**
}

# GBN: receiver extended FSM




- Principle:
  - if it's the expected data packet, send ACK
  - Else, send NAK
- Making it ACK-only:
  - Send ACK for correctly-received packet with highest *in-order* sequence #
    - Need to remember **expectedseqnum**
  - For corrupted or out-of-order packet:
    - discard (don't buffer) -> no receiver buffering!
    - ACK packet with highest in-order sequence #

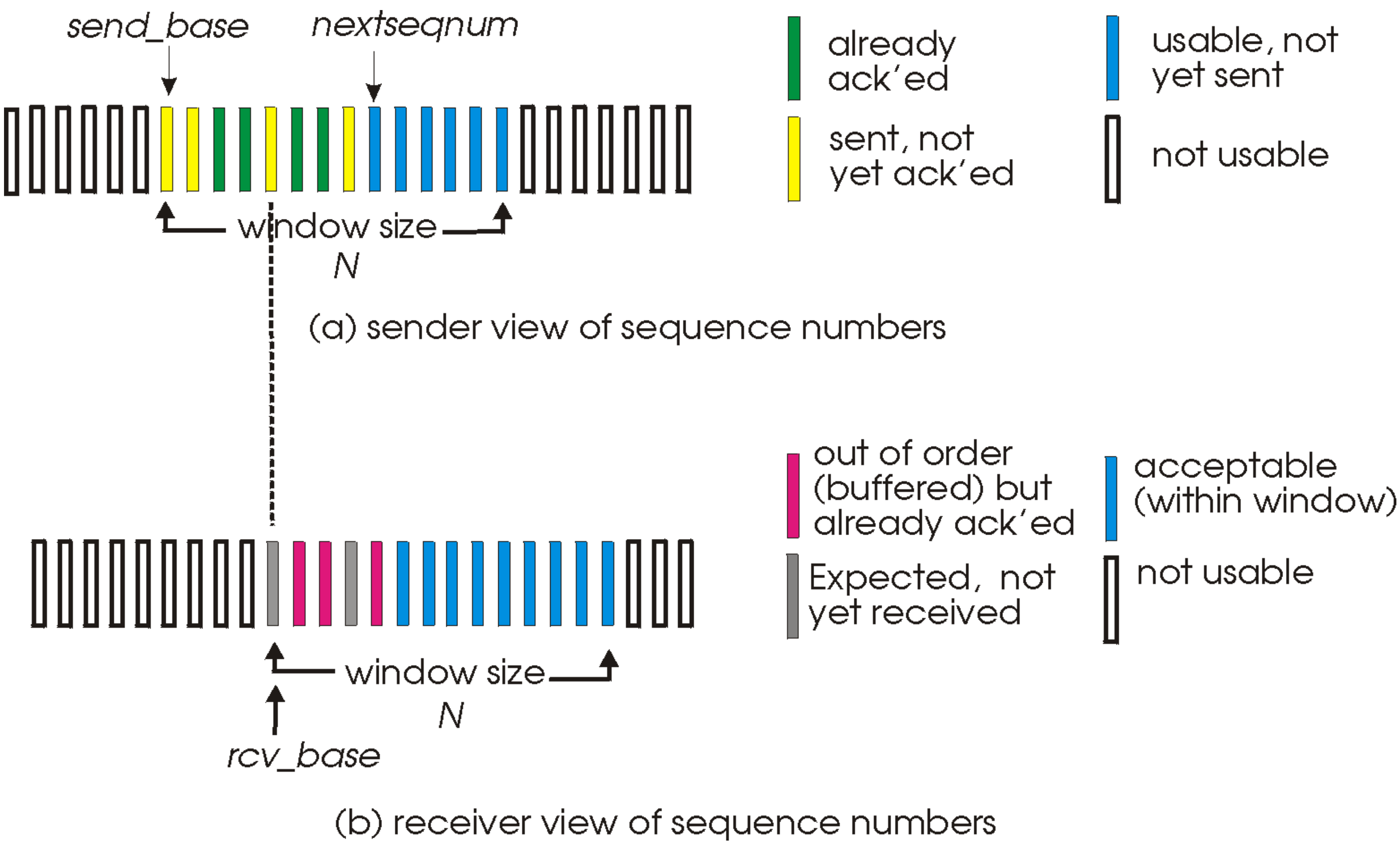
# GBN in action



# Selective Repeat

- Actually easier to understand
- Receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only re-sends packets for which ACK not received
  - sender timer for each unACKed packet
- Sender window
  - N consecutive sequence #'s
  - again limits sequence #s of sent, unACKed packets

# Selective repeat: sender, receiver windows



# Selective repeat

## sender

**Data from above :**

- If next available sequence # in window, send packet

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

- Mark packet n as received
- If n smallest unACKed packet, advance window base to next unACKed sequence #

**timeout(n):**

- Resend packet n, restart timer

## receiver

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

- Send ACK(n)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

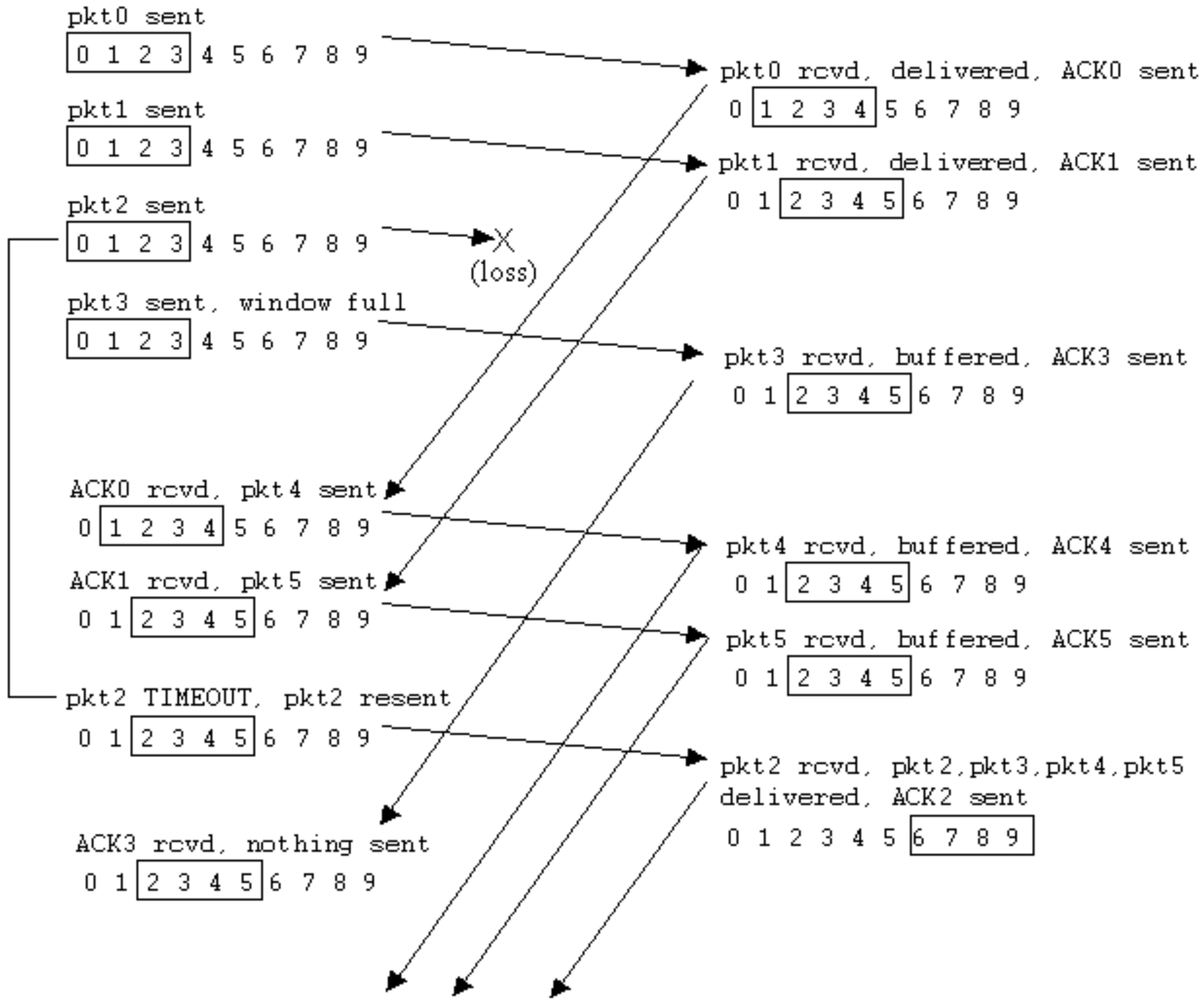
**pkt n in** [rcvbase-N, rcvbase-1]

- Send ACK(n)

**otherwise:**

- Ignore

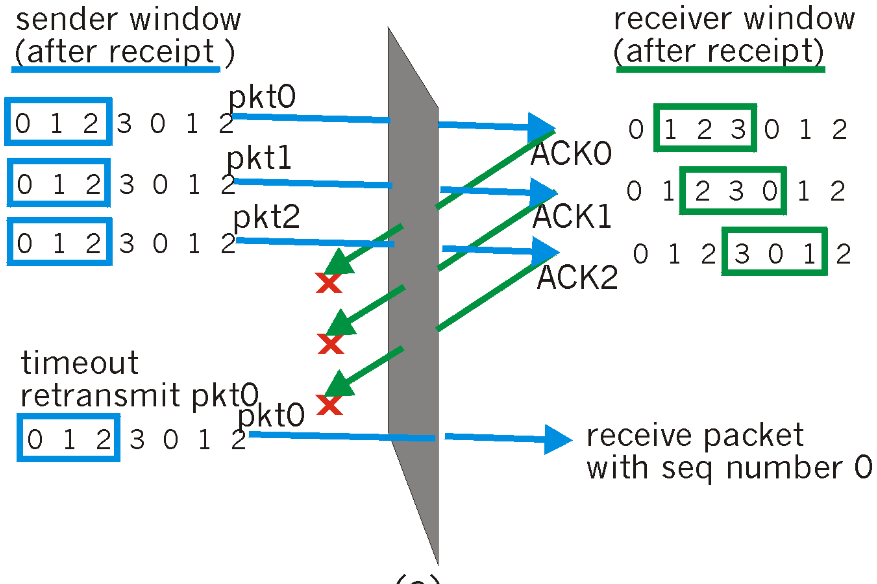
# Selective repeat in action



# Selective repeat: dilemma

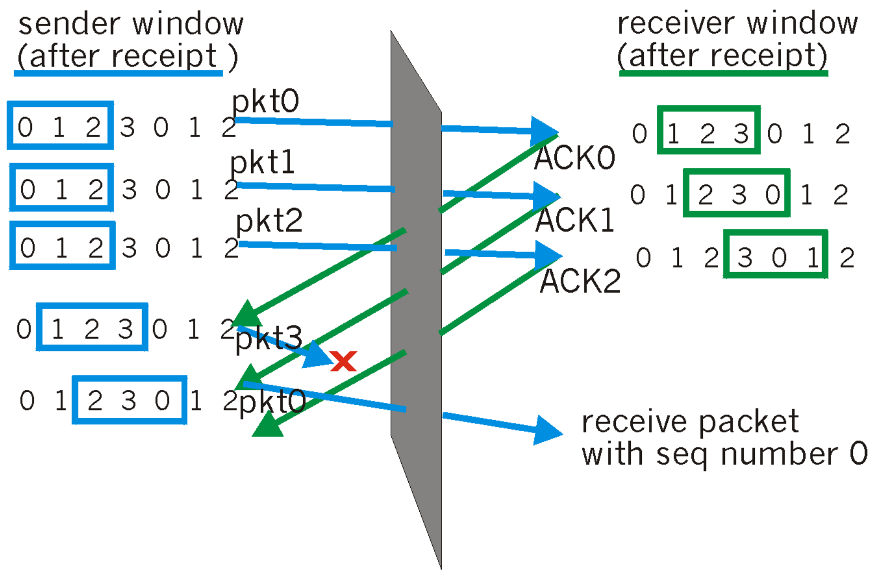
Example:

- sequence #'s: 0, 1, 2, 3
- window size=3




(a)

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)




(b)

Q: what relationship between sequence # size and window size?

# Quiz Time!

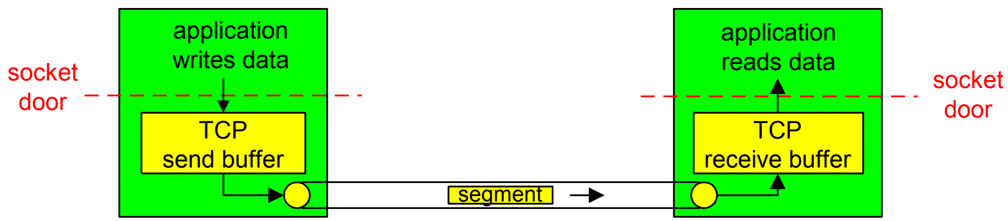
# Chapter 3 outline

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

# TCP: Overview

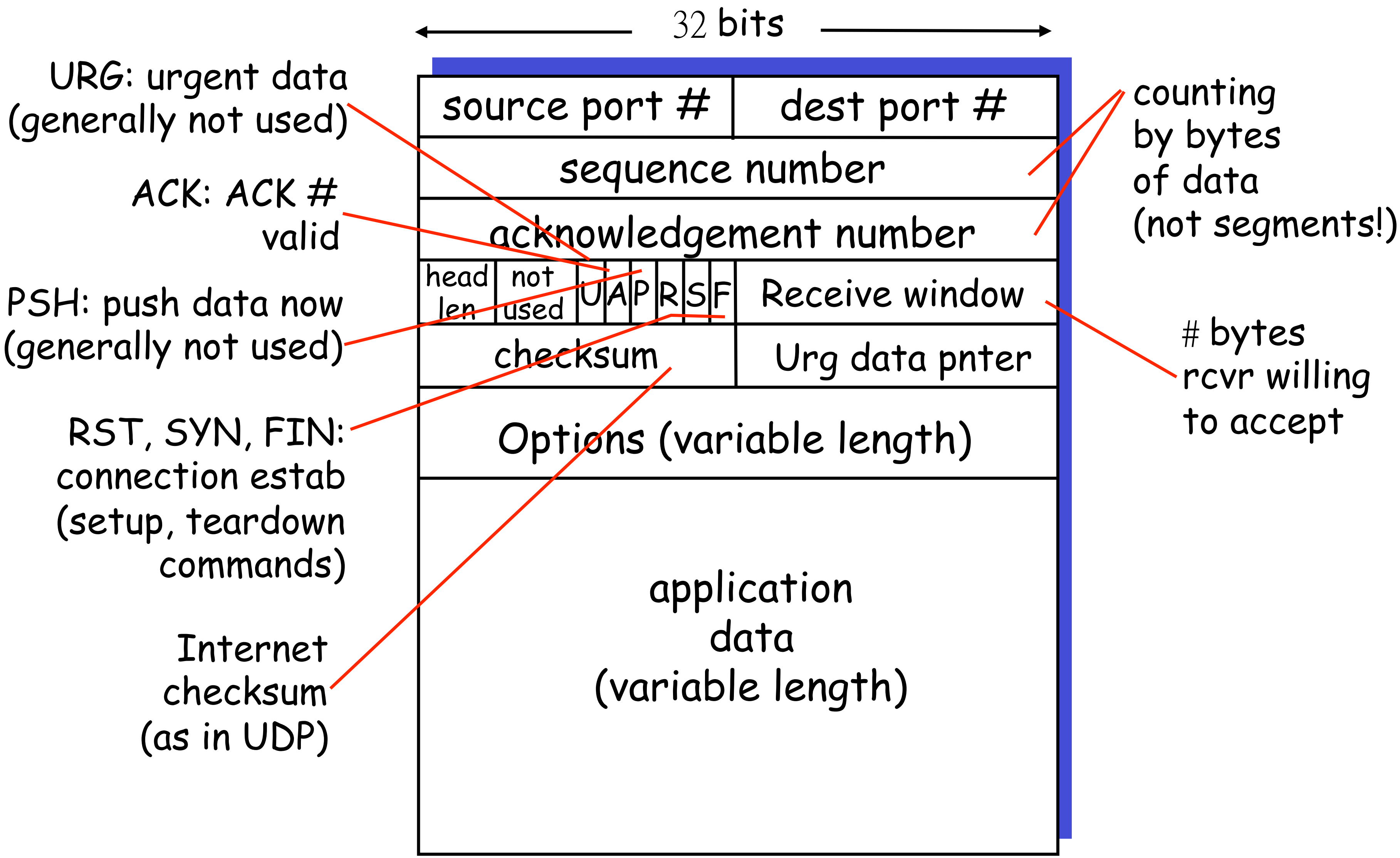
RFCs: 793, 1122, 1323, 2018, 2581

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




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

# TCP segment structure



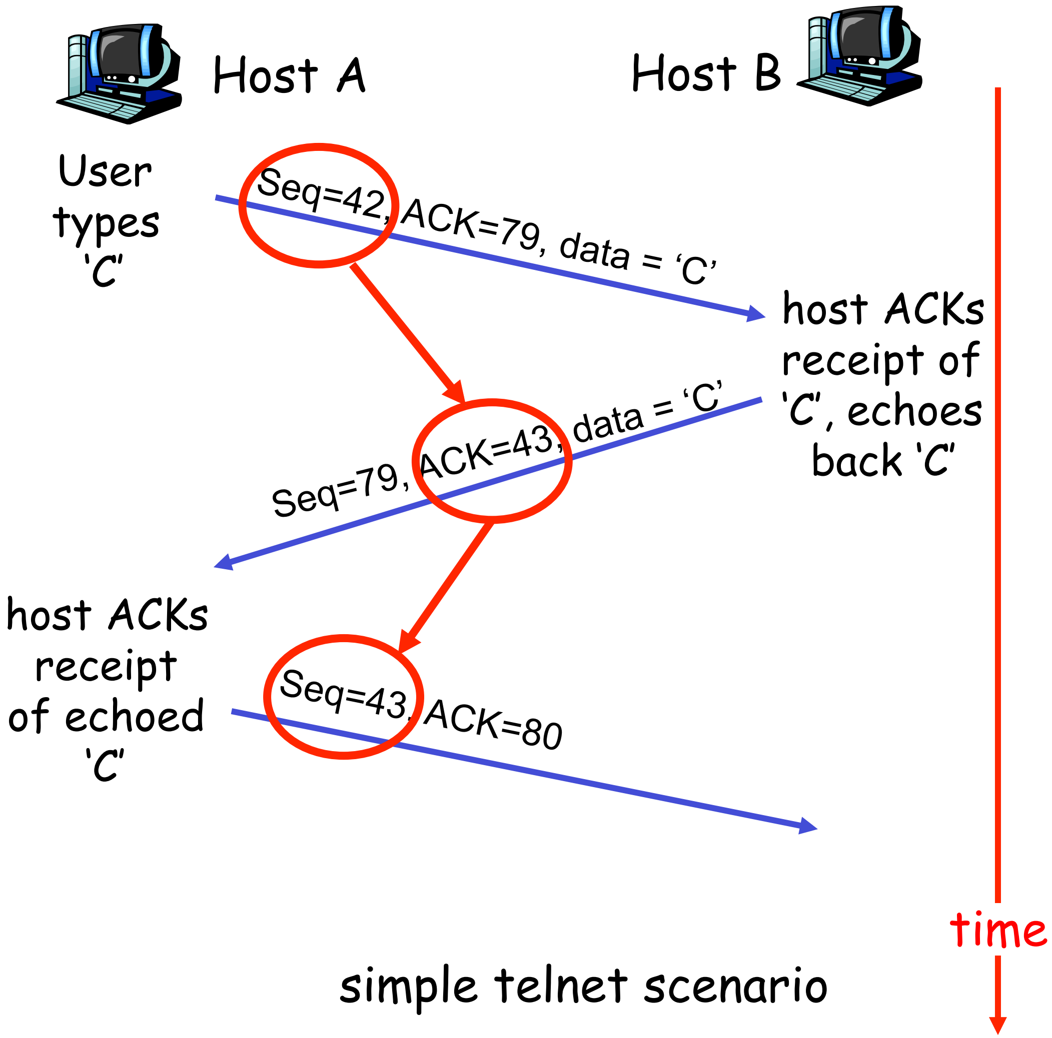
# TCP seq. #'s and ACKs

**Seq. #'s:**

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

**ACKs:**

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




simple telnet scenario

# TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- 1 sec? 1 min? Or else?
- too short?
- too long?

# TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

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

Q: how to estimate RTT?

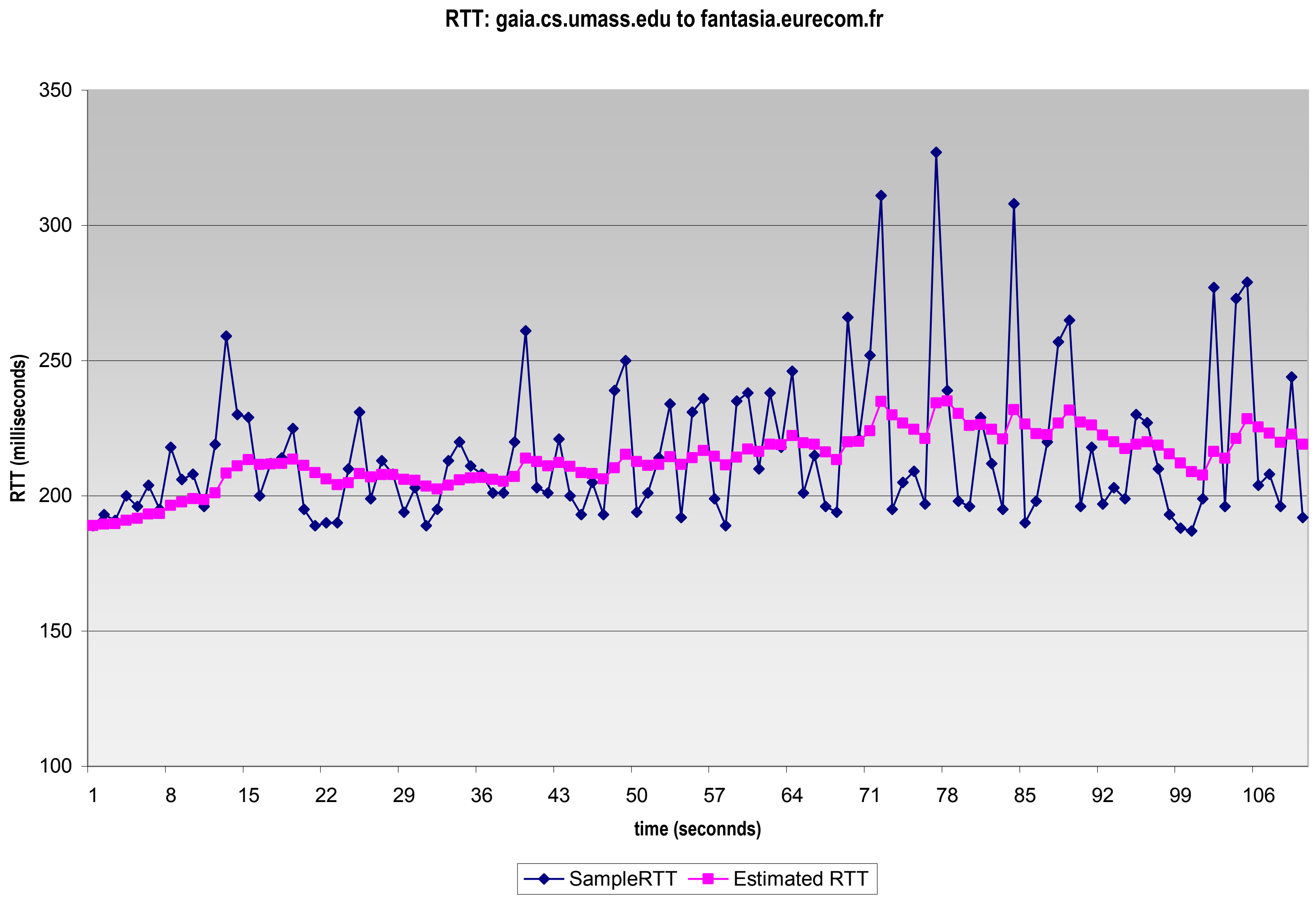
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
  - Why?
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - How?
  - average several recent measurements, not just current **`SampleRTT`**

# TCP Round Trip Time and Timeout

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

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

# Example RTT estimation:



# TCP Round Trip Time and Timeout

## Setting the timeout

- `EstimtedRTT` plus "safety margin"
  - large variation in `EstimatedRTT` -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

(typically, β = 0.25)
```

Then set timeout interval:

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

# Chapter 3 outline

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

# TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

# TCP sender events:

**data rcvd from app:**

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

**timeout:**

- retransmit segment that caused timeout
- restart timer

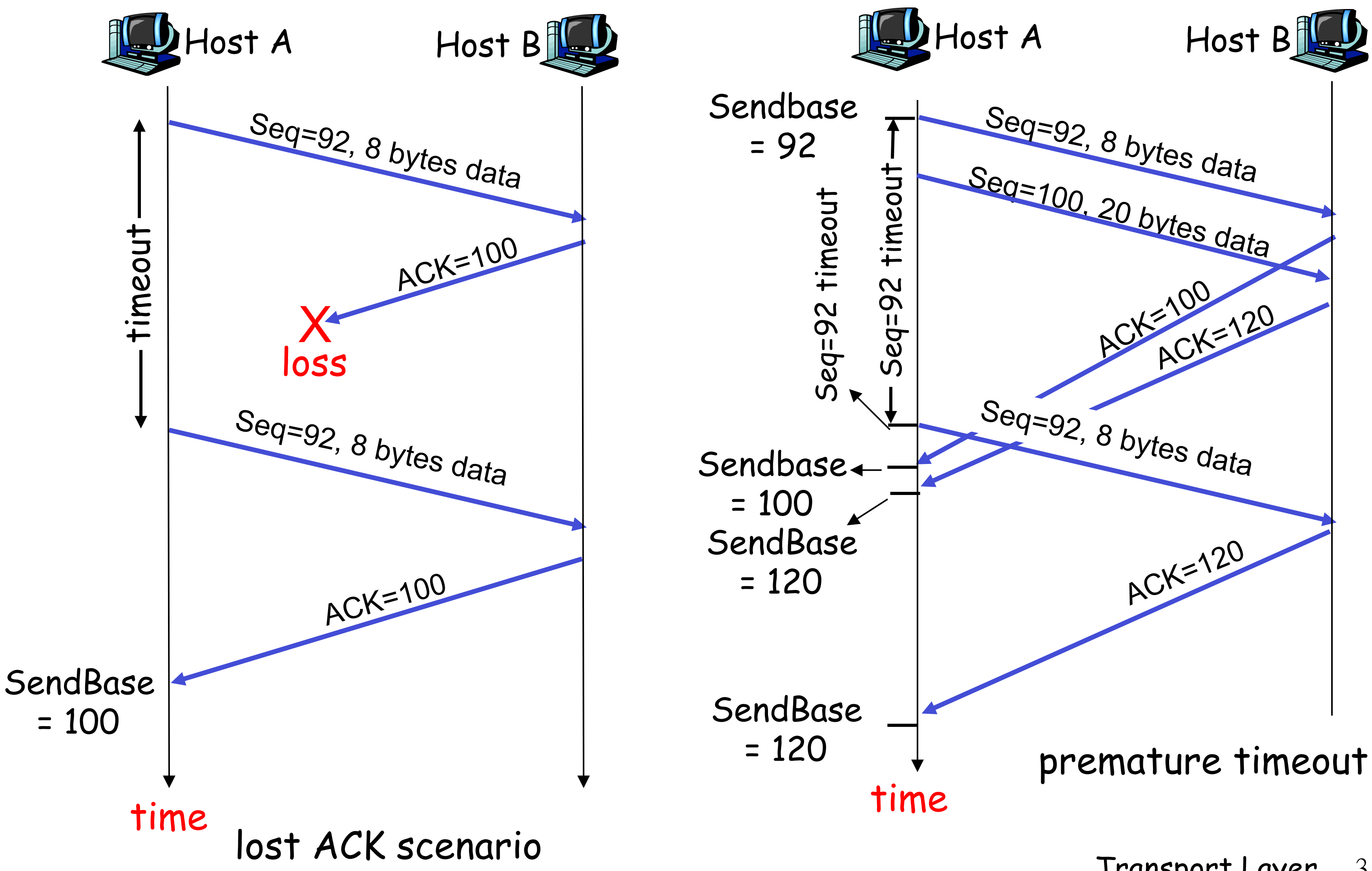
**Ack rcvd:**

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

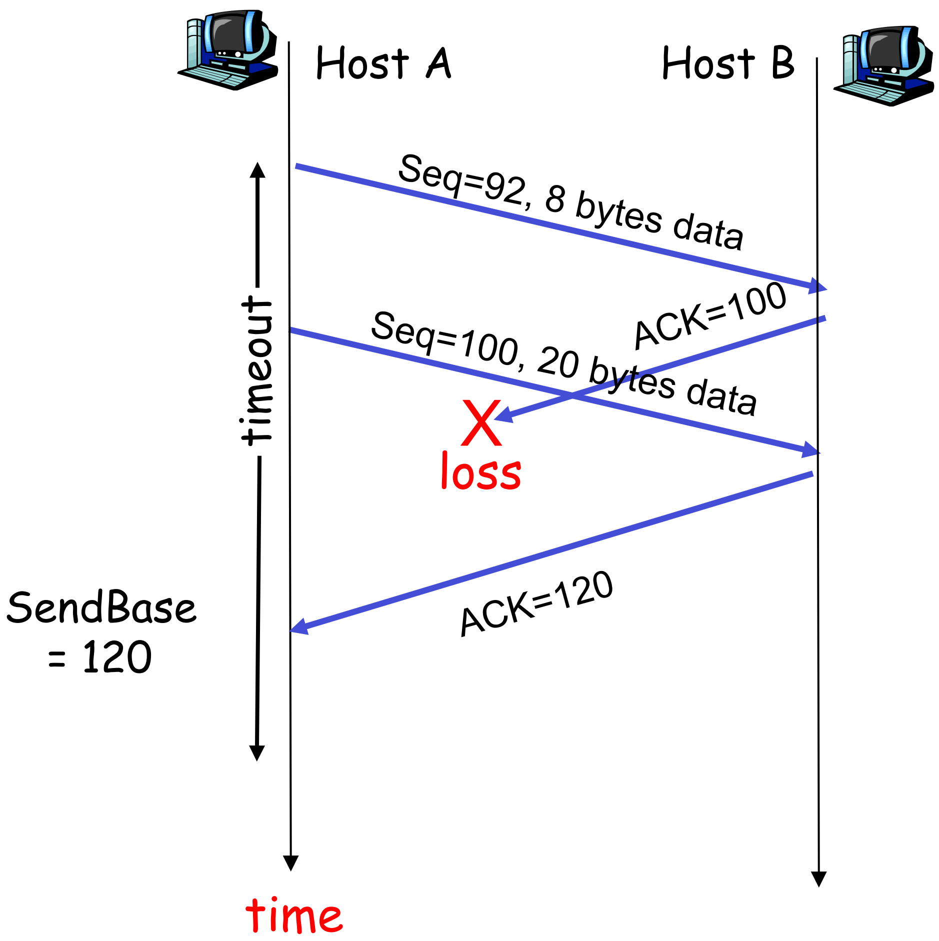
# TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

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

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

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

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

# TCP: retransmission scenarios



# TCP retransmission scenarios (more)




Cumulative ACK scenario

# 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 |

# 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.
- If sender receives 3 dup ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

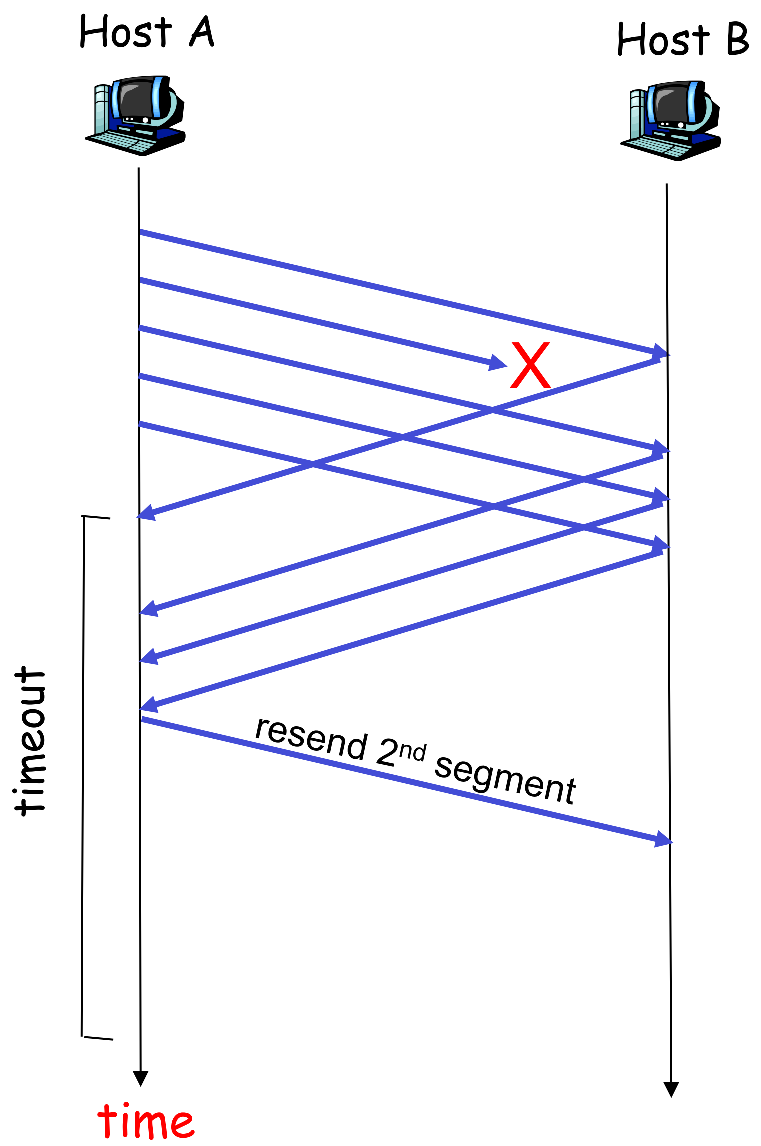



Figure 3.37 Resending a segment after triple duplicate ACK

# Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
           if (y > SendBase) {
               SendBase = y
               if (there are currently not-yet-acknowledged segments)
                    start timer
             }
           else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                     resend segment with sequence number y
                 }
```

a duplicate ACK for already ACKed segment

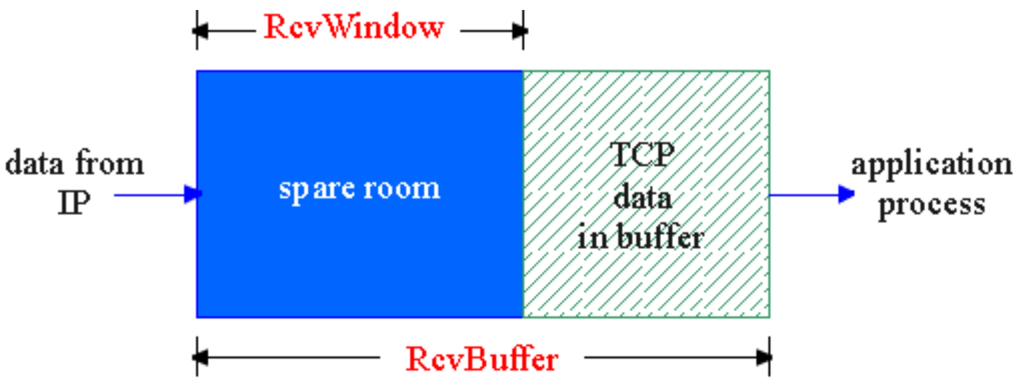
fast retransmit

# Chapter 3 outline

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

# TCP Flow Control

- receiver side of TCP connection has a receive buffer:



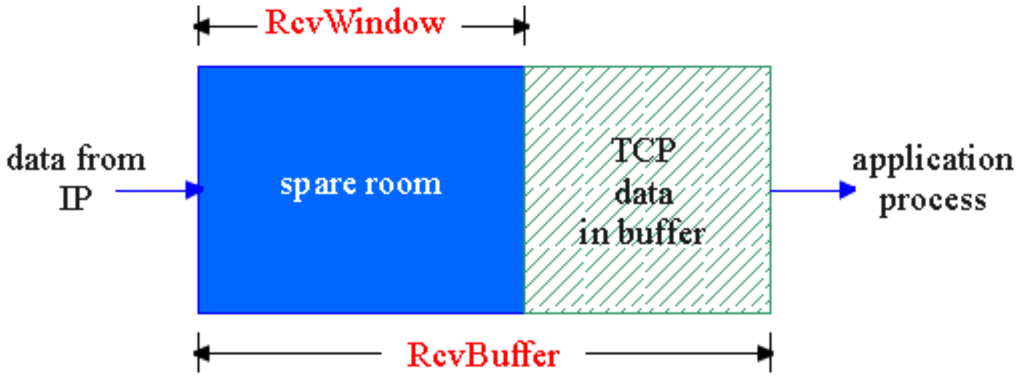

- app process may be slow at reading from buffer

**flow control**

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

- speed-matching service: matching the send rate to the receiving app's drain rate

# TCP Flow control: how it works




(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= **`RcvWindow`**

= **`RcvBuffer-[LastByteRcvd - LastByteRead]`**

- Rcvr advertises spare room by including value of **`RcvWindow`** in segments
- Sender limits unACKed data to **`RcvWindow`**
  - guarantees receive buffer doesn't overflow

# Chapter 3 outline

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

# TCP Connection Management

**Recall:** TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
- *client:* connection initiator
  `connect();`
- *server:* contacted by client
  `listen();`

## Three way handshake:

**Step 1:** client host sends TCP SYN segment to server
- specifies initial seq #
- no data

**Step 2:** server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data

# TCP Connection Management (cont.)

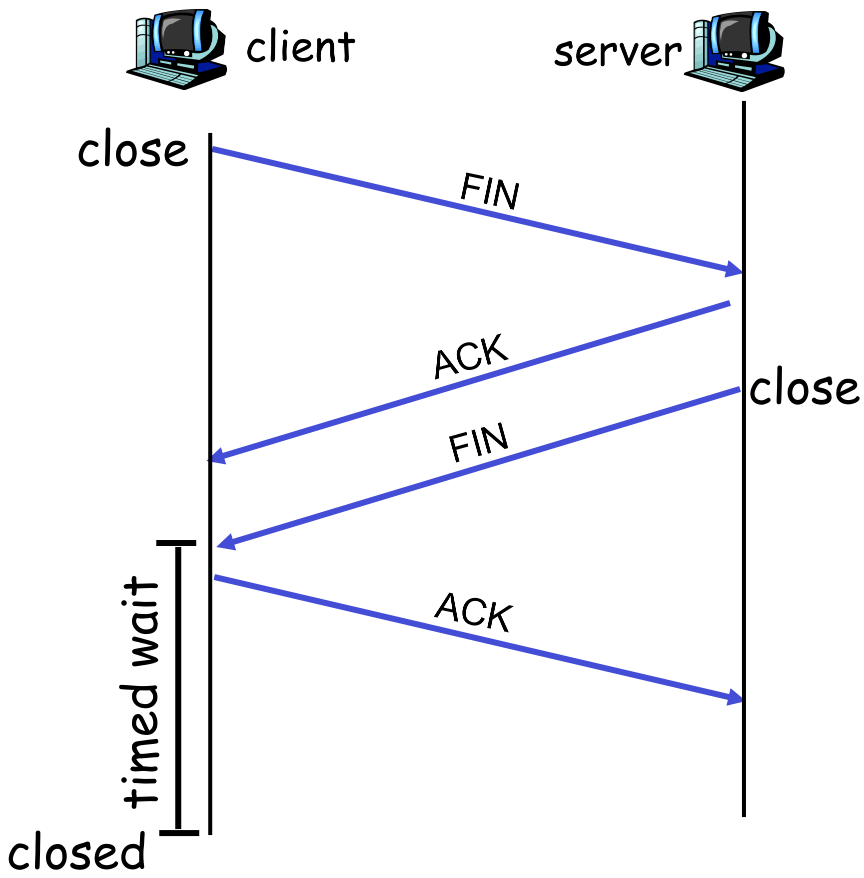
## Closing a connection:

client closes socket:

`close();`

**Step 1:** client end system sends TCP FIN control segment to server

**Step 2:** server receives FIN, replies with ACK. Closes connection, sends FIN.



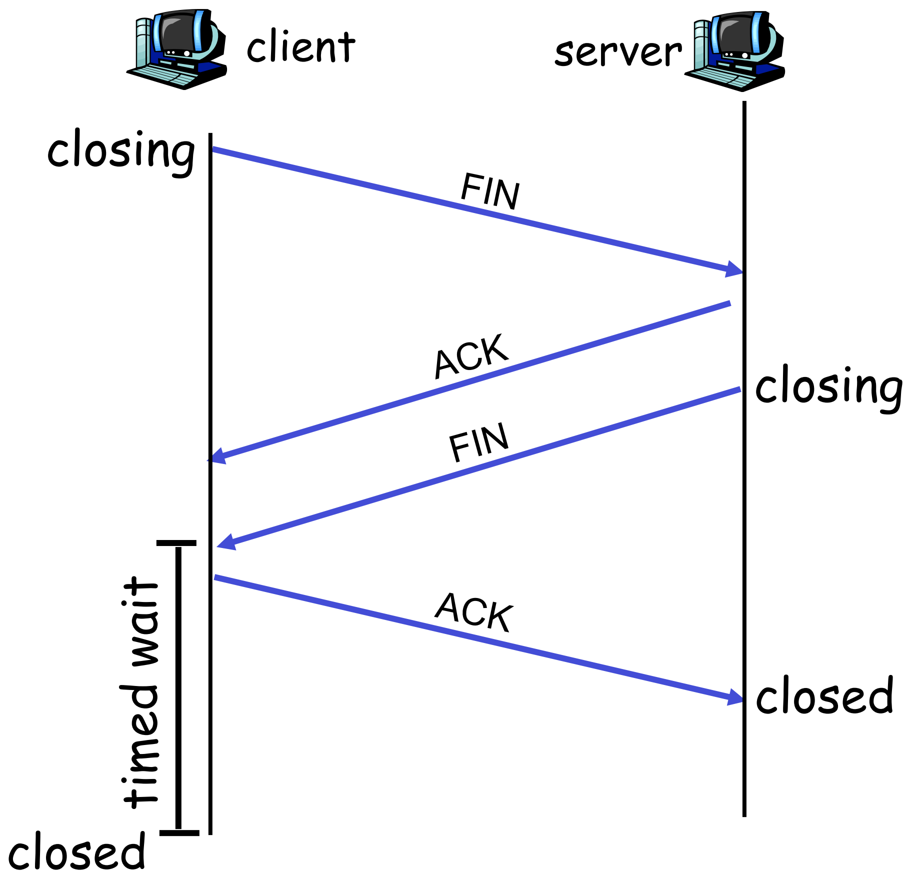
# TCP Connection Management (cont.)

**Step 3:** client receives FIN, replies with ACK.

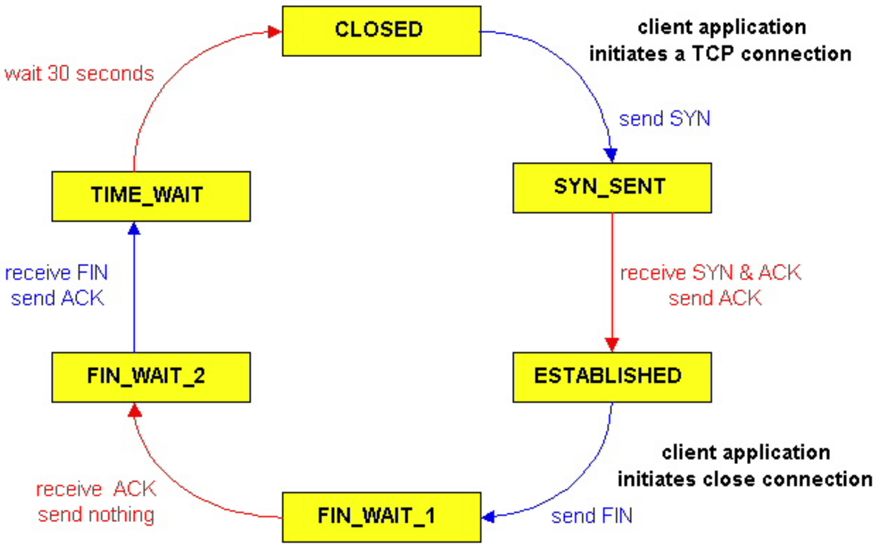
- Enters "timed wait" - will respond with ACK to received FINs

**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.

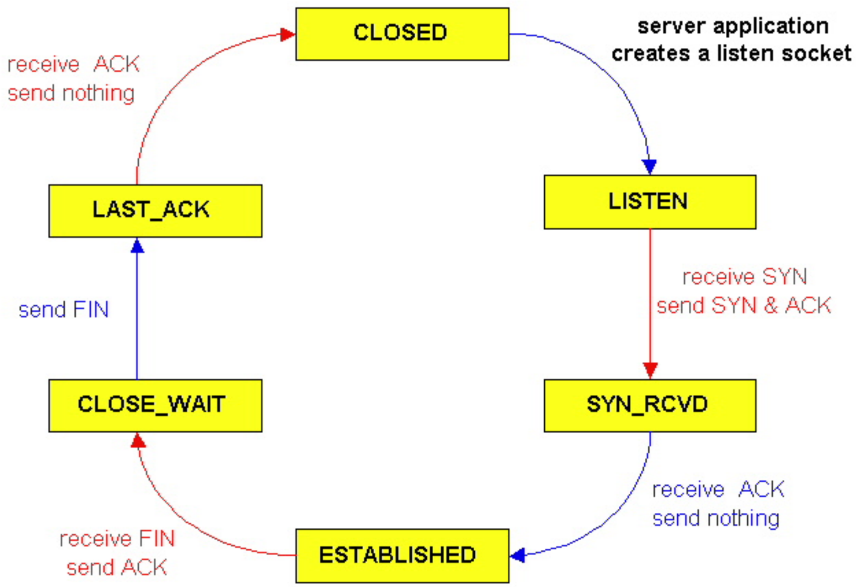


# TCP Connection Management (cont)




TCP client lifecycle

TCP server lifecycle



# Chapter 3 outline

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

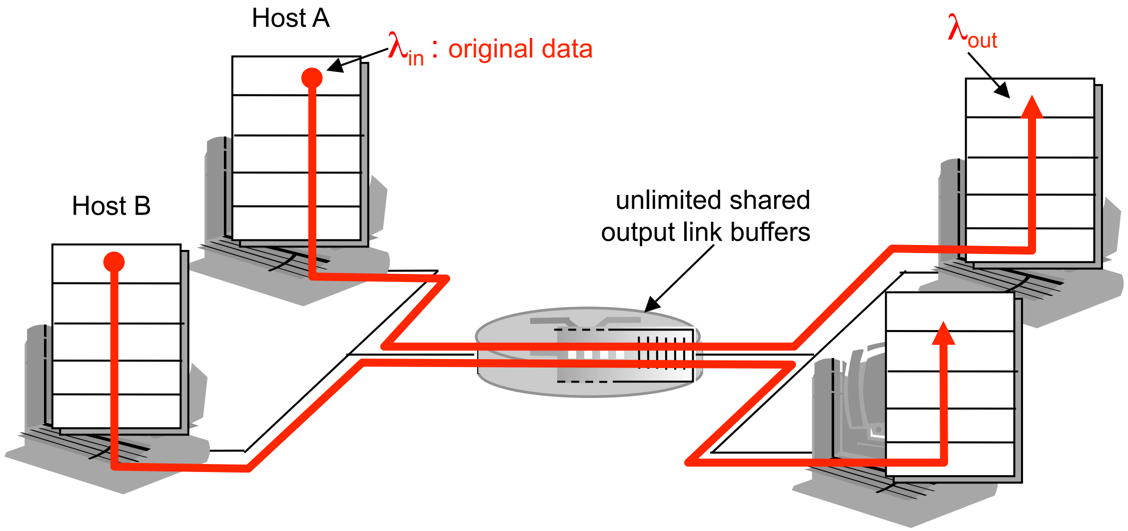
# Principles of Congestion Control

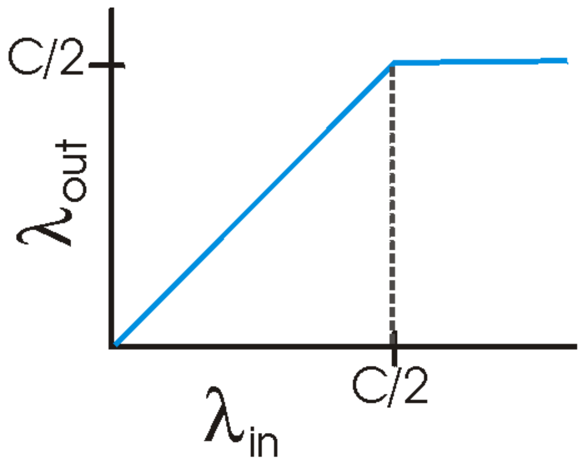
Congestion:

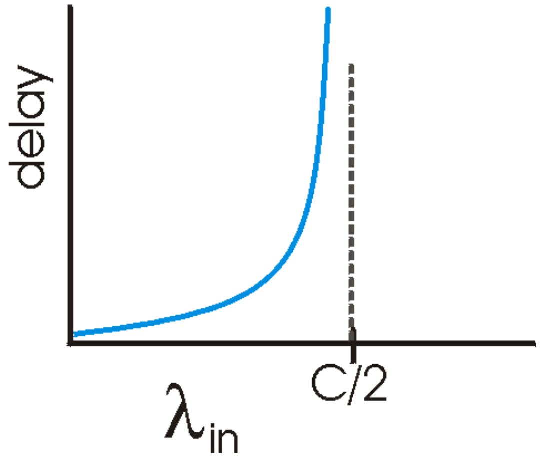
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queuing in router buffers)
- a top-10 problem!

# Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



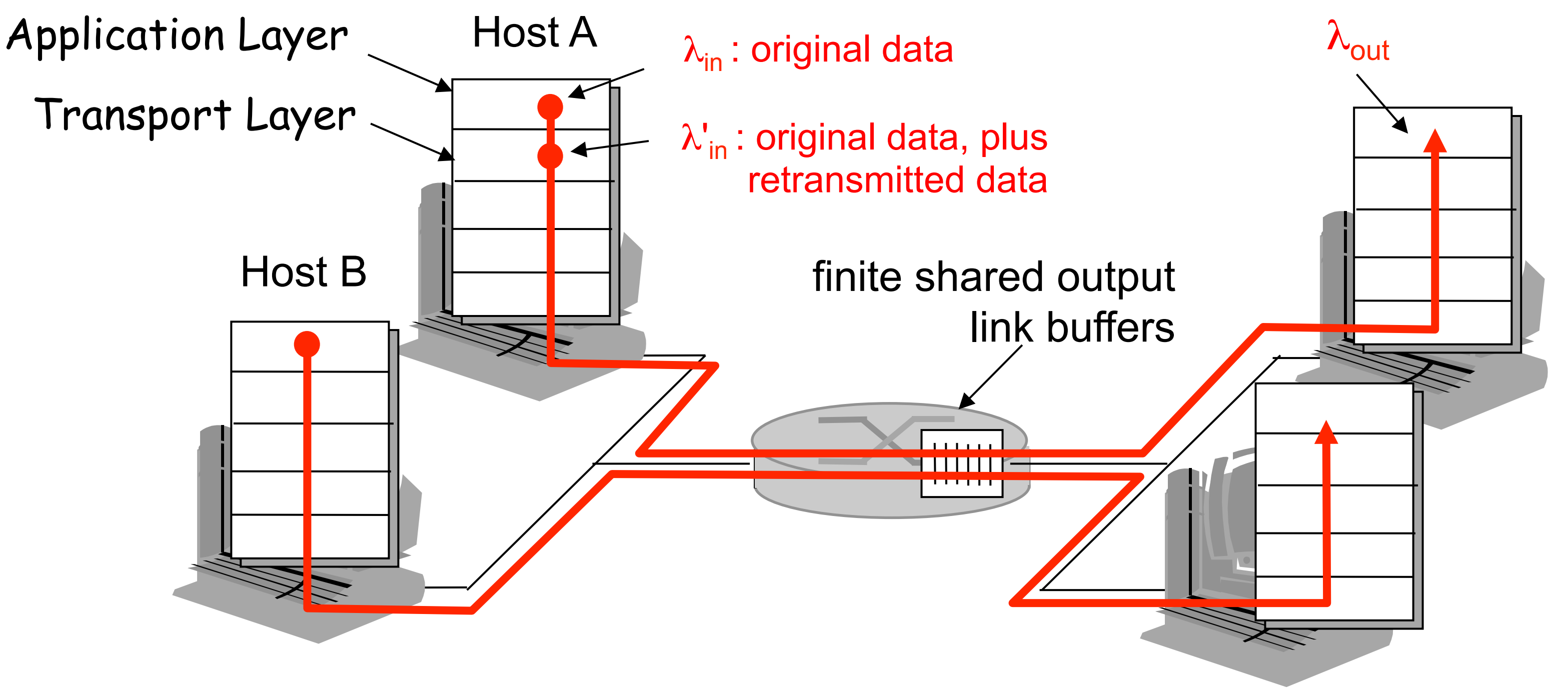




- maximum achievable throughput
- large delays when congested

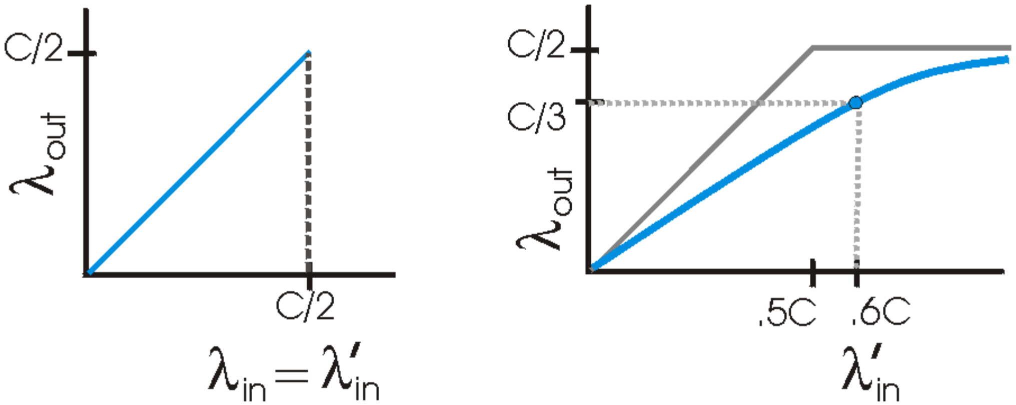
# Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet



# Causes/costs of congestion: scenario 2

- "perfect" case, always: $\lambda_{in} = \lambda_{out,}$ (goodput)
- retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of lost packet makes $\lambda'_{in}$ larger (than perfect case) for same $\lambda_{out}$




(a) (b)

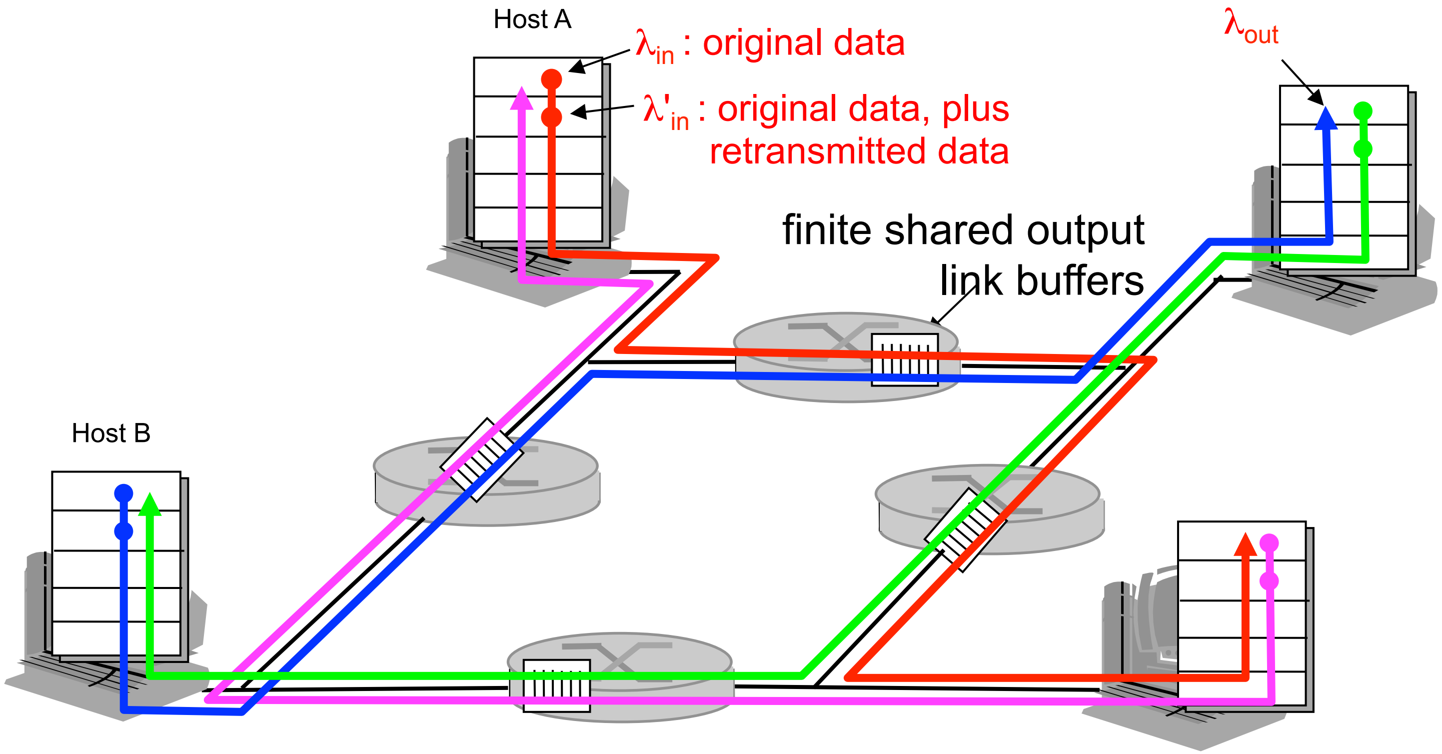
"costs" of congestion:

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

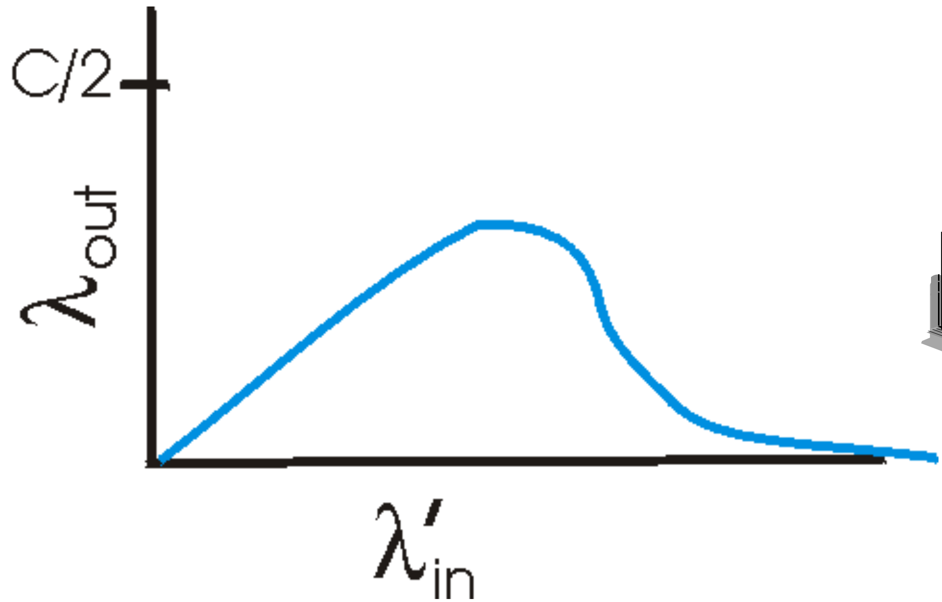
# Causes/costs of congestion: scenario 3

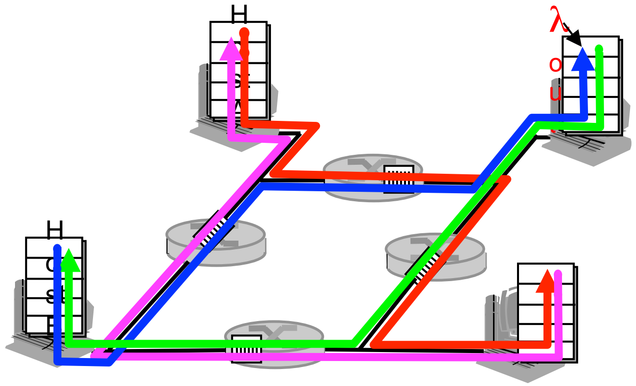
- Four senders
- Multi-hop paths
- Timeout/retransmit

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



# Causes/costs of congestion: scenario 3



Another "cost" of congestion:

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

# Message: Congestion is bad

But what can we do about it?

# Quiz Time!

# 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)
  - tells explicit rate that sender should send at

# Chapter 3 outline

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

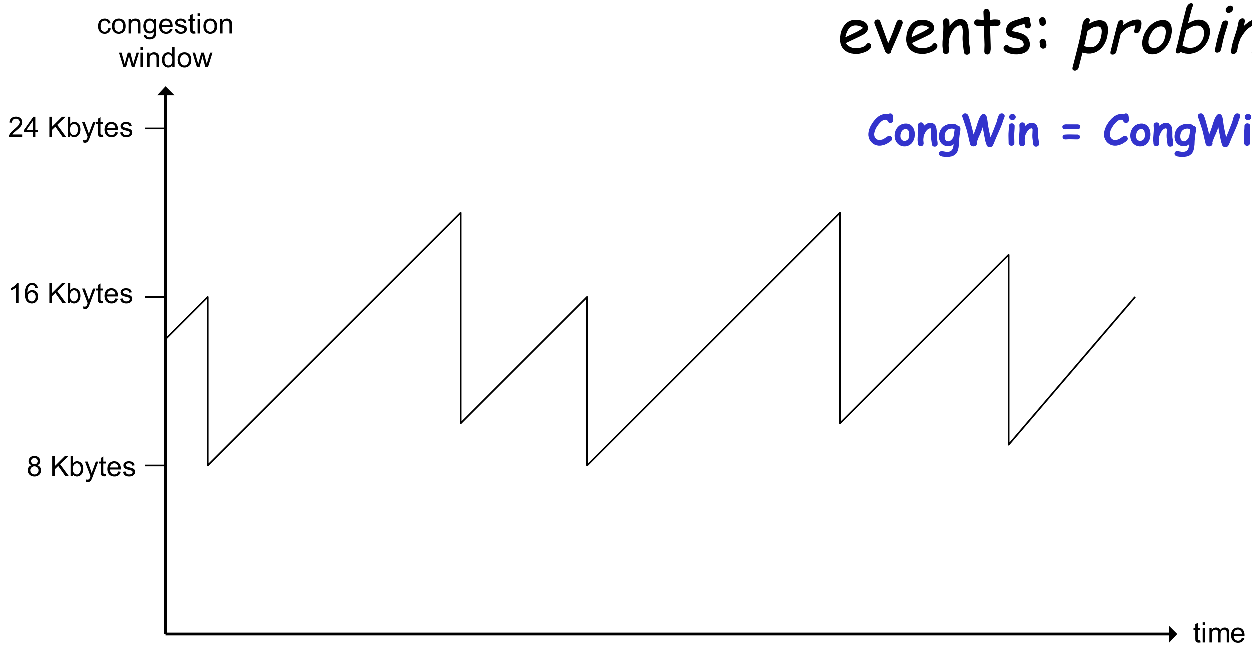
# TCP AIMD

multiplicative decrease: cut **CongWin** in half after loss event

CongWin = CongWin * 0.5

additive increase: increase **CongWin** by 1 MSS every RTT in the absence of loss events: *probing*

CongWin = CongWin + 1




Long-lived TCP connection

# TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:

  **LastByteSent-LastByteAcked ≤ CongWin**

- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **CongWin** is dynamic, a function of perceived network congestion

How does sender perceive congestion?

- loss event
- How to tell whether there's a loss event?
- TCP sender reduces rate (**CongWin**) after loss event

three mechanisms:

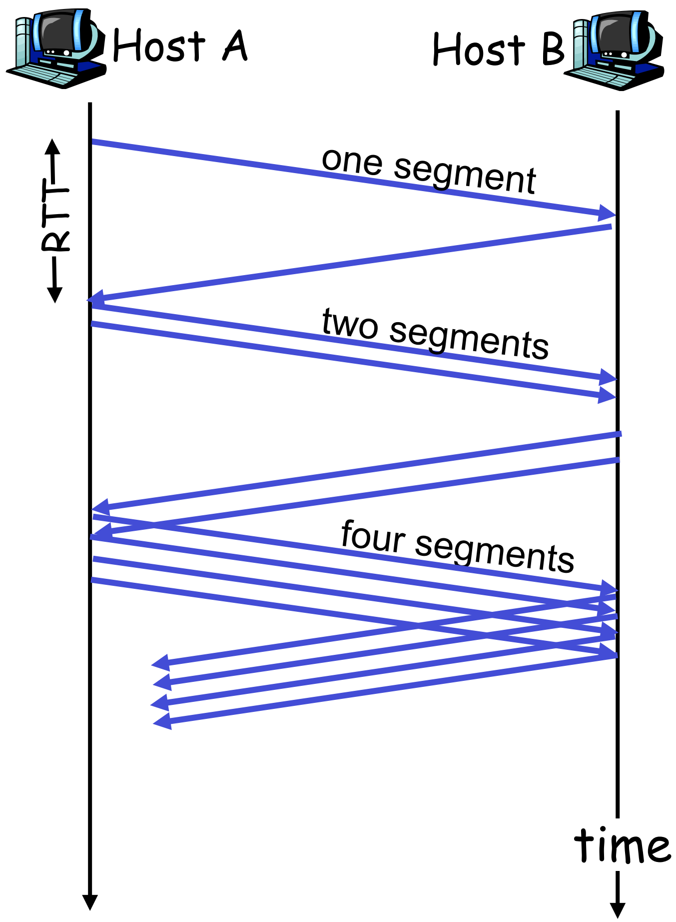
- AIMD
- slow start
- conservative after timeout events

# TCP Slow Start

- When connection begins, `CongWin` = 1 MSS
  - Example:
  - MSS = 500 bytes
  - RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

# TCP Slow Start (more)

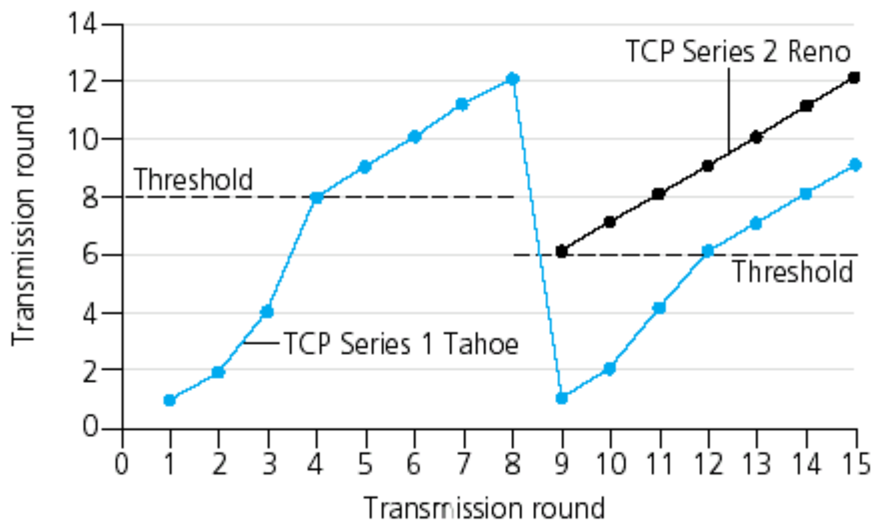
- When connection begins, increase rate exponentially:
  - double `CongWin` every RTT
  - done by incrementing `CongWin` for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



# Refinement

Q: When should the exponential increase switch to linear?

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




## Implementation:

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event

# Refinement

- After 3 dup ACKs:
  - `CongWin` is cut in half
  - window then grows linearly
- <u>But</u> after timeout event:
  - `CongWin` instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

**Philosophy:**

Why Half the CongWin vs. 1?

# Refinement

- After 3 dup ACKs:
  - `CongWin` is cut in half
  - window then grows linearly
- But after timeout event:
  - `CongWin` instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

**Philosophy:**

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

## Quiz Time!

# Summary: TCP Congestion Control

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

# TCP sender congestion control

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

# TCP throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT

# TCP Futures: TCP over "long, fat pipes"

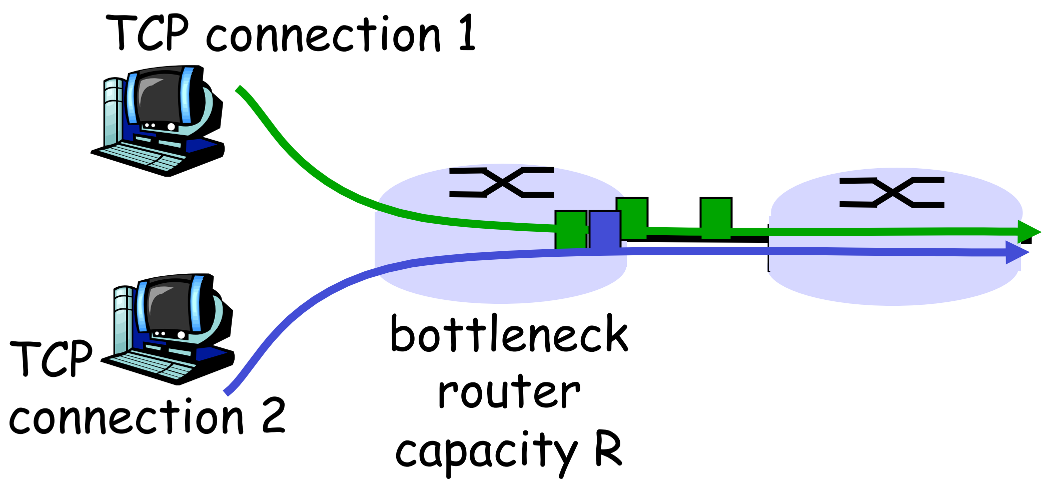
- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ $L = 2 \cdot 10^{-10}$ *Wow*
- New versions of TCP for high-speed

# TCP Fairness

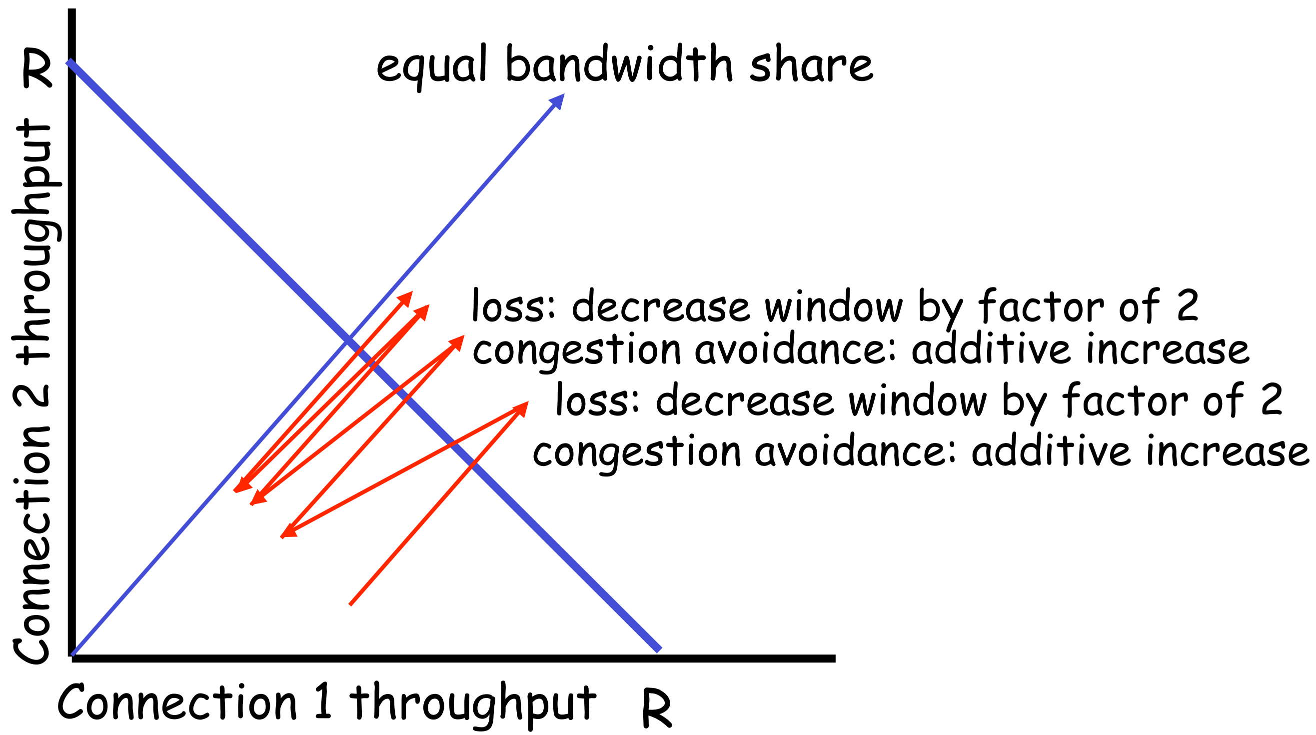
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



# 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:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

## Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: 9 users, link of rate R supporting 9 TCP connections;
  - new app/user asks for 1 TCP, gets rate R/10
  - new app/user asks for 9 TCPs, gets R/2 !

# Chapter 3: Summary

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

## Next:

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