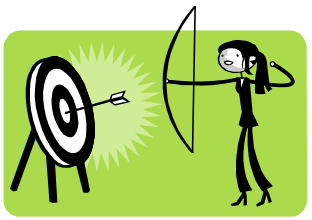
Chapter 6: Transport Layer



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

Keypoints and Difficulties

Keypoints:

- Port numbers
- TCP

Difficulties:

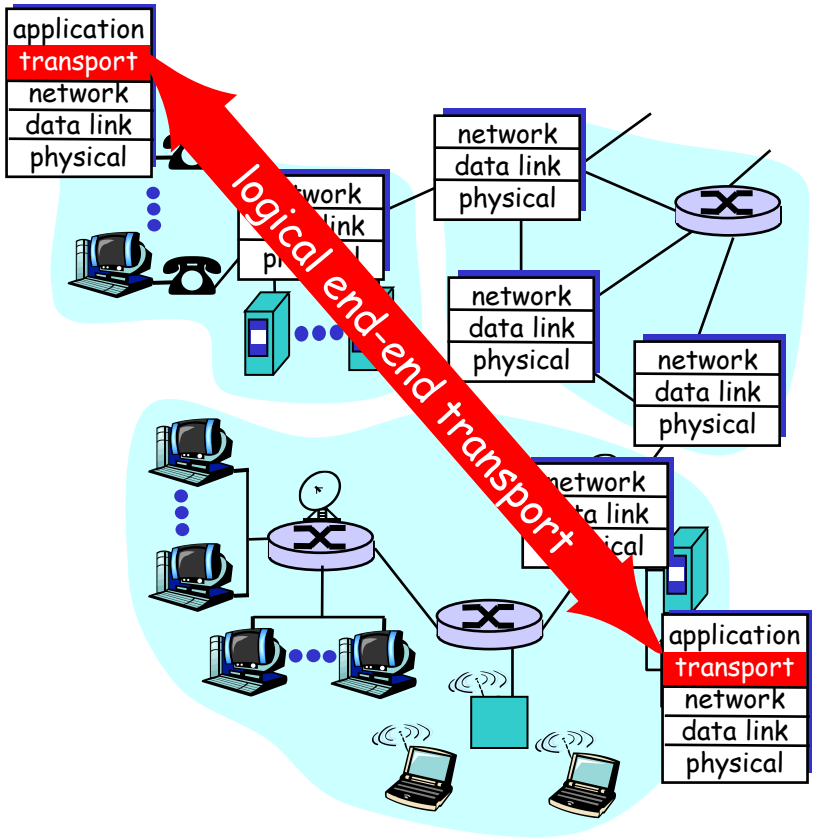
- TCP connection management
- TCP Reliable data transfer
- TCP congestion control

Chapter 6 Outline

- 6.1 Transport-layer services
- 6.2 Multiplexing and demultiplexing
- 6.3 Connectionless transport: UDP
- 6.4 Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Reliable data transfer
- 6.5 TCP congestion control

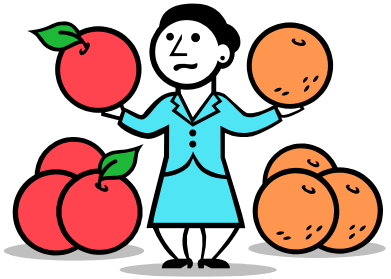
Transport Services and Protocols

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



Transport vs. Network Layer

- *Network layer:*
  - Logical communication between hosts



- *Transport layer:*
  - Logical communication between processes
  - Relies on, enhances, network layer services

Transport vs. Network Layer

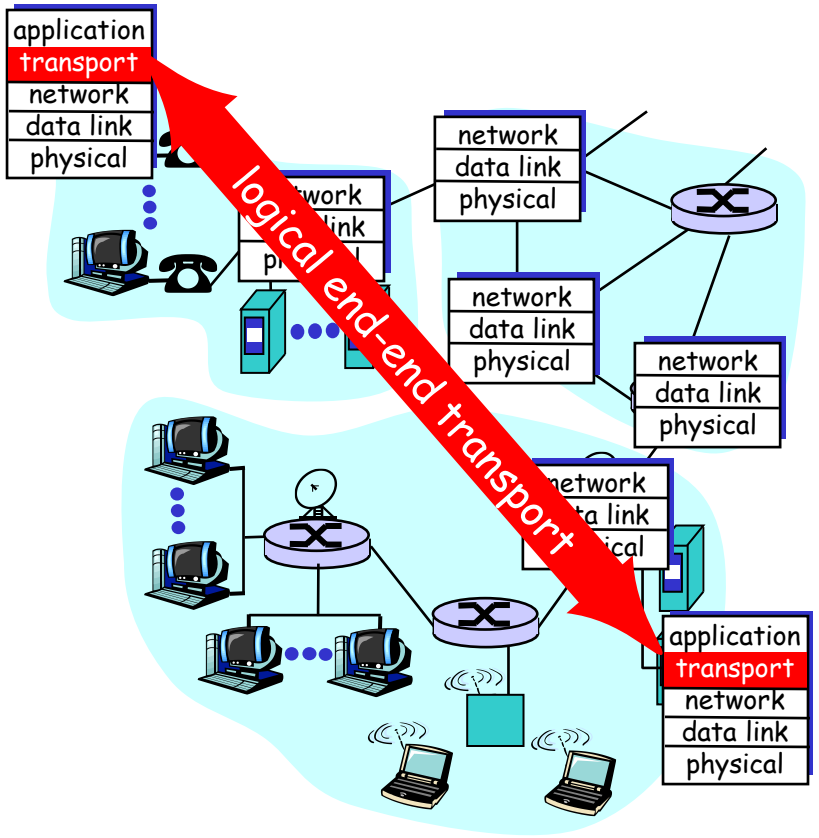
Household analogy:

12 kids sending letters to 12 kids

- Processes = kids
- App messages = letters in envelopes
- Hosts = houses
- Network-layer protocol = postal service
- Transport protocol = Ann and Bill

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 6 Outline

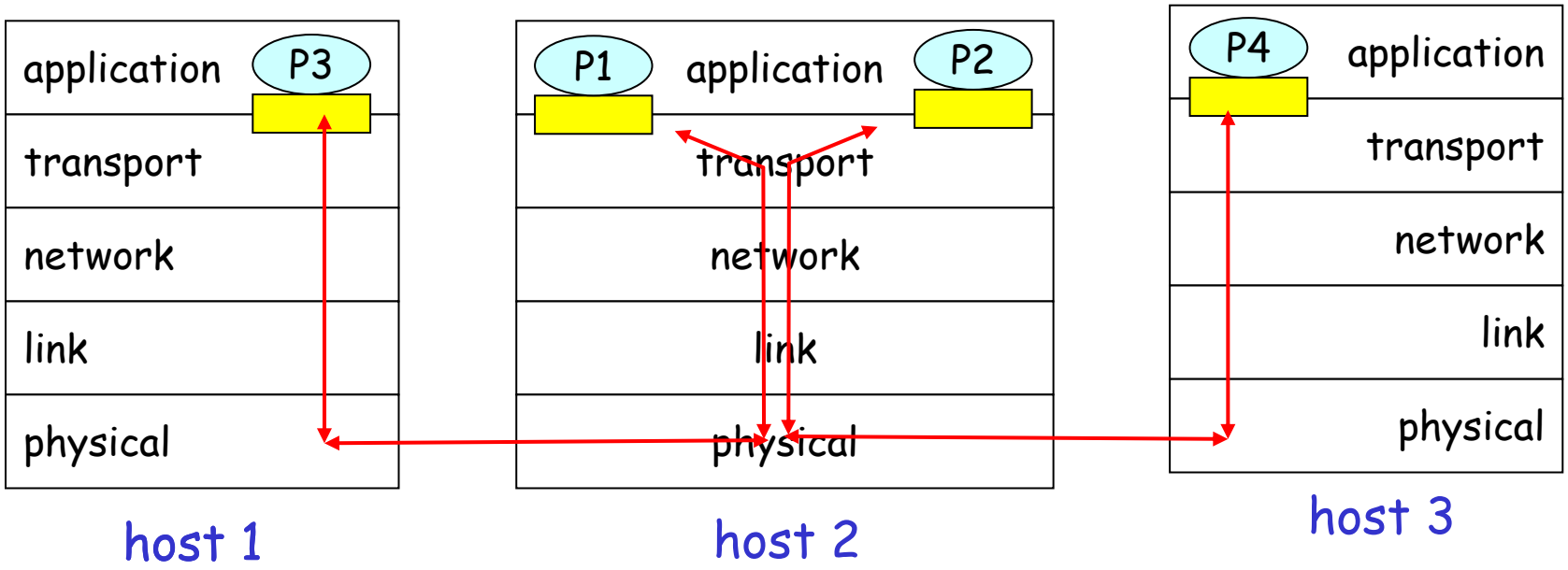
- 6.1 Transport-layer services
- 6.2 Multiplexing and demultiplexing
- 6.3 Connectionless transport: UDP
- 6.4 Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Reliable data transfer
- 6.5 TCP congestion control

Multiplexing/Demultiplexing

- 1 host ←→ 1 or more processes
- 1 process ←→ 1 or more sockets
- Transport layer interacts with socket

- Demultiplexing at rcv host
- Multiplexing at send host

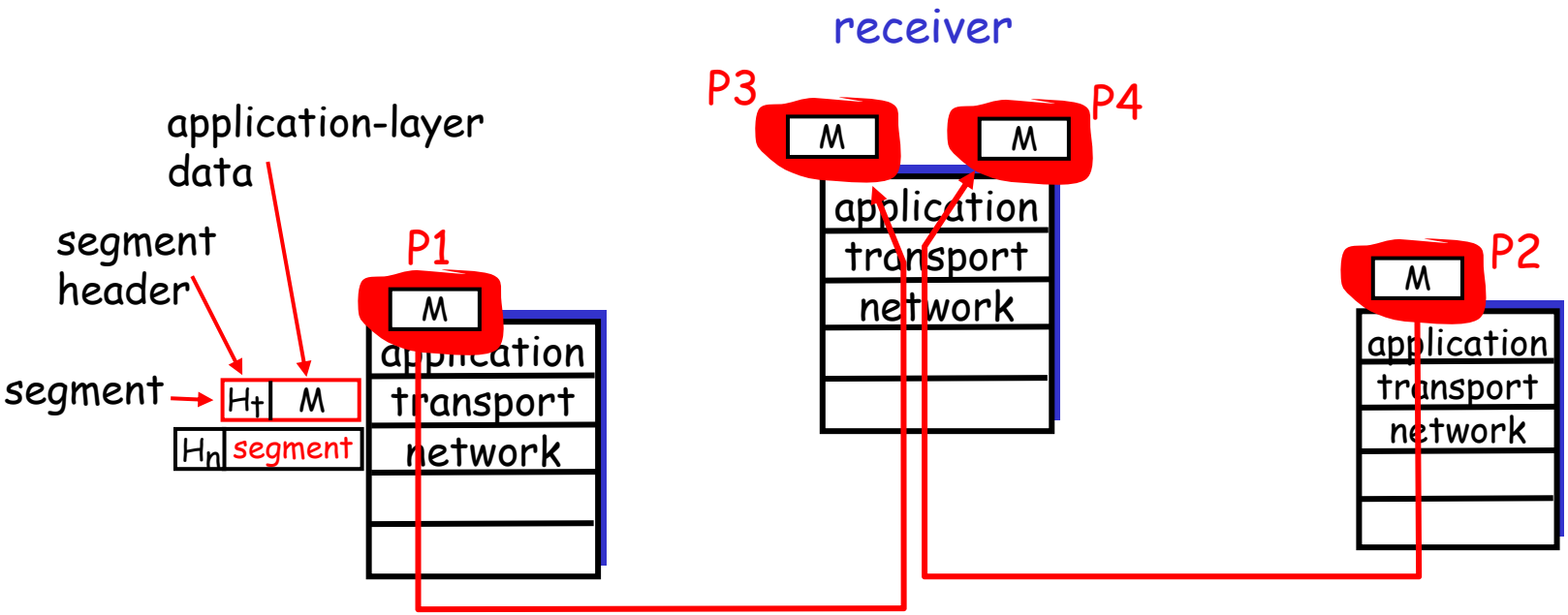
= socket = process



Multiplexing/Demultiplexing

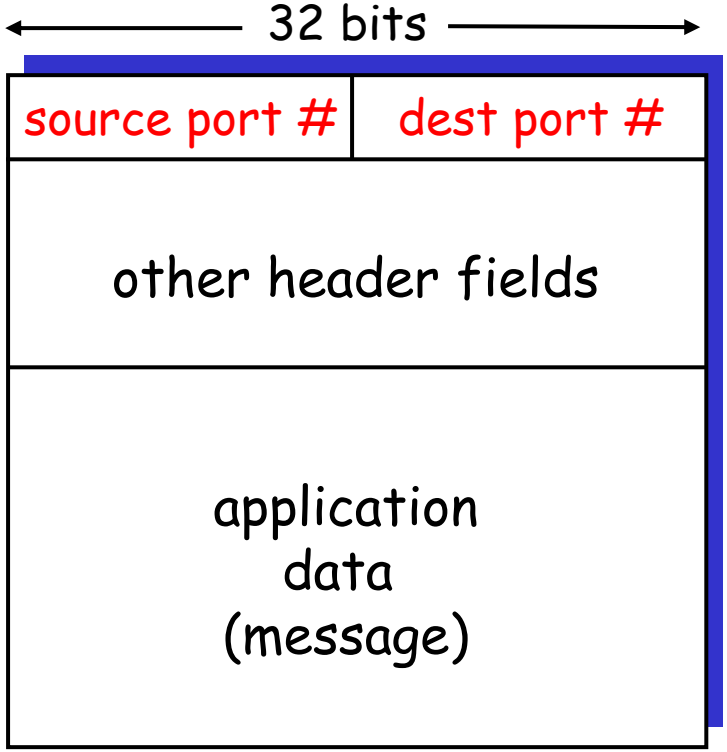
Segment - unit of data exchanged between transport layer entities
- aka TPDU: transport protocol data unit

Demultiplexing at rcv host: : delivering received segments to correct app layer processes through socket



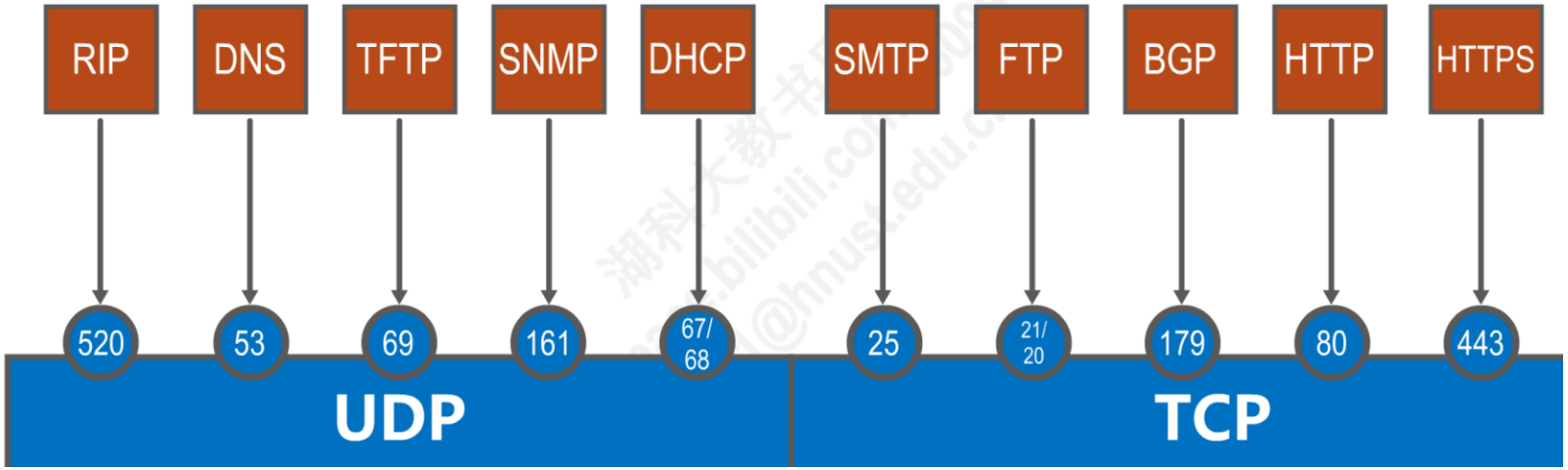
How Demultiplexing Works

- Host receives IP datagrams
  - Each datagram has source IP address, destination IP address
  - Each datagram carries 1 transport-layer segment
  - Each 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

Well-known port number



The port number only has local meaning, and the same port number in different computers is not related.

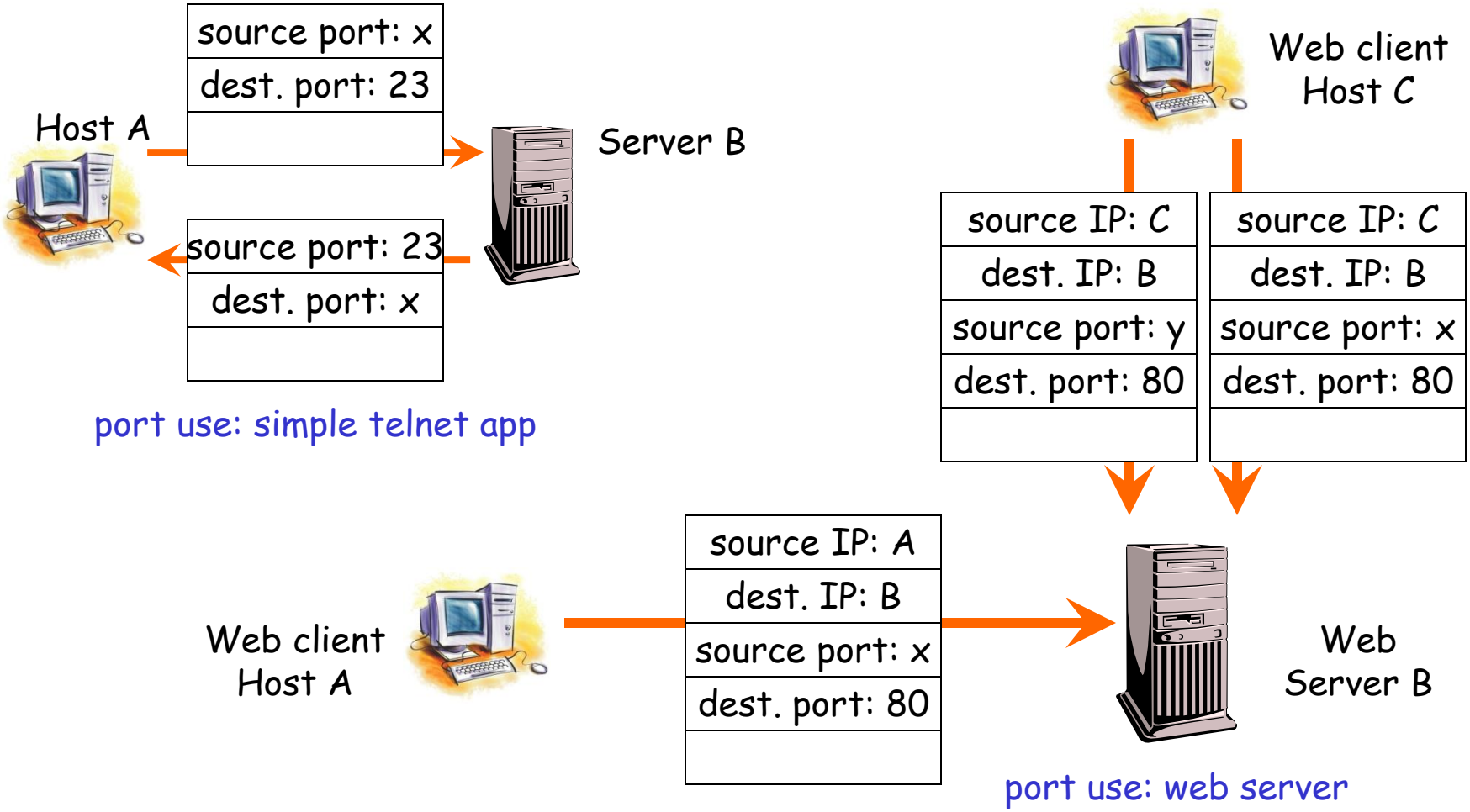
Multiplexing/Demultiplexing

Multiplexing at send host:

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

- Multiplexing/demultiplexing
  - Based on sender, receiver port numbers, IP addresses
  - <source IP #, dest. IP#, source port #, dest. port#>

Multiplexing/Demultiplexing: examples



Connectionless Demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 =
   new DatagramSocket(12534);
DatagramSocket mySocket2 =
   new DatagramSocket(12535);
```

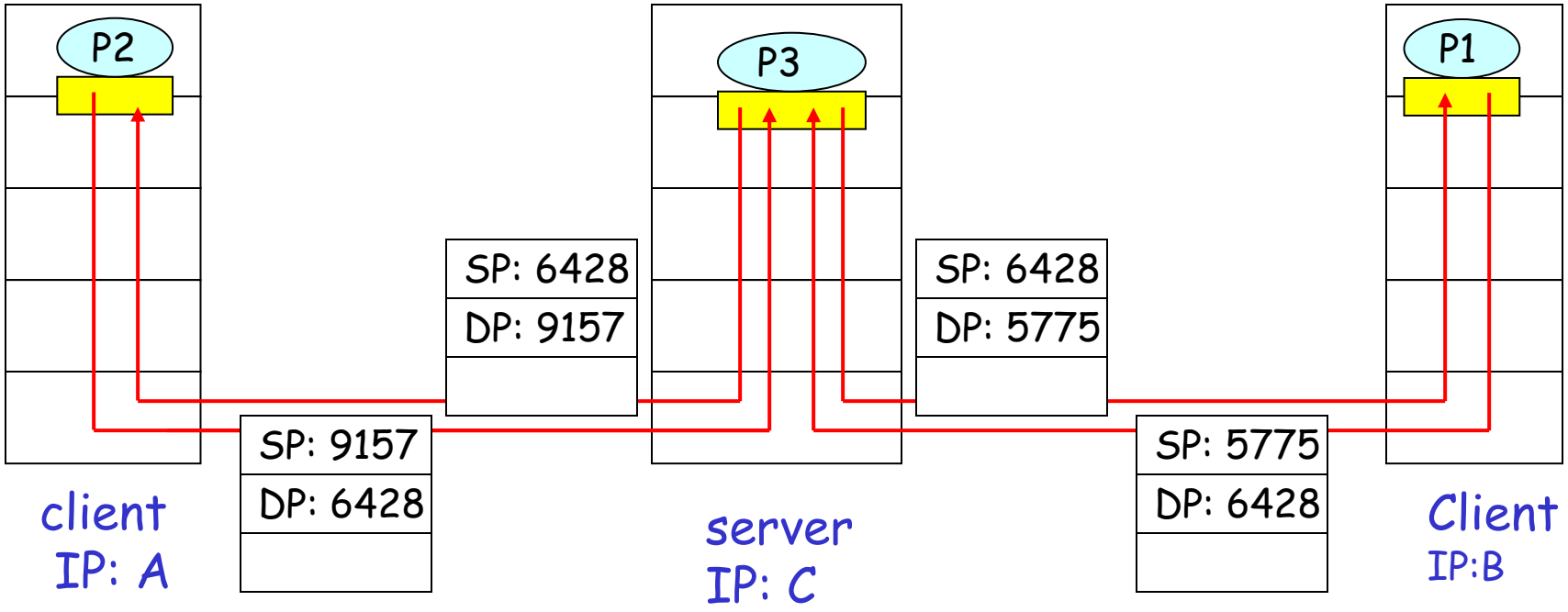
- UDP socket identified by two-tuple:

(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)

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

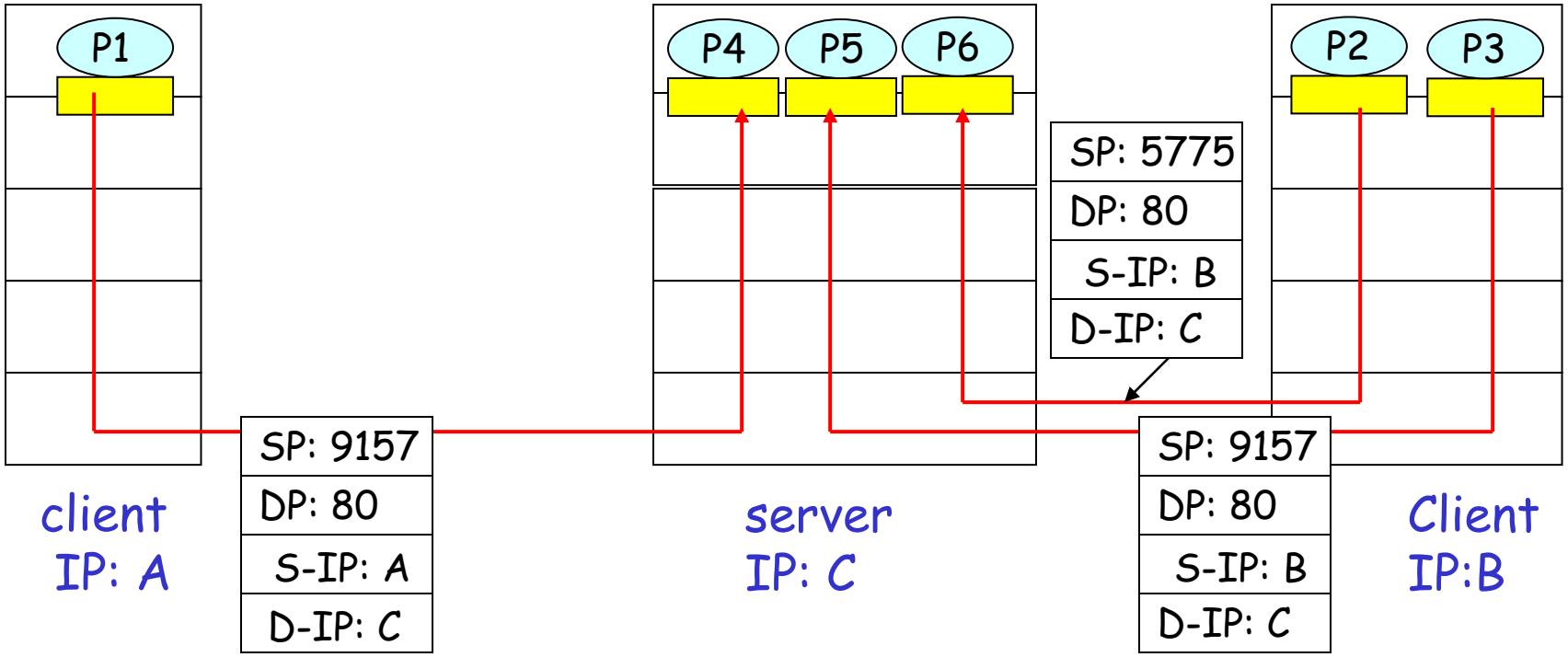


SP provides "return address"

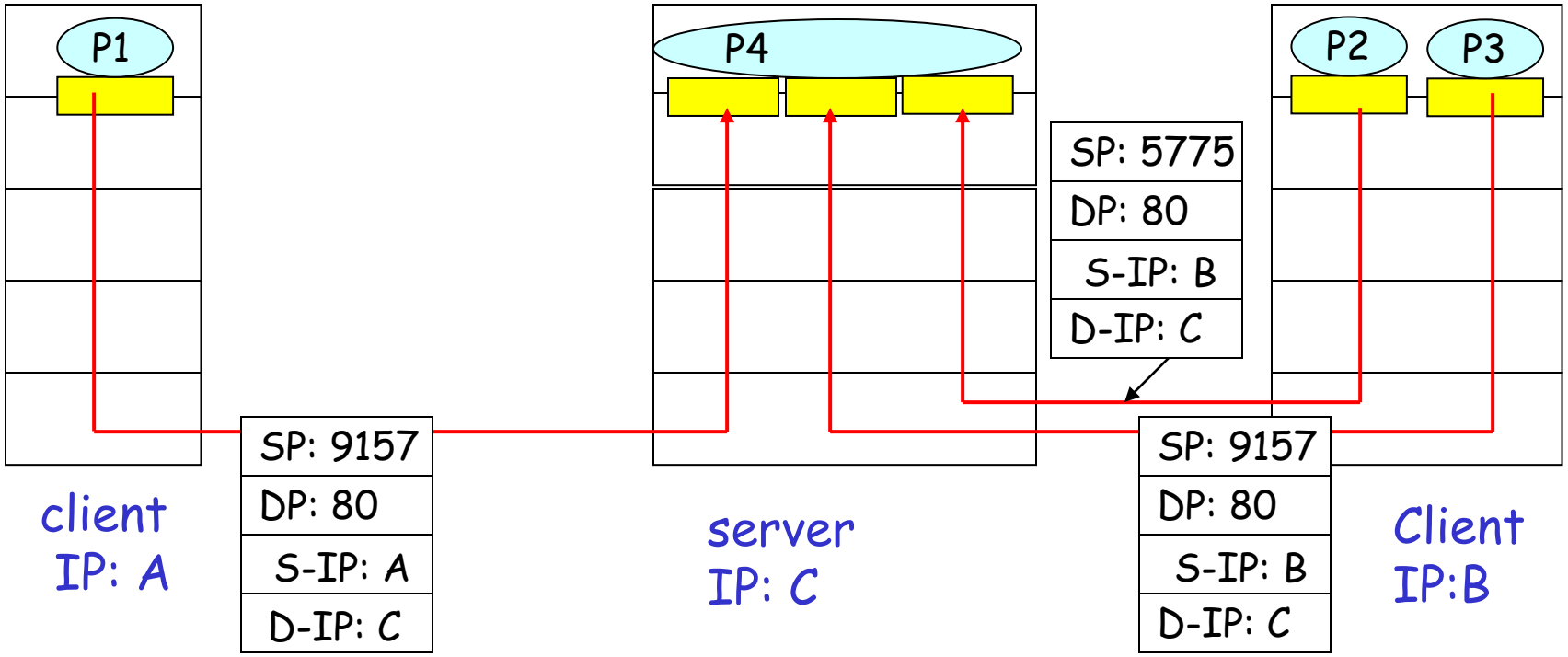
Connection-Oriented Demux

- 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 6 Outline

- 6.1 Transport-layer services
- 6.2 Multiplexing and demultiplexing
- 6.3 Connectionless transport: UDP
- 6.4 Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Reliable data transfer
- 6.5 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

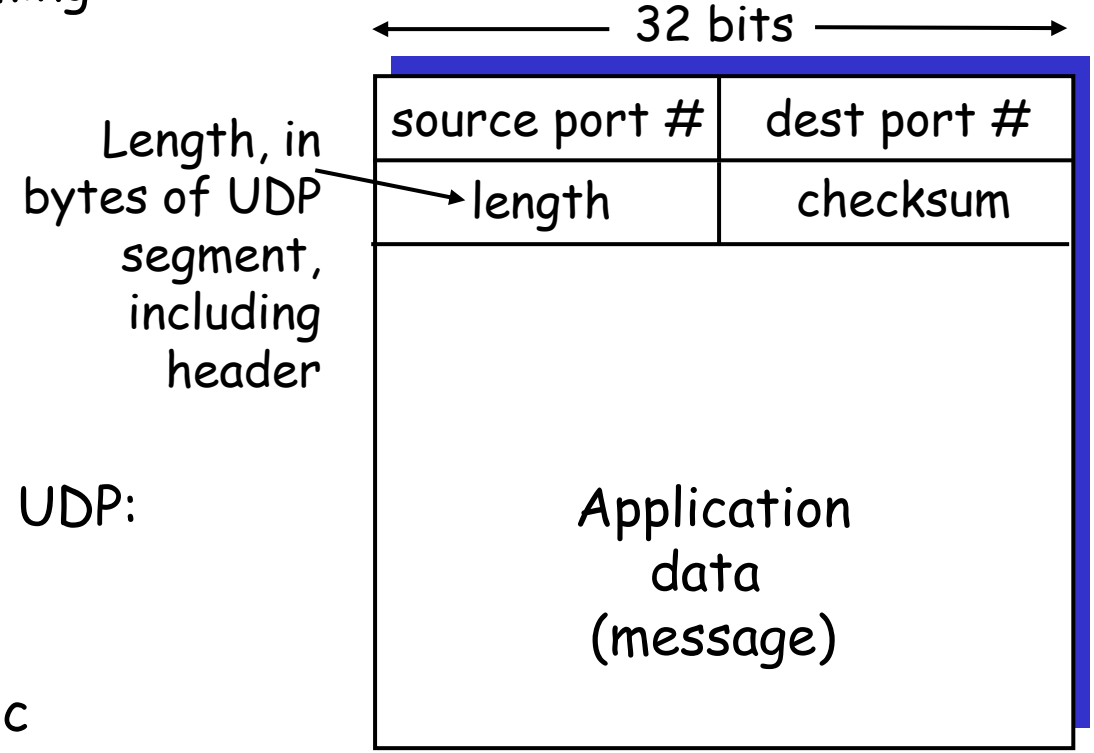
- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost
  - Delivered out of order to app
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

- 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

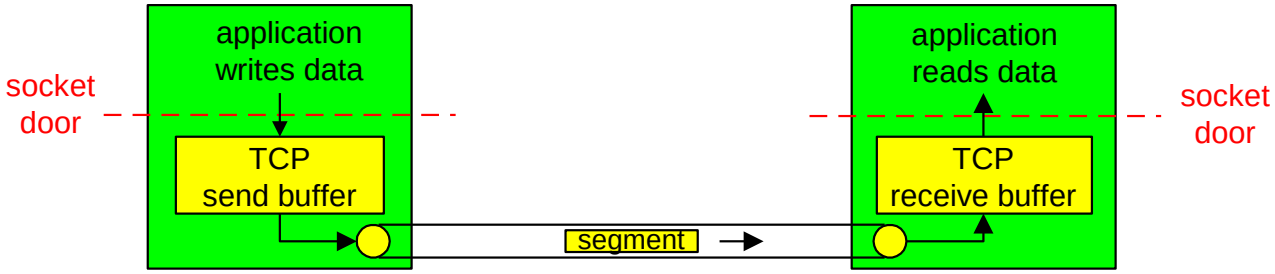
Chapter 6 Outline

- 6.1 Transport-layer services
- 6.2 Multiplexing and demultiplexing
- 6.3 Connectionless transport: UDP
- 6.4 Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Reliable data transfer
- 6.5 TCP congestion control

TCP: Overview

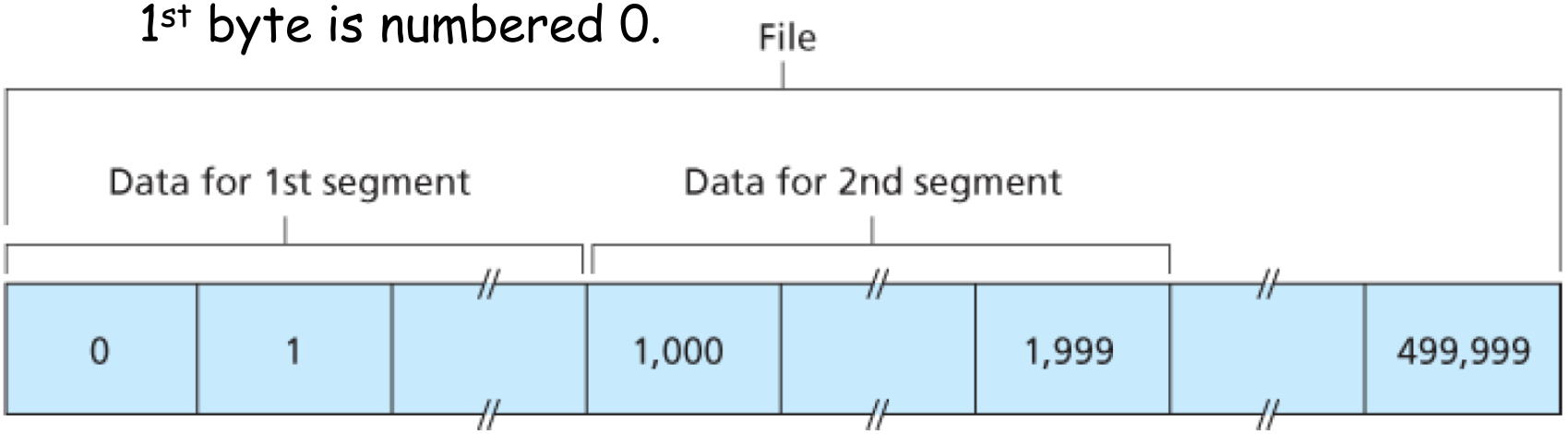
RFCs: 793, 1122, 1323, 2018, 2581

- End-to-End:
  - One sender, one receiver
- In-order *byte steam*:
  - No "message boundaries"
- Reliable :
  - TCP congestion and flow control set window size
- Connection-oriented:
  - Handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- Full duplex data:
  - Bi-directional data flow in same connection
  - MSS: maximum segment size



TCP: byte stream

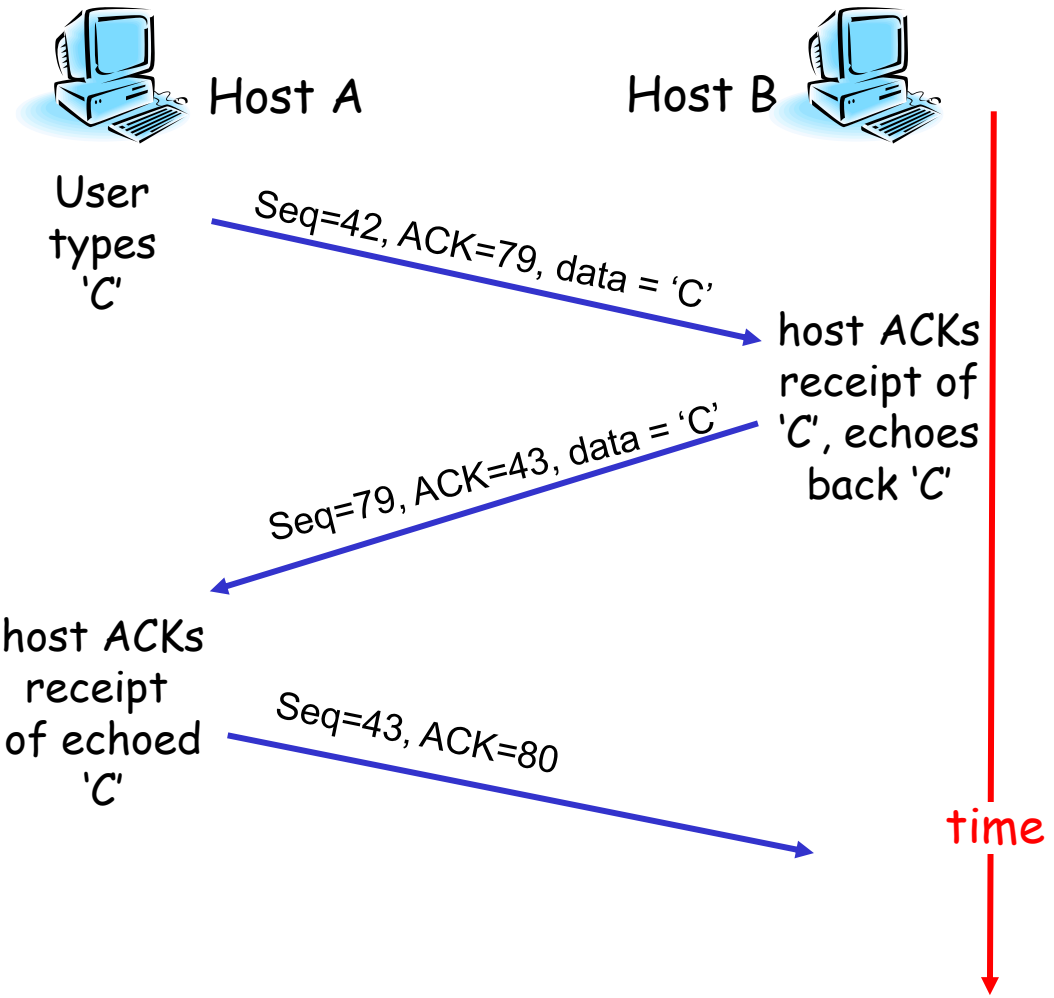
- Each byte is numbered in segment's data
  - E.g., Suppose that a process in Host A wants to send a stream of data (a file of 500,000 bytes) to a process in Host B over a TCP connection. MSS is 1,000 bytes, and the 1st byte is numbered 0.



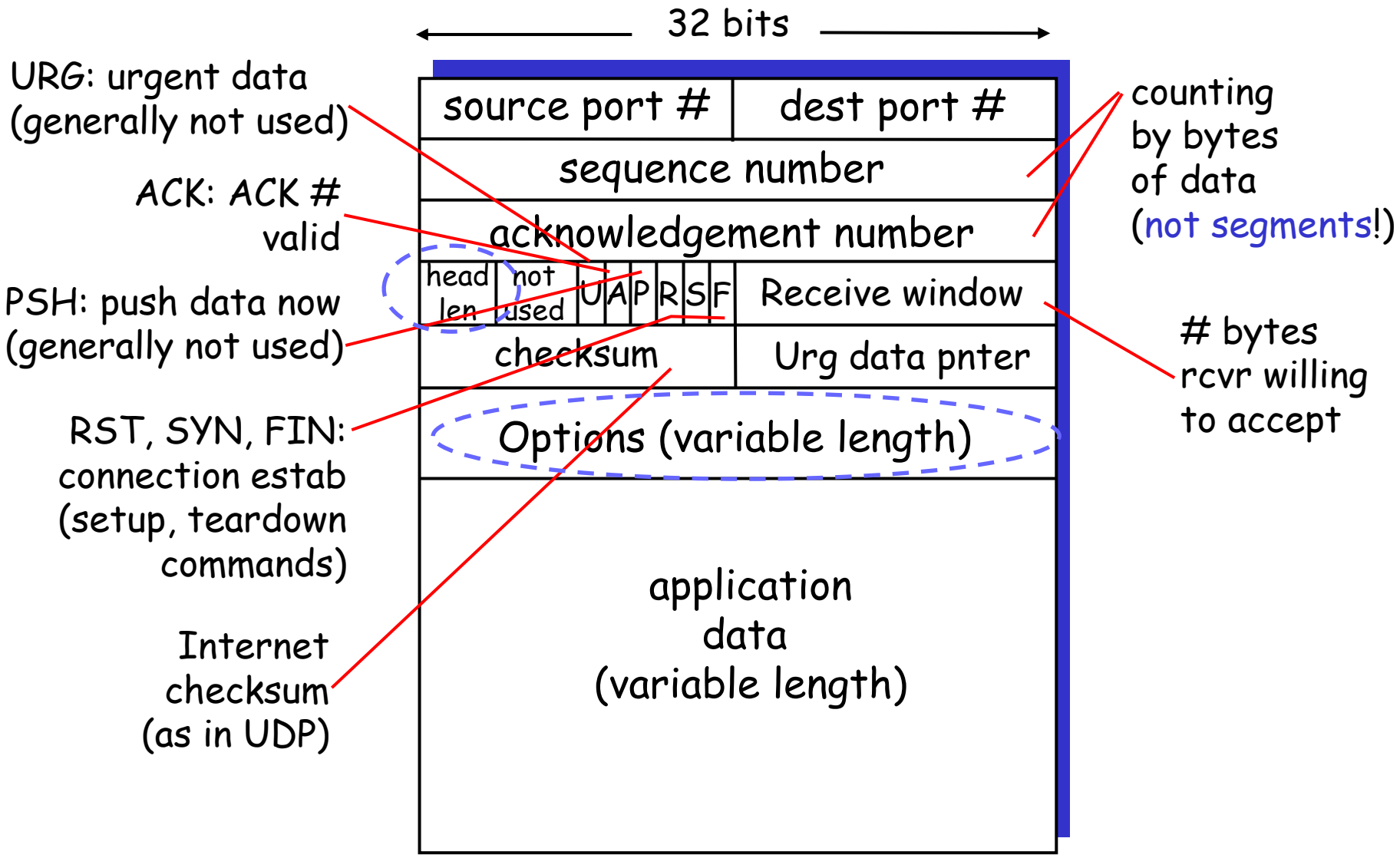
  - Seq # of the 1st segment: 0; Seq # of the 2nd segment: 1000;
  - Seq # of the 3rd segment: ?

TCP Seq. #'s and ACKs

Simple telnet scenario



TCP Segment Structure



Chapter 6 Outline

- 6.1 Transport-layer services
- 6.2 Multiplexing and demultiplexing
- 6.3 Connectionless transport: UDP
- 6.4 Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Reliable data transfer
- 6.5 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

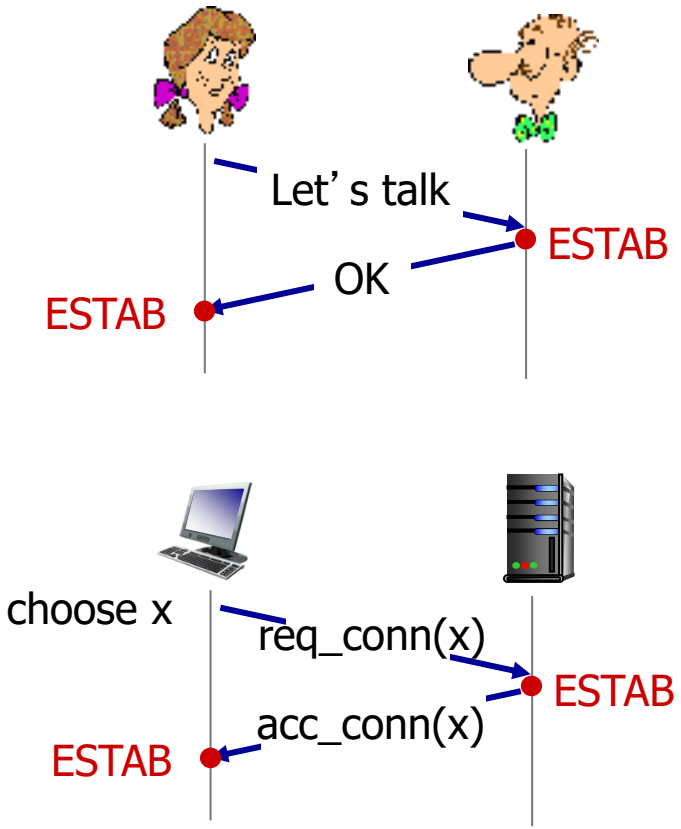
```
Socket clientSocket
  = new Socket("hostname", "port number");
```

- *Server:* contacted by client

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

Agreeing to establish a connection

2-way handshake:

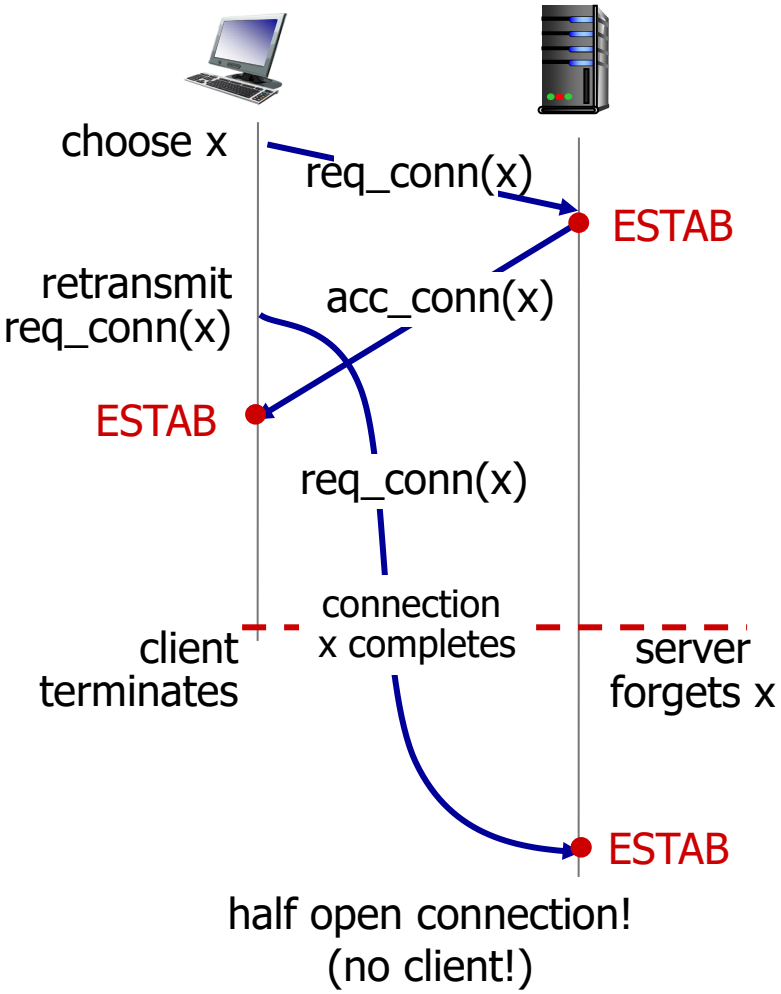


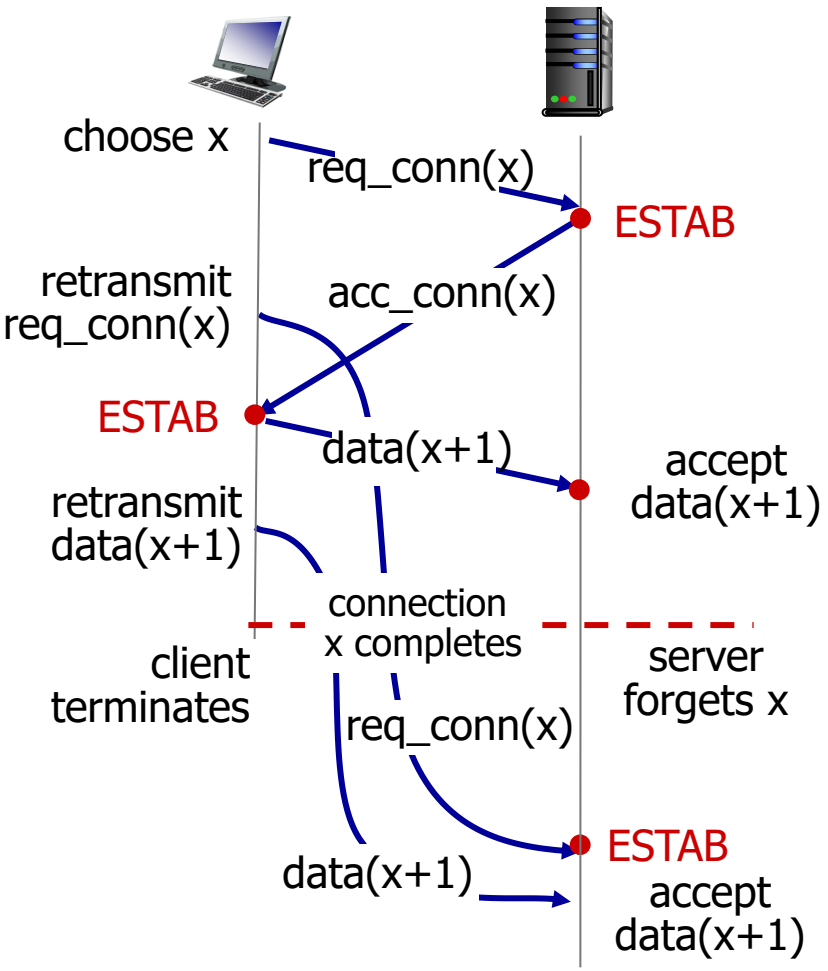
Q: will 2-way handshake always work in network?

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

Agreeing to establish a connection

2-way handshake failure scenarios:





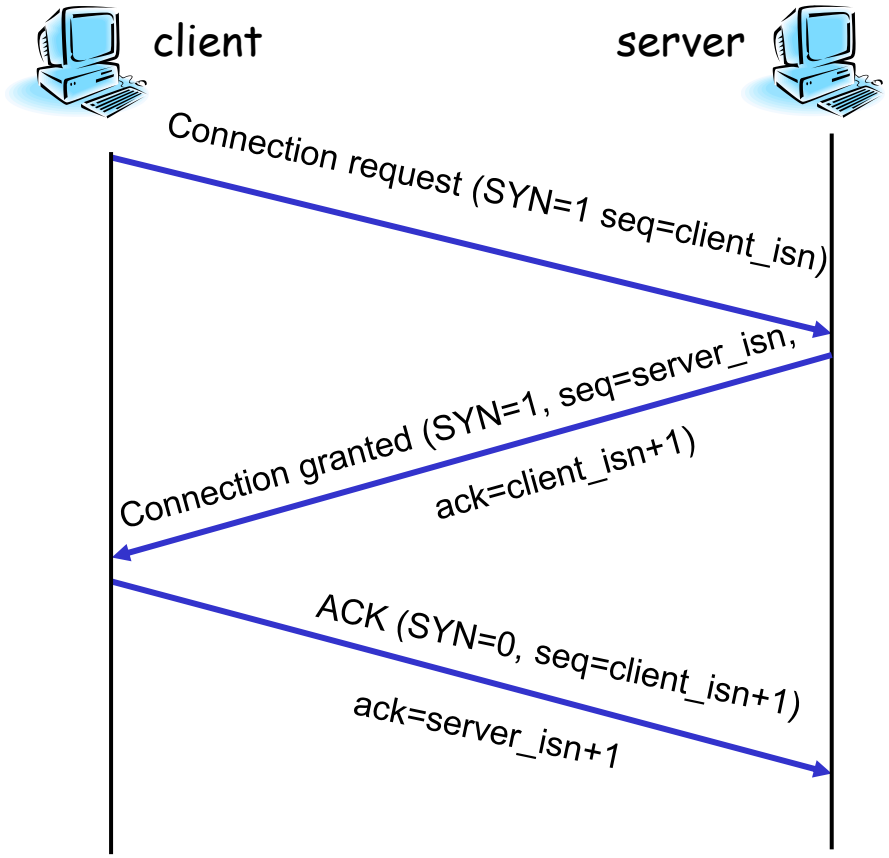
TCP Connection Management (Cont.)

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- Specifies client initial seq #
- No application 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.)

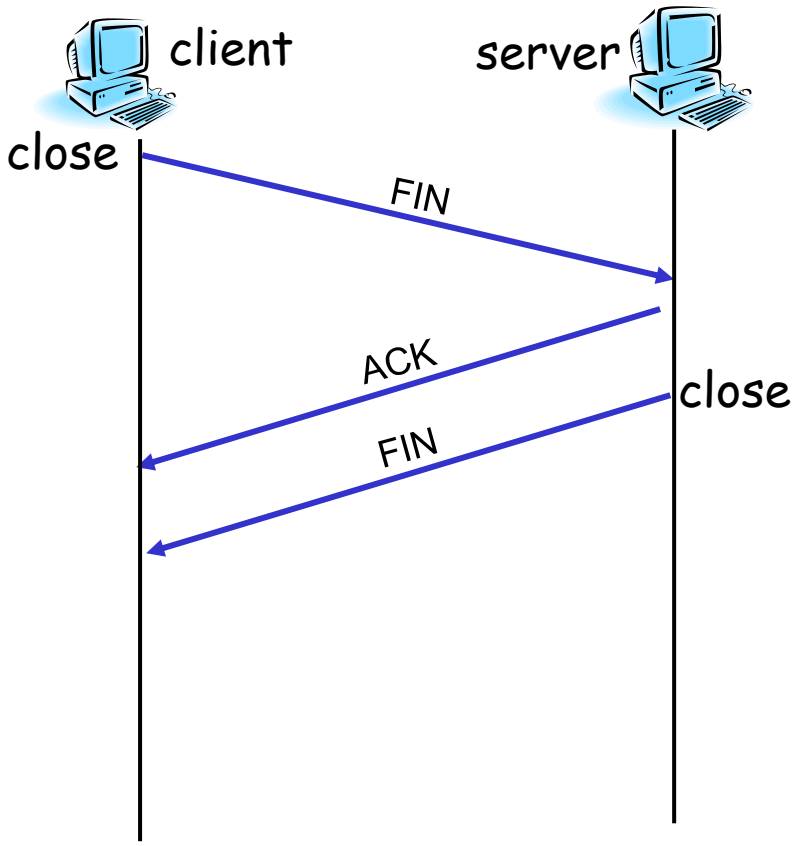
Closing a connection:

client closes socket:

```
clientSocket.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



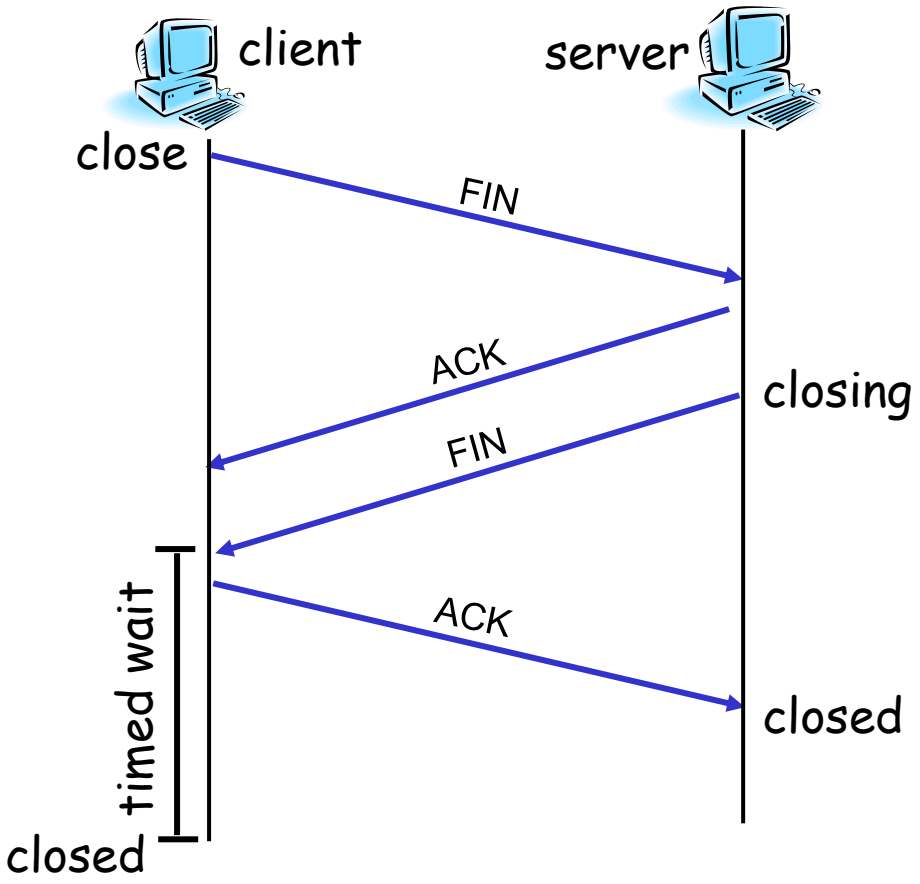
TCP Connection Management (cont.)

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



Exercises-1

1. Host A sends a TCP segment (SYN=1,seq=11220) to host B and expects to establish a TCP connection with host B. if host B accepts the connection request, the correct TCP segment sent by host B to host A may be ( )

(1)SYN=0,ACK=0,seq=11221,ack=11221

(2)SYN=1,ACK=1,seq=11220,ack=11220

(3)SYN=1,ACK=1,seq=11221,ack=11221

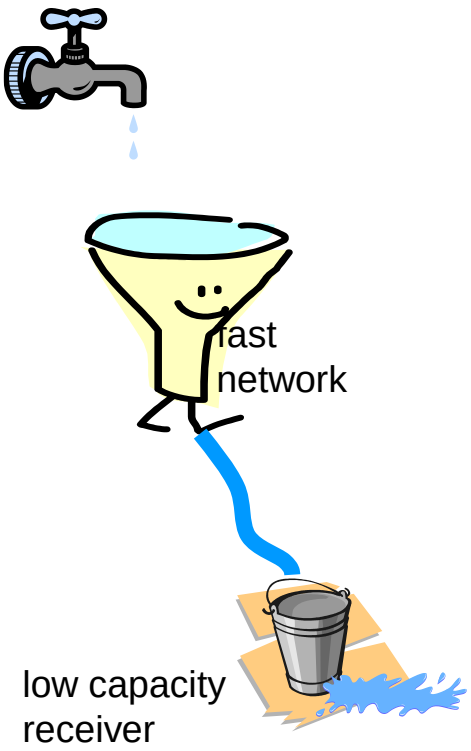
(4)SYN=0,ACK=0,seq=11220,ack=11220

Chapter 6 Outline

- 6.1 Transport-layer services
- 6.2 Multiplexing and demultiplexing
- 6.3 Connectionless transport: UDP
- 6.4 Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Reliable data transfer
- 6.5 TCP congestion control

TCP Flow Control

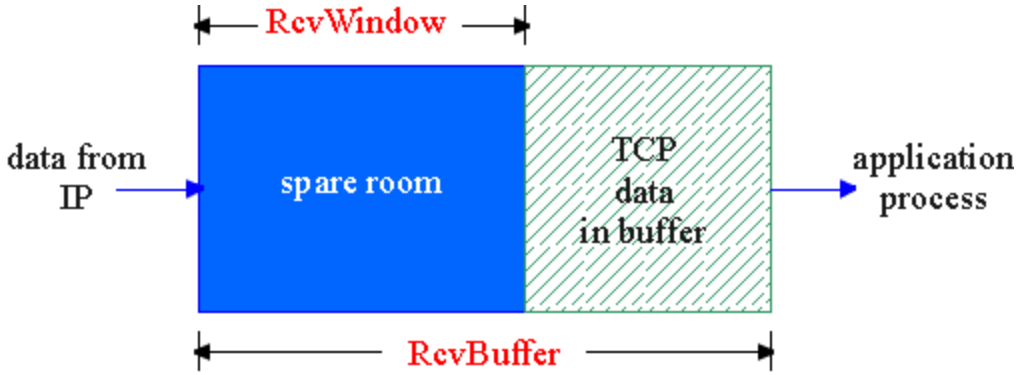
- Sender should not overwhelm receiver's capacity to receive data
- If necessary, sender should slow down transmission rate to accommodate receiver's rate
- Different from Congestion Control whose purpose was to handle congestion in network
- But both congestion control and flow control work by slowing down data transmission



flow control problem

TCP Flow Control

- Receive side of TCP connection has a receive buffer:



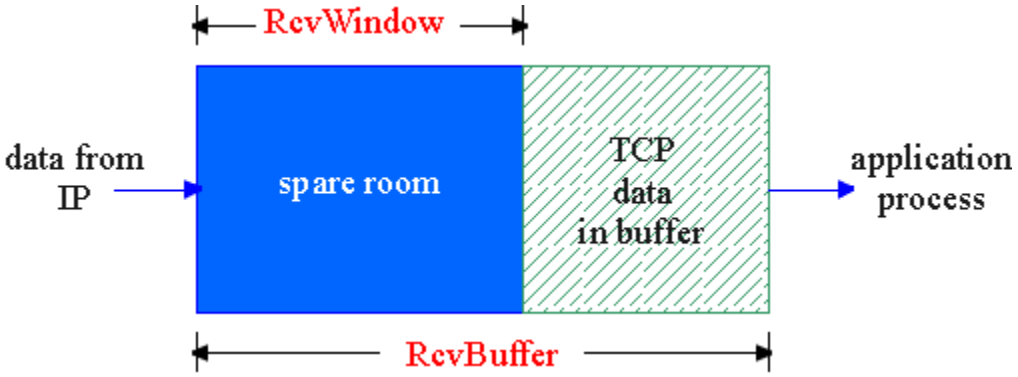
- App process may be slow at reading from buffer

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

Technical Issue

- Suppose RcvWindow=0 and that receiver has already ACK'd ALL packets in buffer
- Sender does not transmit new packets until it hears RcvWindow>0
- Receiver never sends RcvWindow>0 since it has no new ACKS to send to Sender
- **DEADLOCK!**

- Solution:
  - TCP specs require sender to continue sending packets with one data byte while RcvWindow=0, just to keep receiving ACKS from receiver
  - At some point the receiver's buffer will empty and RcvWindow>0 will be transmitted back to sender

Exercises-2

1. A TCP connection is established between host A and host B, and the MSS length of TCP is 1000 bytes. If the current congestion window of host A is 4000 bytes, after host A sends two MSS segments to host B continuously, host A successfully receives the confirmation segment of the first segment sent by host B, and the size of the receiving window in the confirmation segment is 2000 bytes, then the maximum number of bytes that host A can also send to host B is ( )

(1)1000

(2)2000

(3)3000

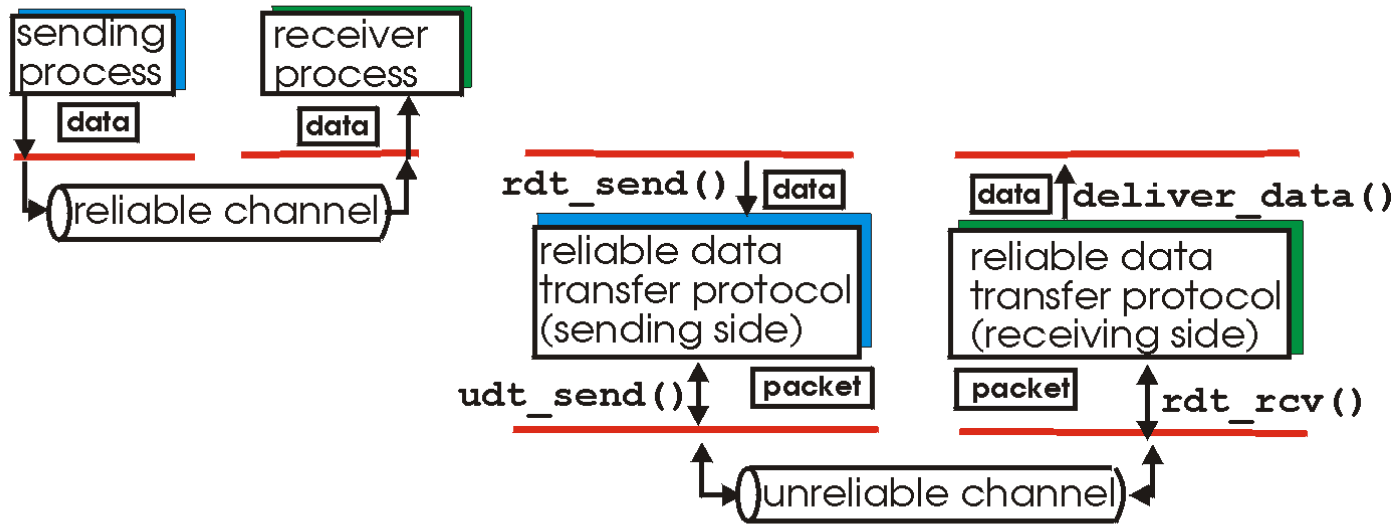
(4)4000

Chapter 6 Outline

- 6.1 Transport-layer services
- 6.2 Multiplexing and demultiplexing
- 6.3 Connectionless transport: UDP
- 6.4 Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Reliable data transfer
- 6.5 TCP congestion control

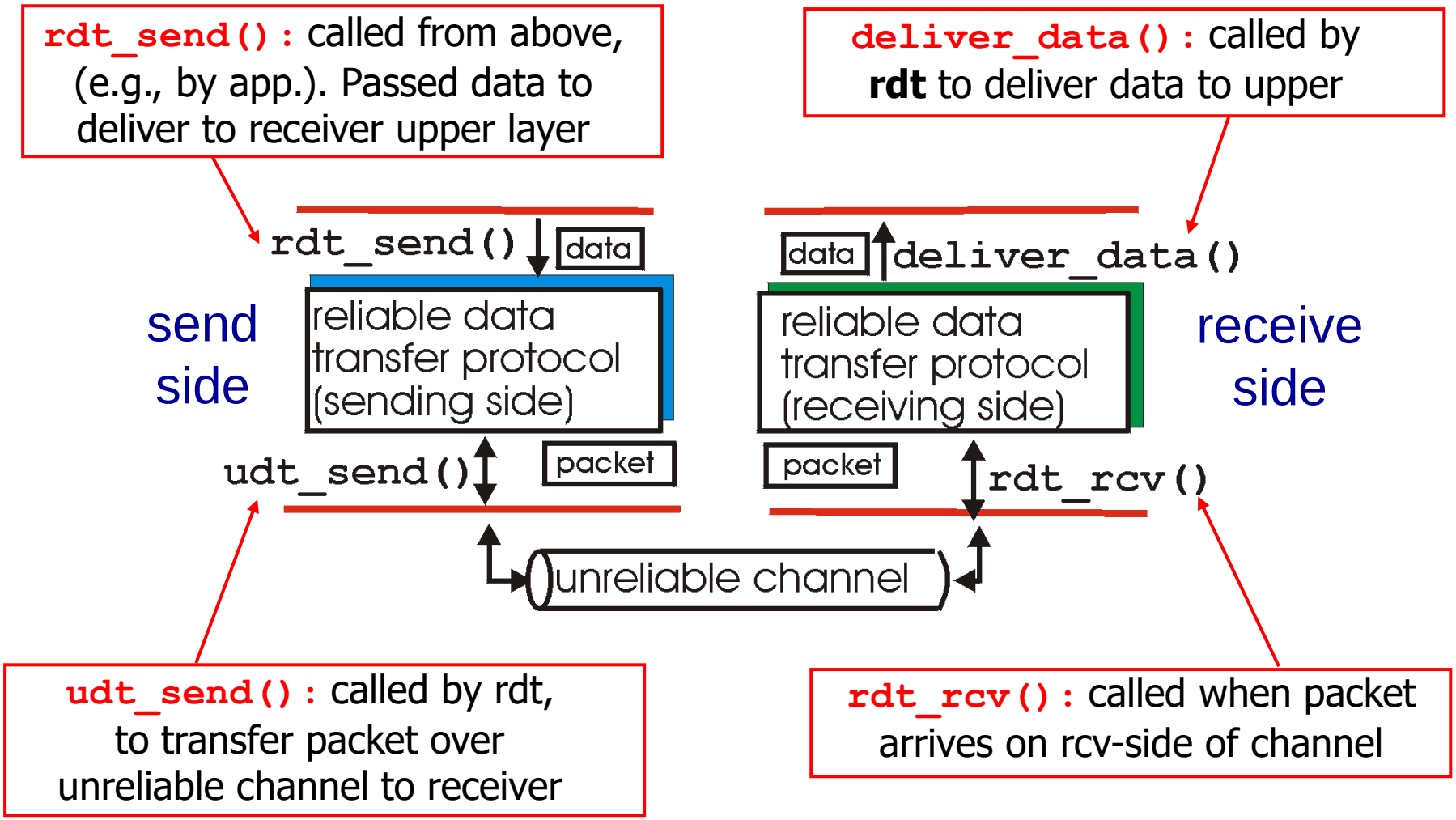
Reliable Data Transfer

- 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

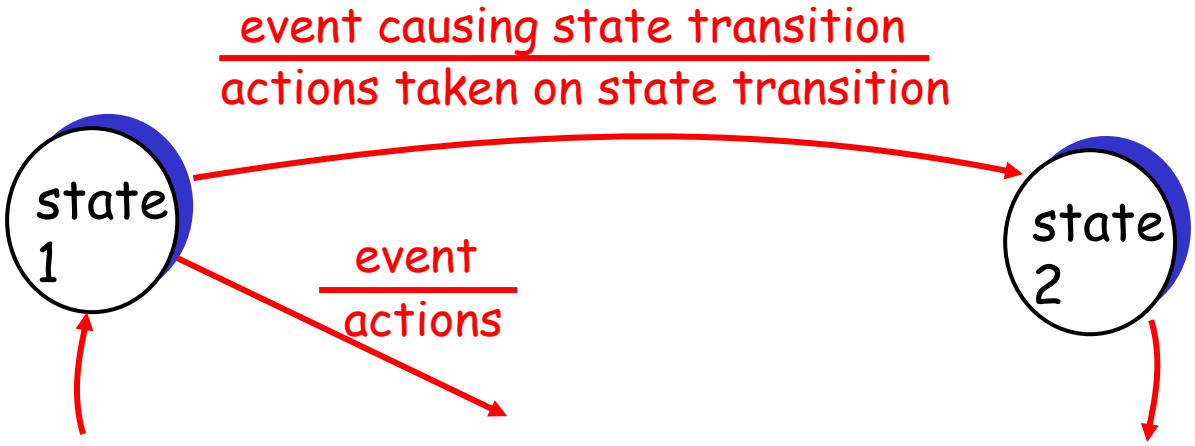


Reliable Data Transfer: Getting Started

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

state: when in this "state" next state uniquely determined by next event

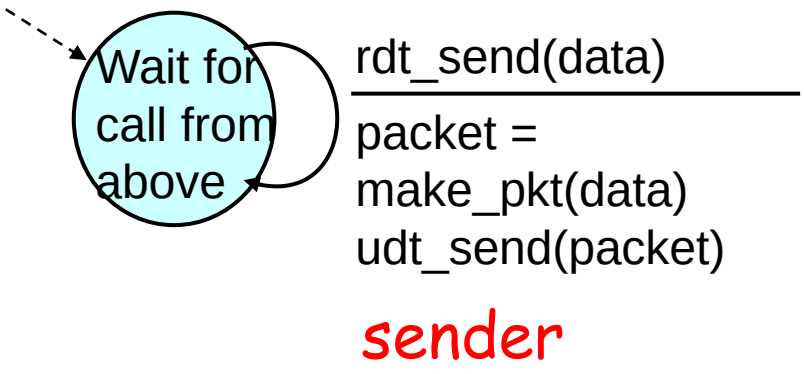


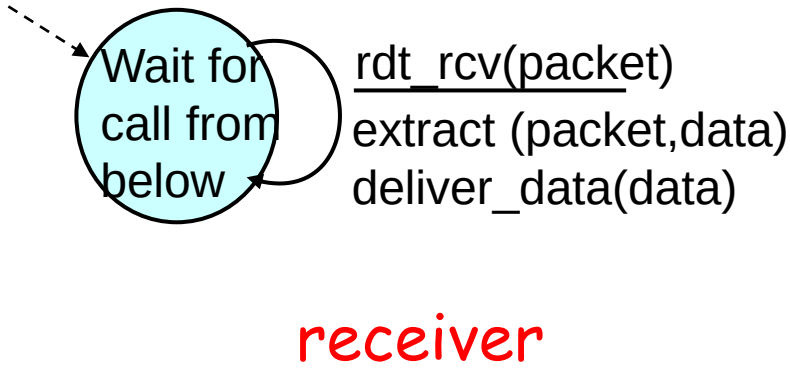
Summary of the Protocols

- Rdt1.0: all packets arrive correctly
- Rdt2.0: all packets arrive, with possible errors only in data packets, and introducing ACK and NAK (no error)
- Rdt2.1: corrupted ACKs/NAKs
- Rdt2.2: similar to rdt2.1 remove NAKs
- Rdt3.0: Allows packets to be lost and errors

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

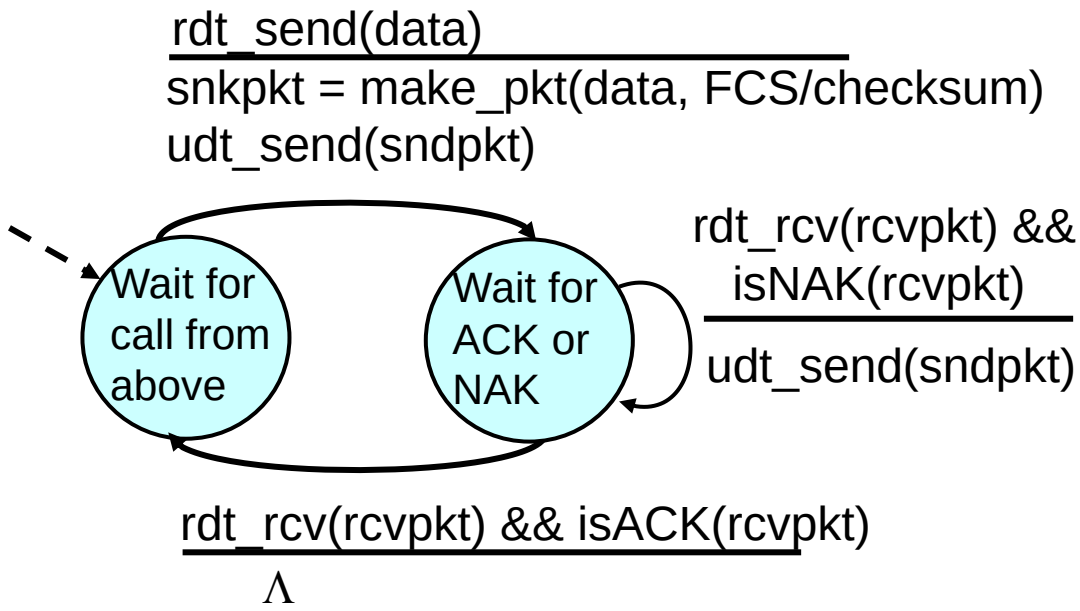


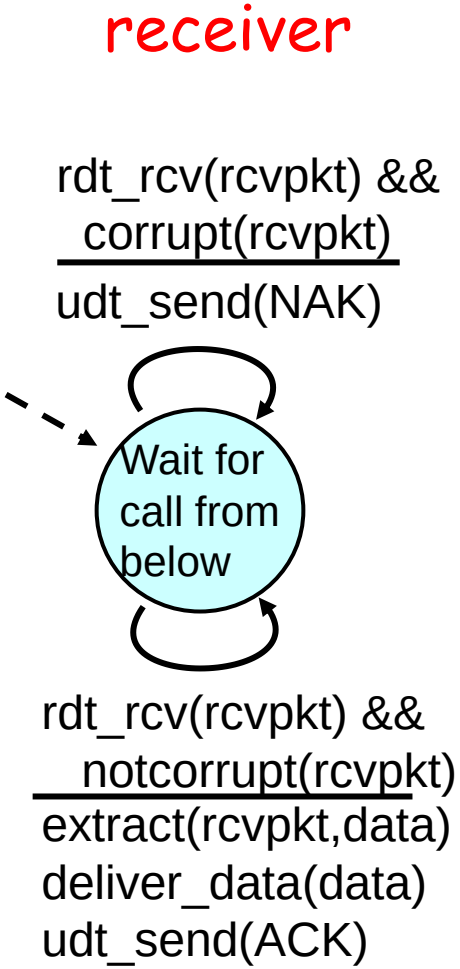


Rdt2.0: Channel with Bit Errors

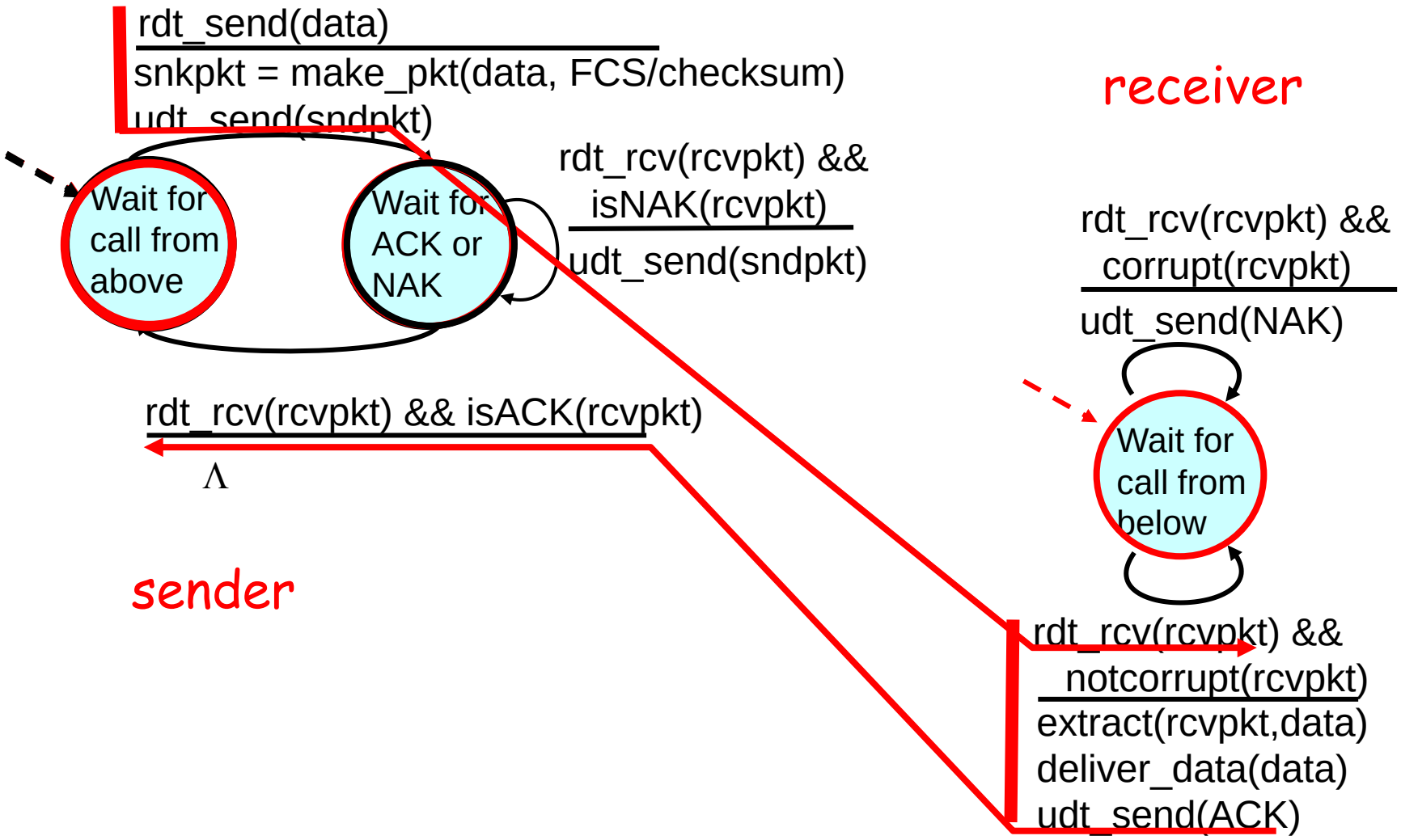
- Underlying channel may flip bits in packet
  - FCS/Checksum to detect bit errors
- *The question:* how to recover from errors:
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors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

Rdt2.0: FSM specification

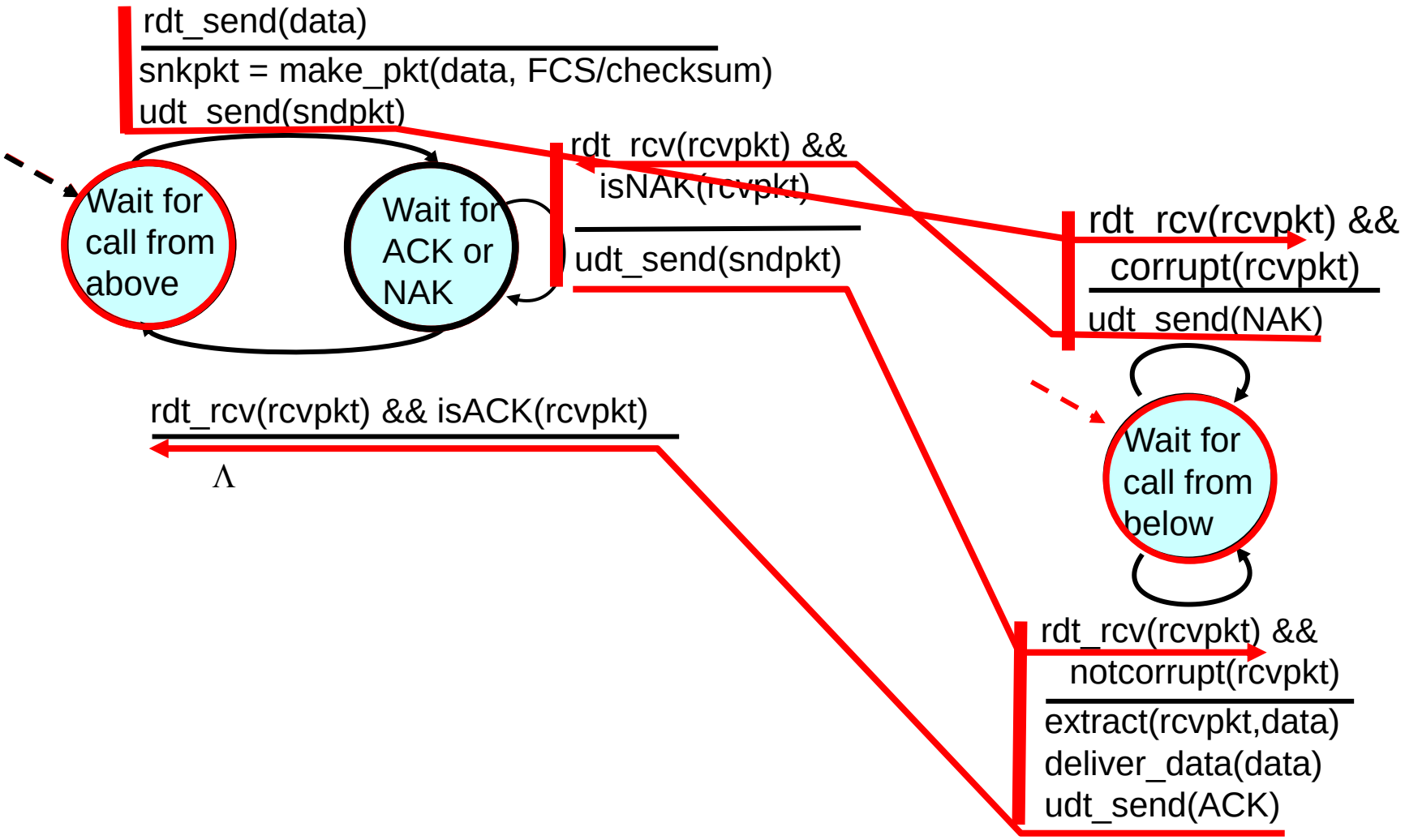




Rdt2.0: Operation with No Errors



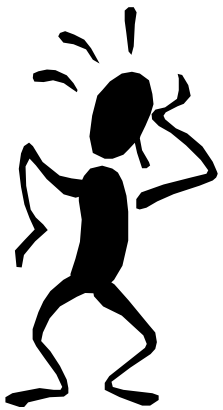
Rdt2.0: Error Scenario



Rdt2.0 Has a Fatal Flaw!

What happens if ACK/NAK corrupted?

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



Handling duplicates:

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

stop and wait

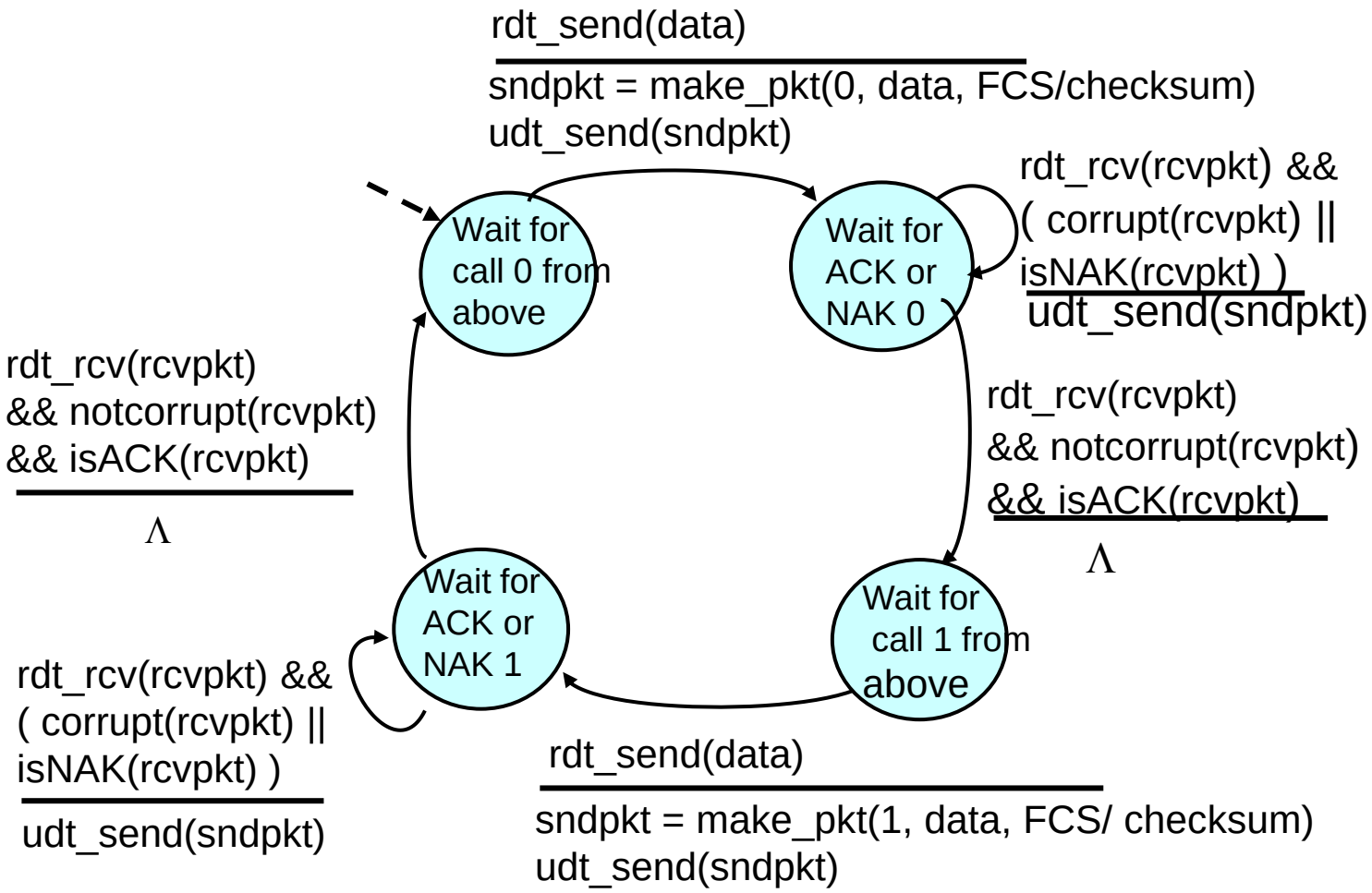
Sender sends one packet, then waits for receiver response

- Sender: whenever sender receives control message it sends a packet to receiver
  - A valid ACK: Sends next packet (if exists) with new sequence #
  - A NAK or corrupt response: resends old packet

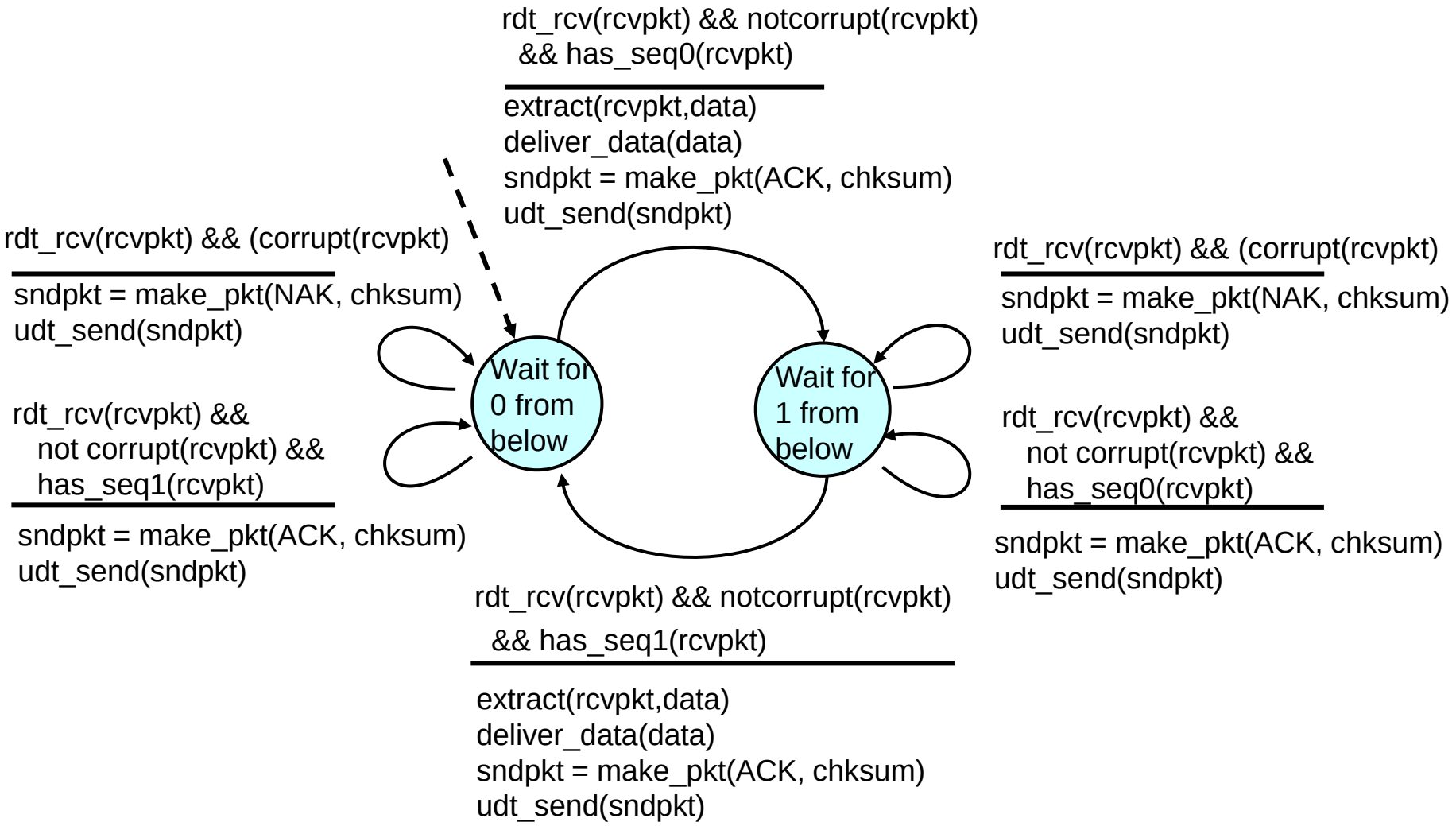
- Receiver: sends ACK/NAK to sender
  - If received packet is corrupt: send NAK
  - If received packet is valid and has different sequence # as prev packet: send ACK and deliver new data up
  - If received packet is valid and has same sequence # as prev packet, i.e., is a retransmission of duplicate: send ACK

- Note: ACK/NAK **do not** contain sequence #

Rdt2.1: Sender, Handles Garbled ACK/NAKs



Rdt2.1: Receiver, Handles Garbled ACK/NAKs



Rdt2.1: Discussion

Sender:

- Seq # added to pkt
- Two seq. #'s (0,1) will suffice. Why?
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must "remember" whether "current" pkt has 0 or 1 seq. #

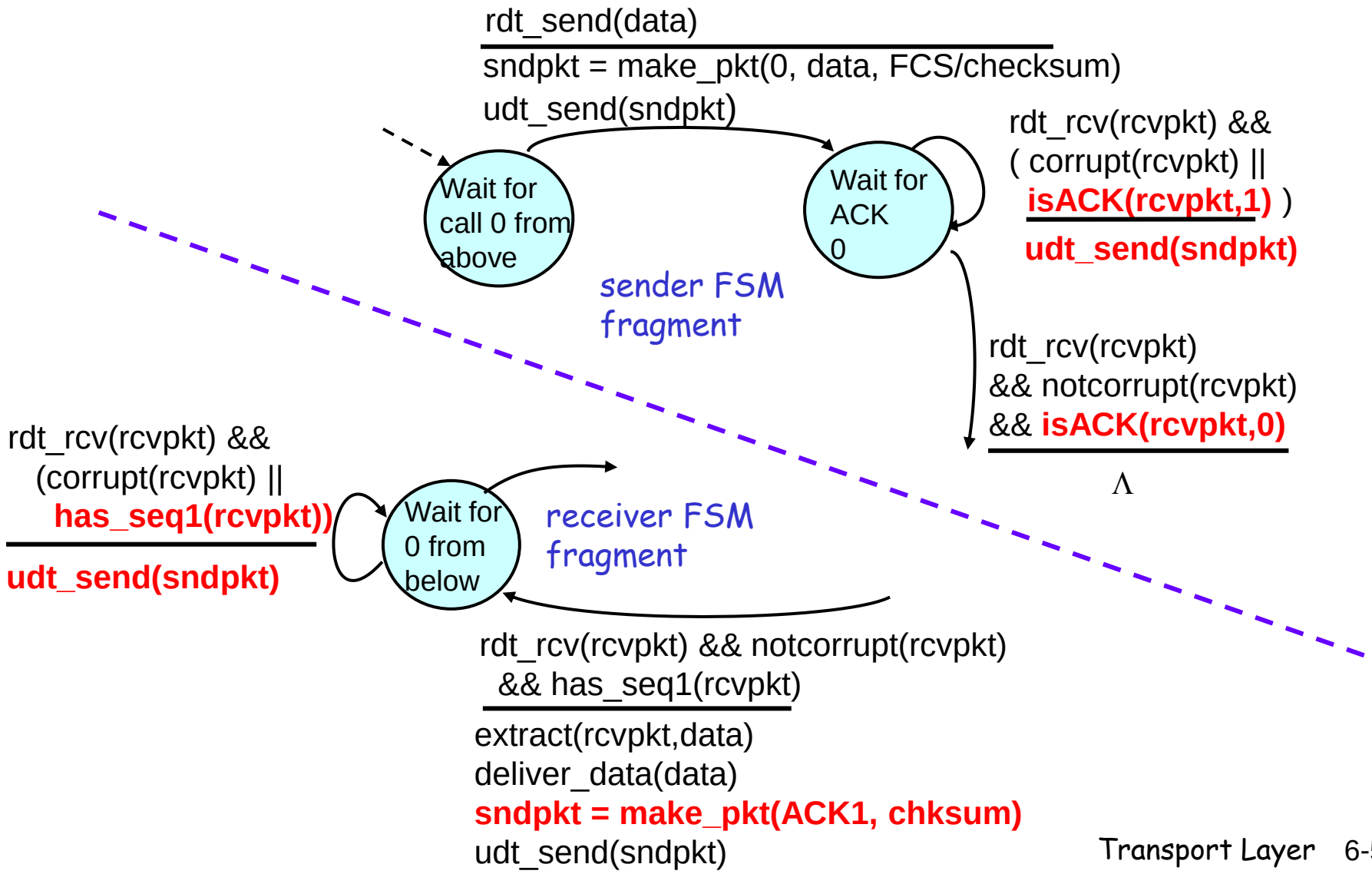
Receiver:

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

Rdt2.2: a NAK-free Protocol

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

Rdt2.2: Sender, Receiver Fragments



Rdt3.0: Channels with Errors and Loss

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

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

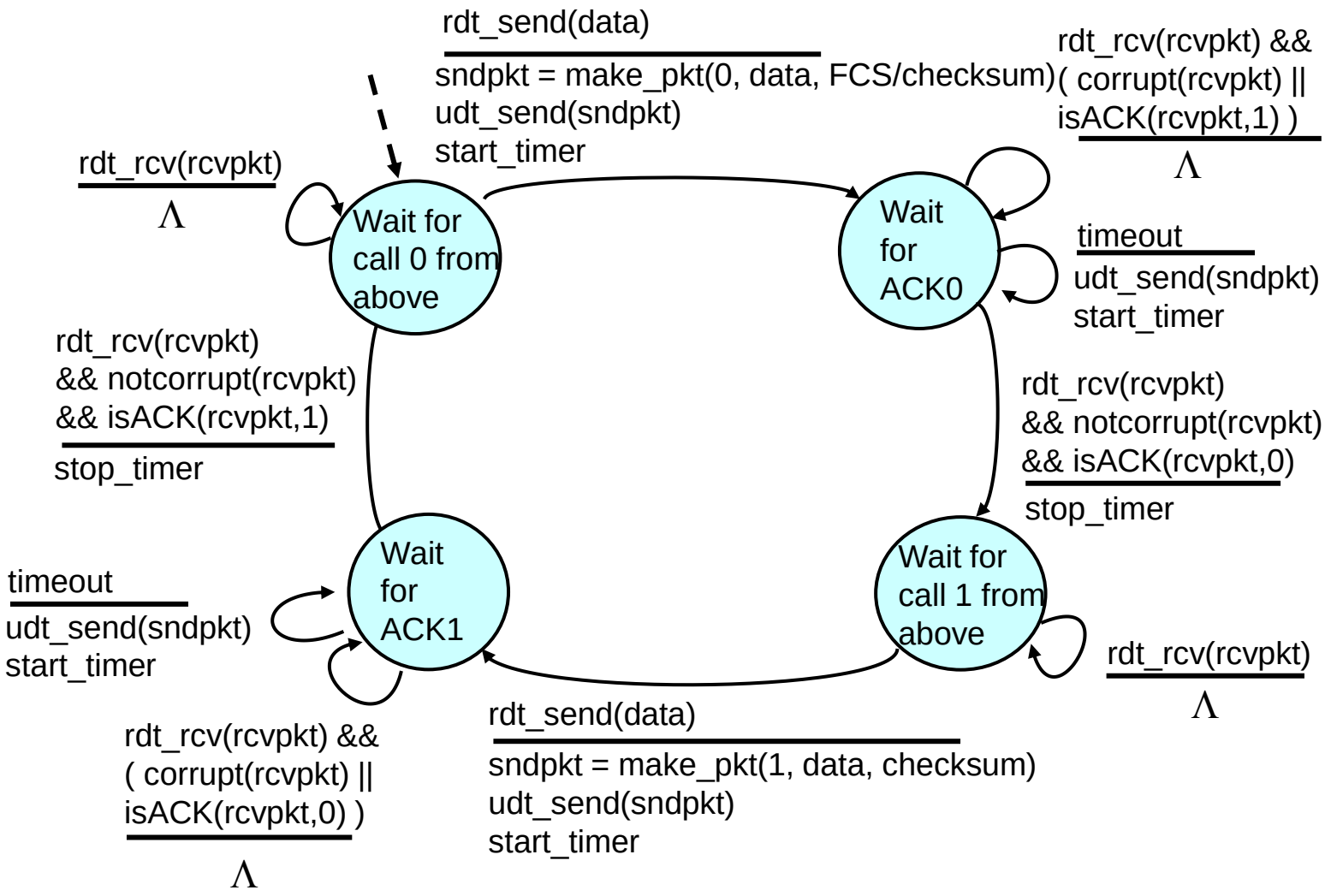
Q: how to deal with loss?

- Sender waits until certain data or ACK lost, then retransmits
- Any drawbacks?

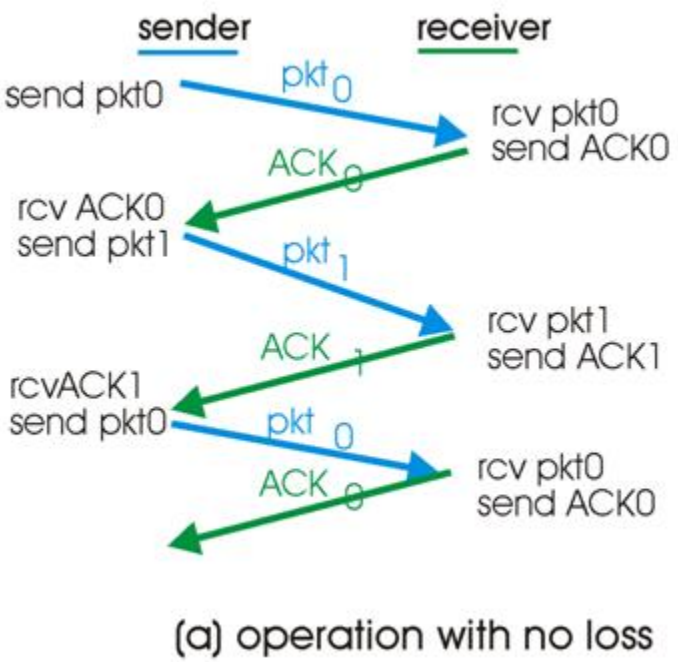
Approach: sender waits "reasonable" amount of time for ACK

- 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

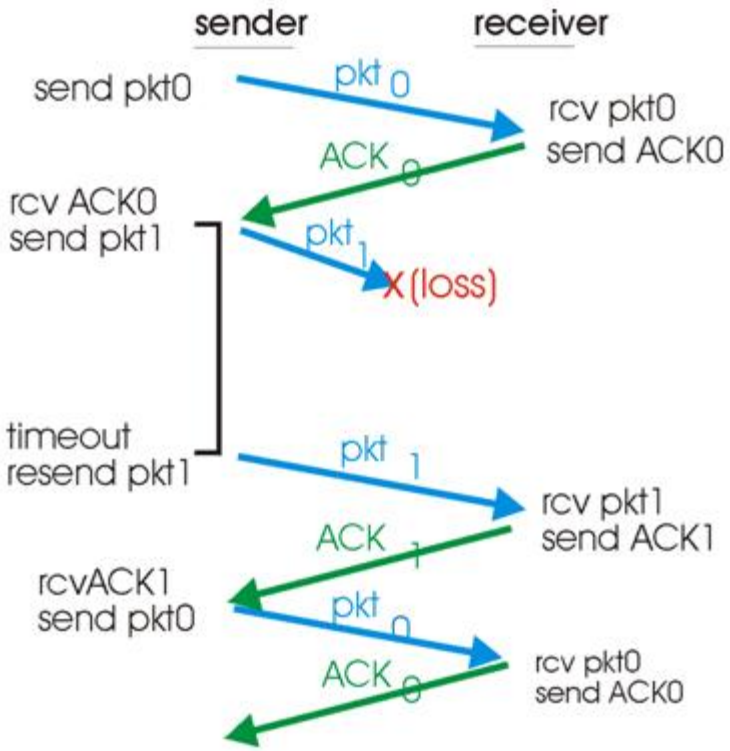
Rdt3.0 Sender



Rdt3.0 In Action

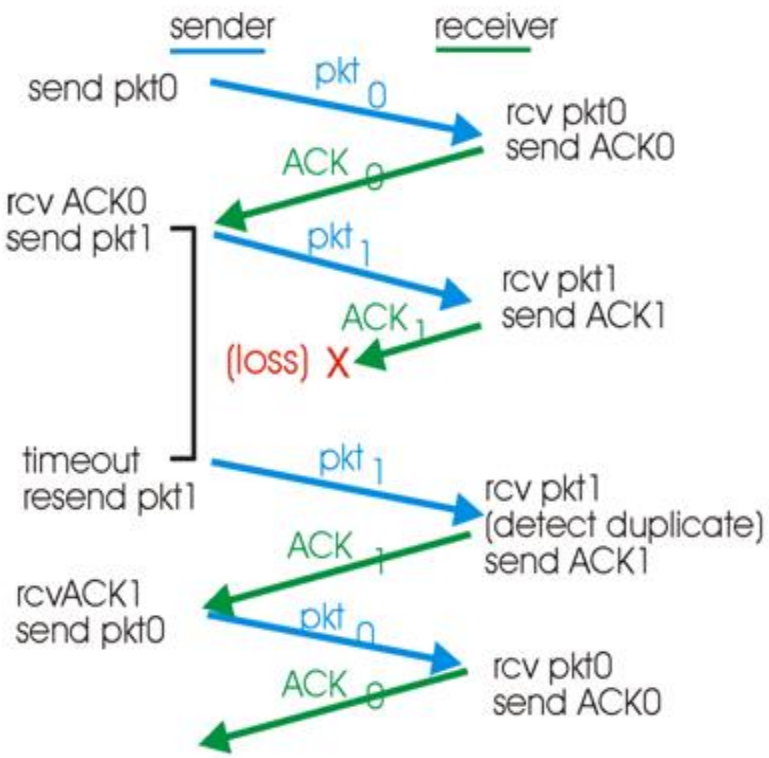


(a) operation with no loss

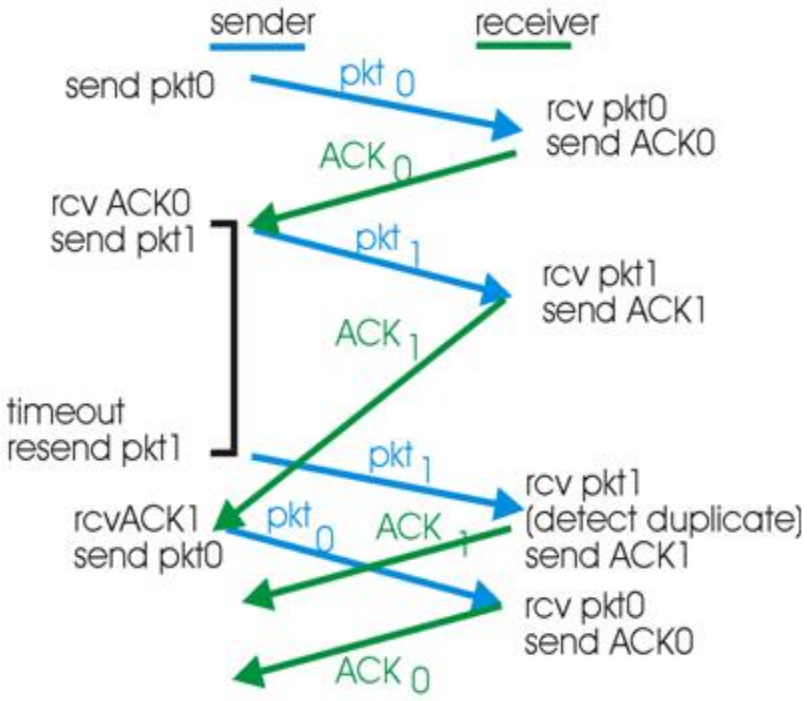


(b) lost packet

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:

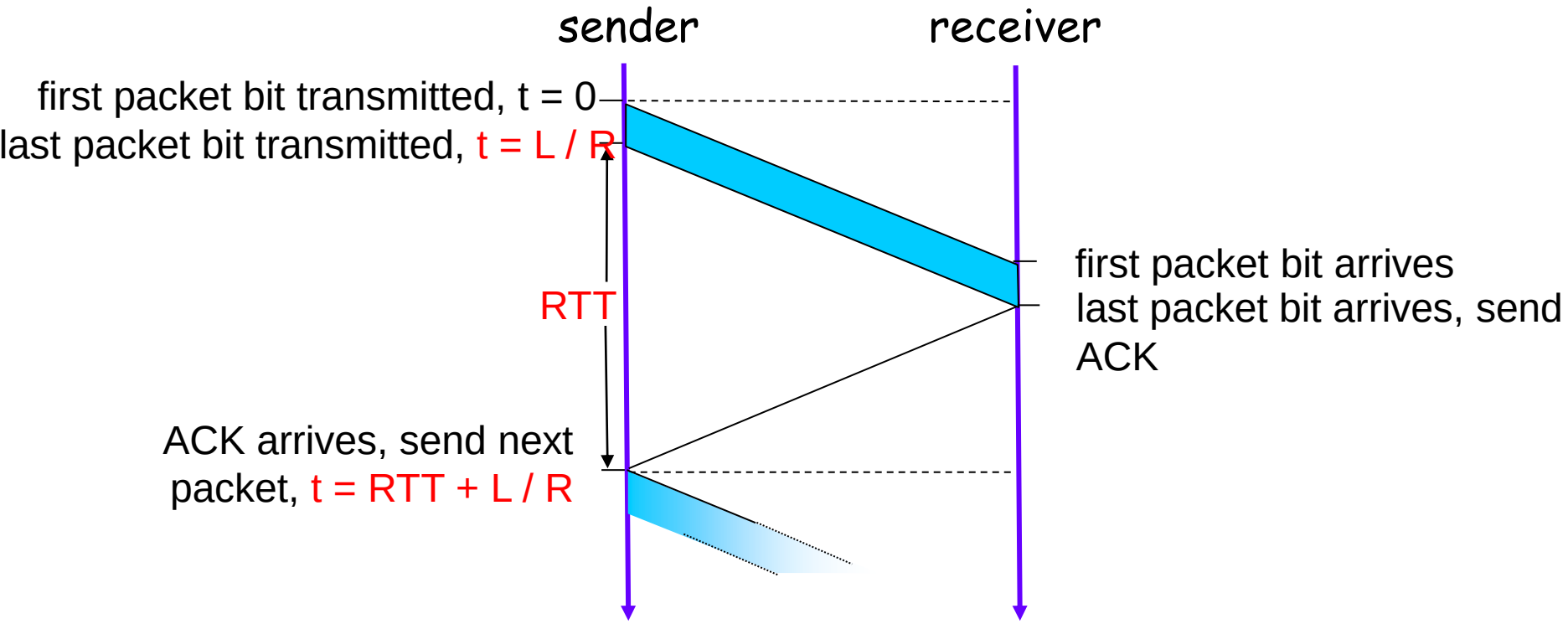
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

- 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

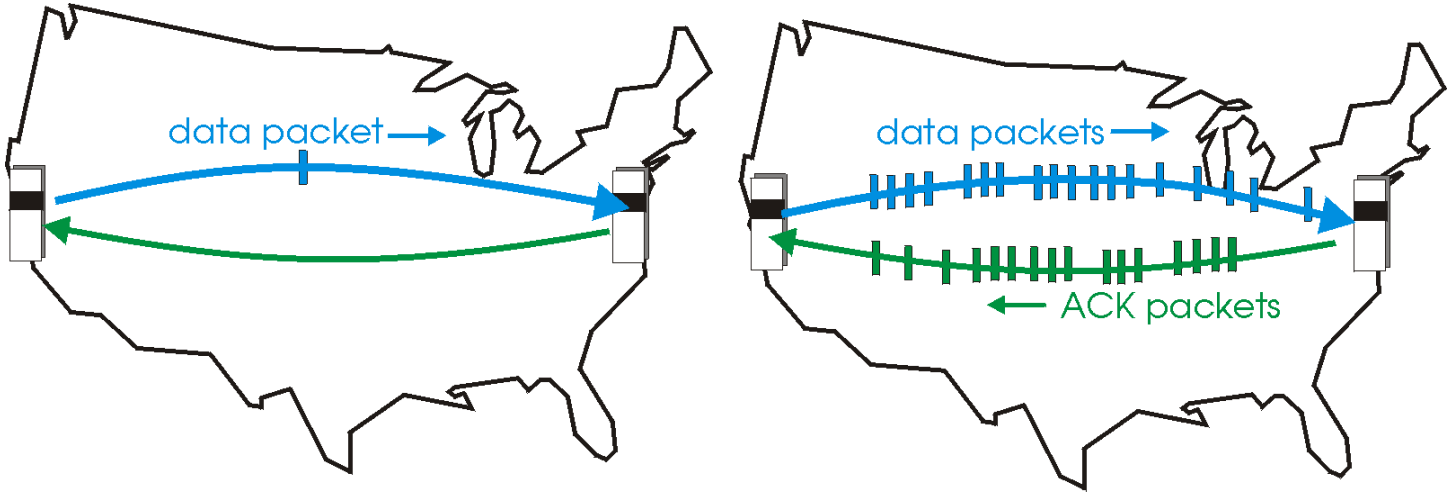


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined Protocols

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

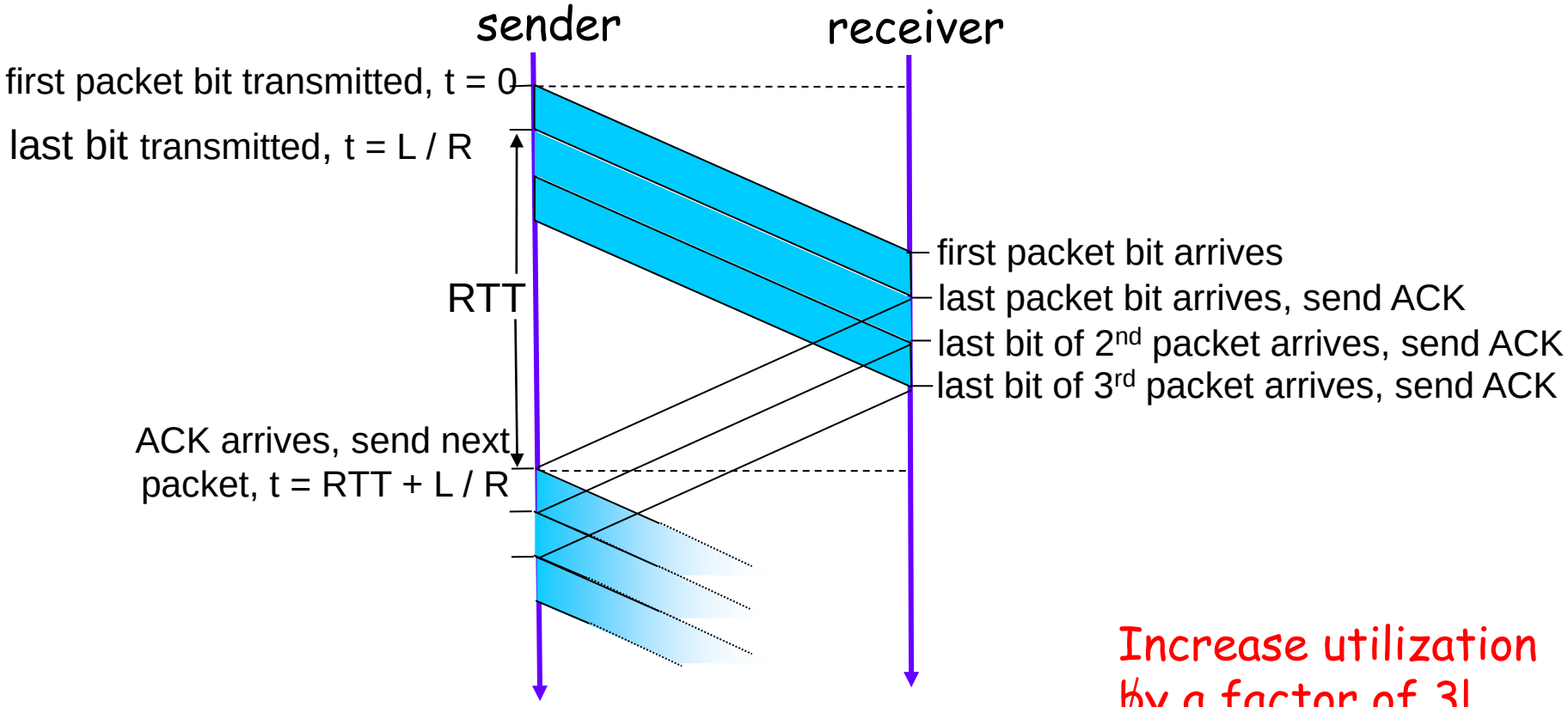
- Range of sequence numbers must be increased
- Buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

Pipelining: Increased Utilization



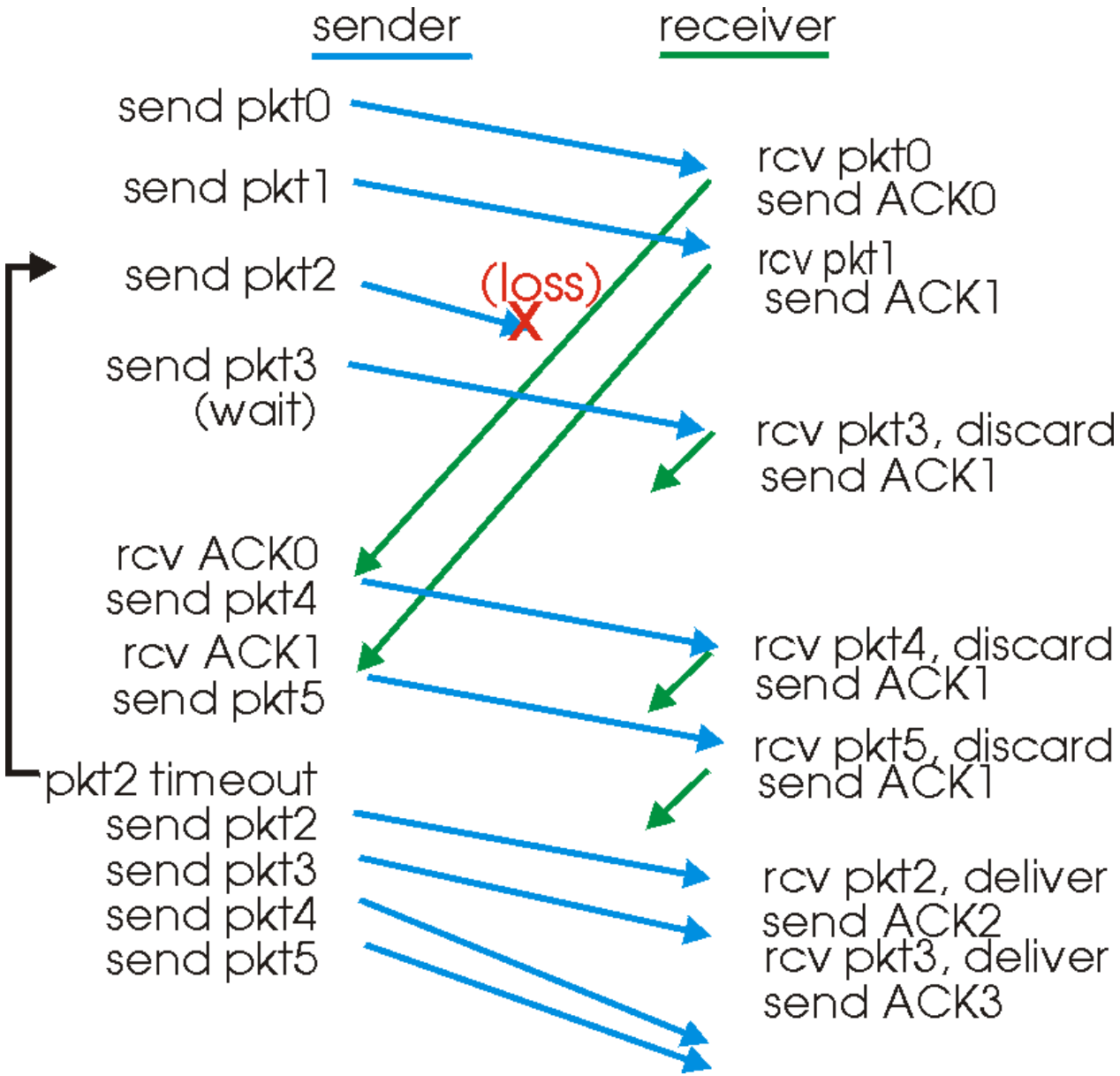
Increase utilization by a factor of 3!

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

Pipelined Protocols

- Advantage: much better bandwidth utilization than stop-and-wait
- Disadvantage: More complicated to deal with reliability issues, e.g., corrupted, lost, out of order data.
  - Two generic approaches to solving this
    - *Go-Back-N protocols*
    - *Selective repeat protocols*

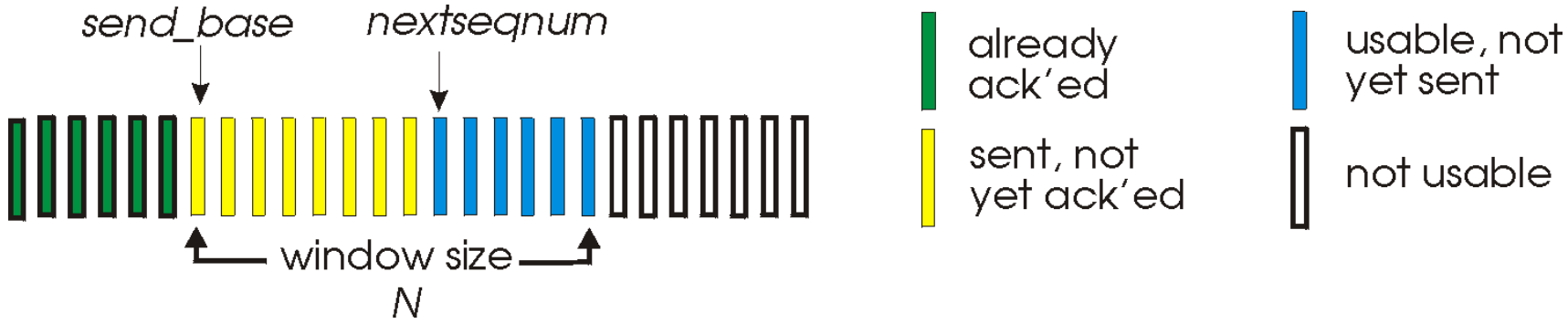
GBN In Action



Go-Back-N

Sender:

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



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - May receive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- *Timeout(n)*: retransmit pkt n and all higher seq # pkts in window
- Called a sliding-window protocol

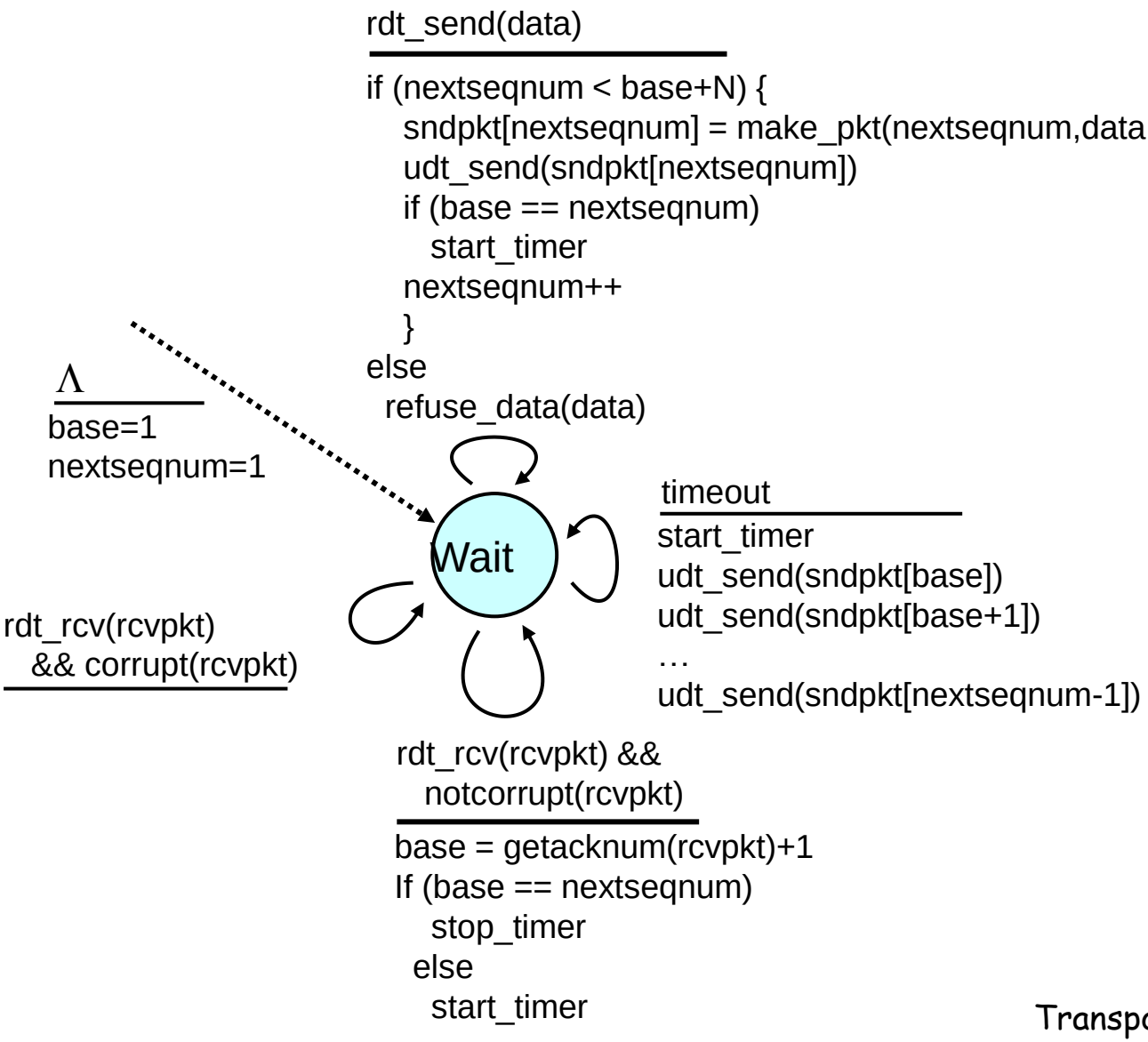
GBN: Sender

- rdt_Send() called: checks to see if window is full
  - No: send out packet
  - Yes: return data to upper level

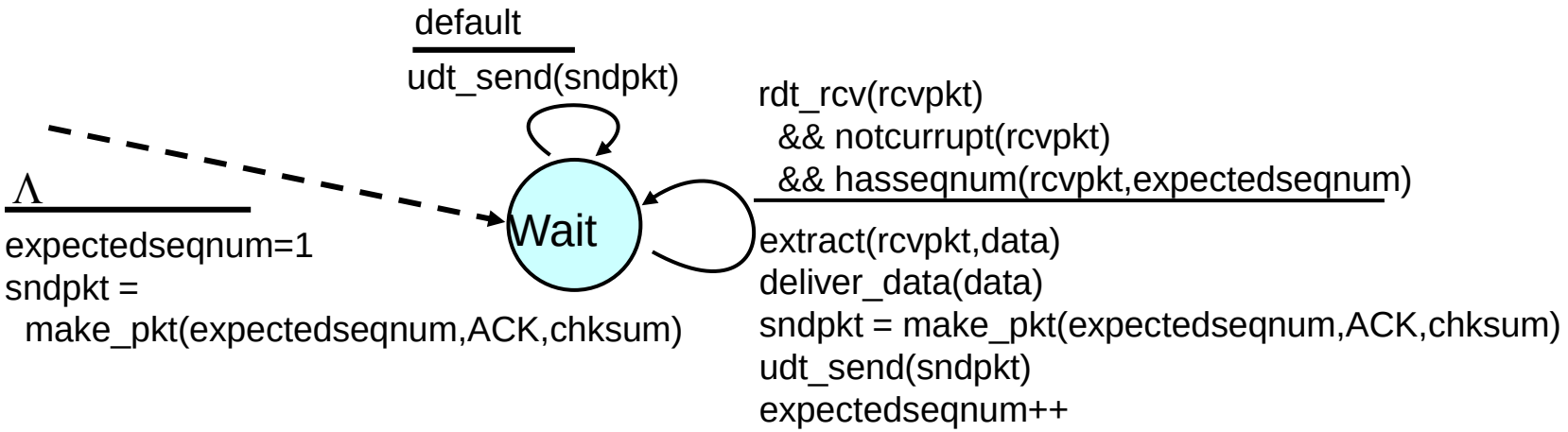
- Receipt of ACK(n): cumulative acknowledgement that all packets up to and including *n* have been received. Updates window accordingly and restarts timer

- Timeout: resends ALL packets that have been sent but not yet acknowledged.
  - This is only event that triggers resend

GBN: Sender Extended FSM



GBN: Receiver Extended FSM



- If expected packet received:
  - Send ACK and deliver packet upstairs
- If out-of-order packet received:
  - Discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #
  - May generate duplicate ACKs

More on Receiver

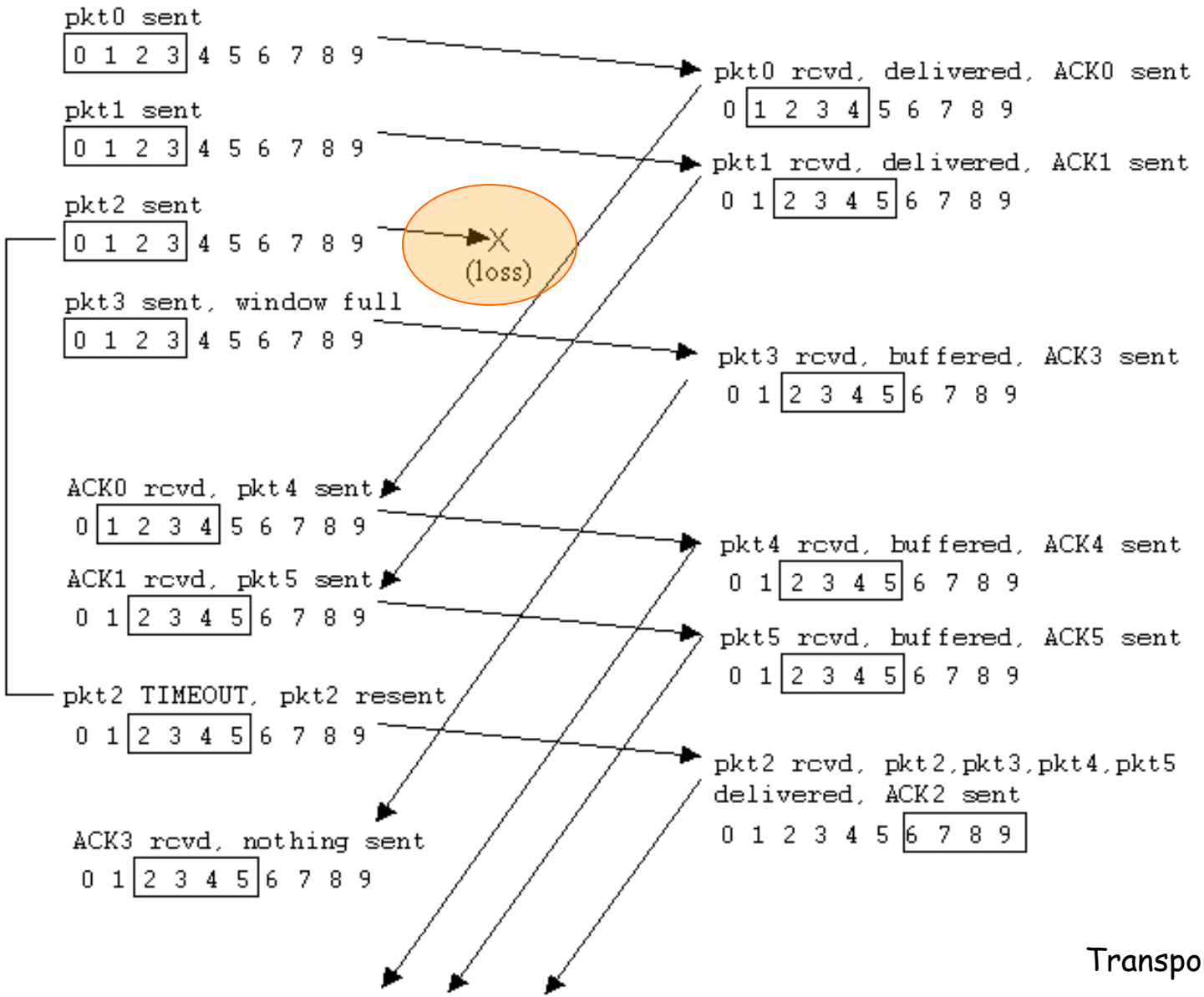
- ACK-only: the receiver always sends ACK for last correctly received packet with highest *in-order* seq #
- Receiver only sends ACKS (no NAKs)
- Need only remember **`expectedseqnum`**
- Can generate duplicate ACKs

- GBN is easy to code but might have performance problems

- In particular, if many packets are in pipeline at one time (bandwidth-delay product large) then one error can force retransmission of huge amounts of data!

- Selective Repeat protocol allows receiver to buffer data and only forces retransmission of required packets

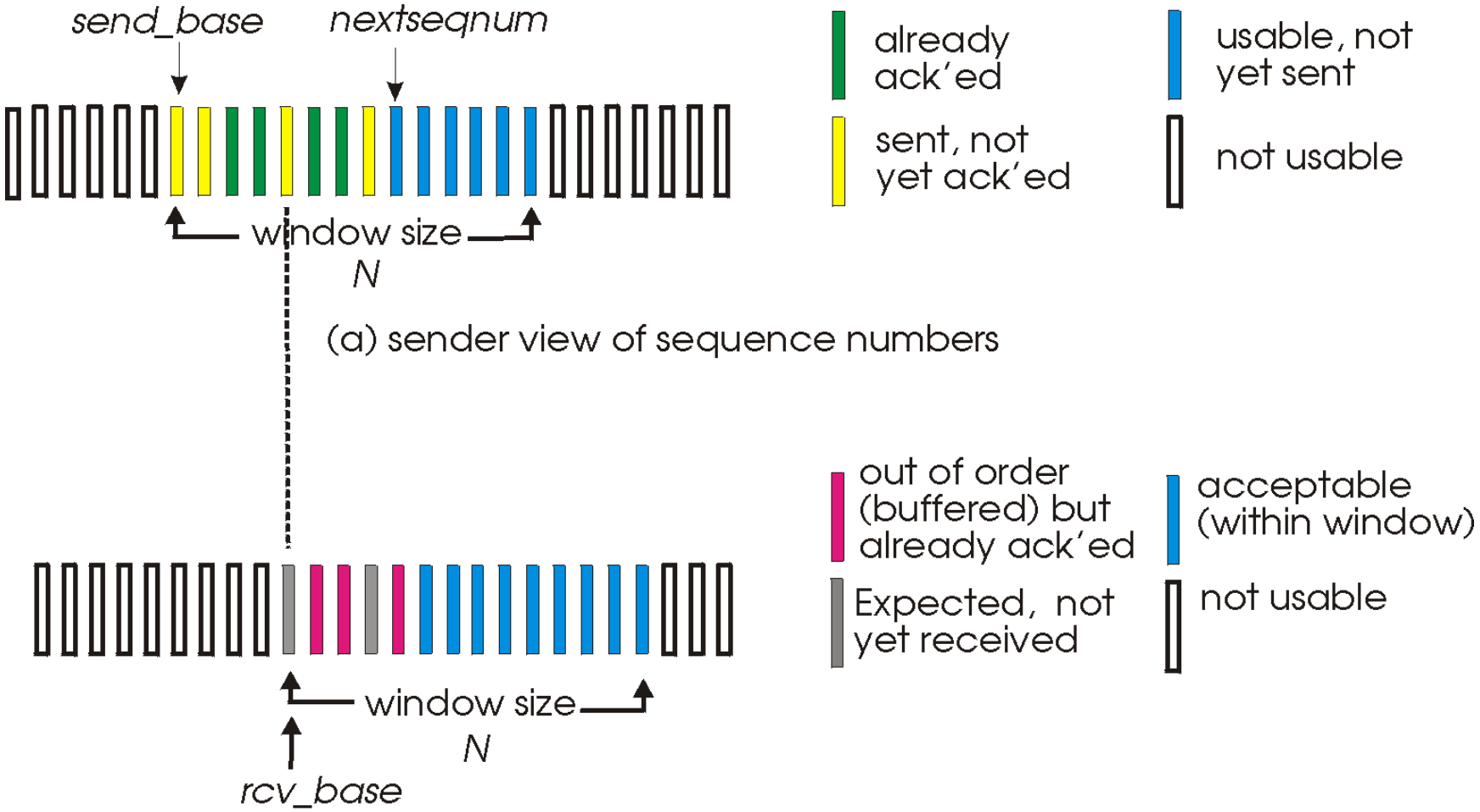
Selective Repeat In Action



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
  - Compare to GBN which only had timer for base packet
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACKed pkts
  - Important: Window size < seq # range

Selective Repeat: Sender, Receiver Windows



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

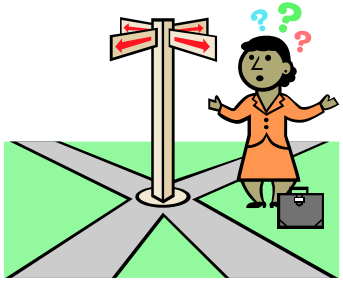
Selective Repeat: Dilemma

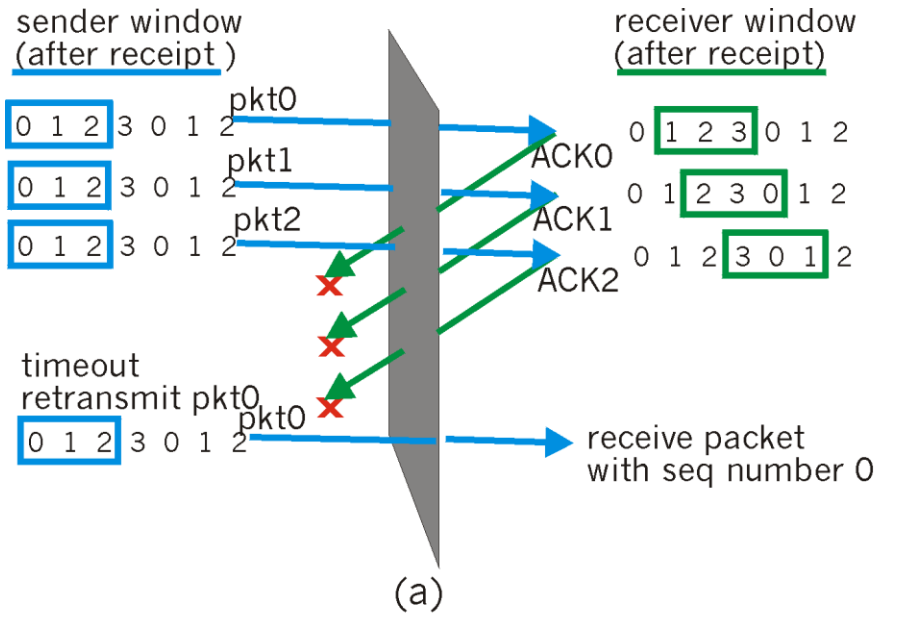
Example:

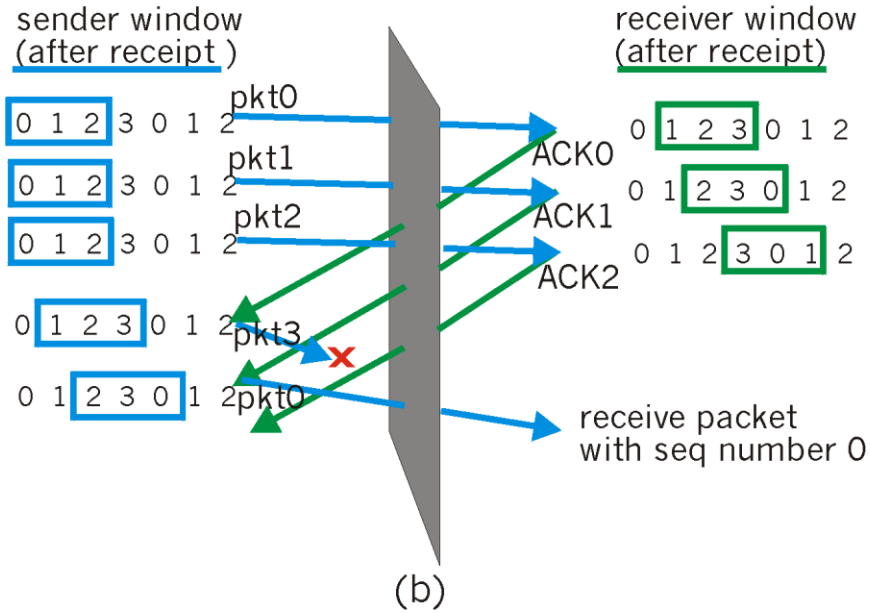
- seq #'s: 0, 1, 2, 3
- window size=3

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

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

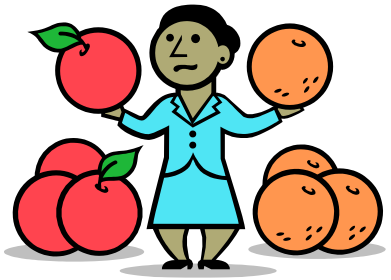






GBN vs. Selective Repeat

- Selective repeat is more complicated as it needs buffering at the receiver, but only retransmit packets required for retransmission

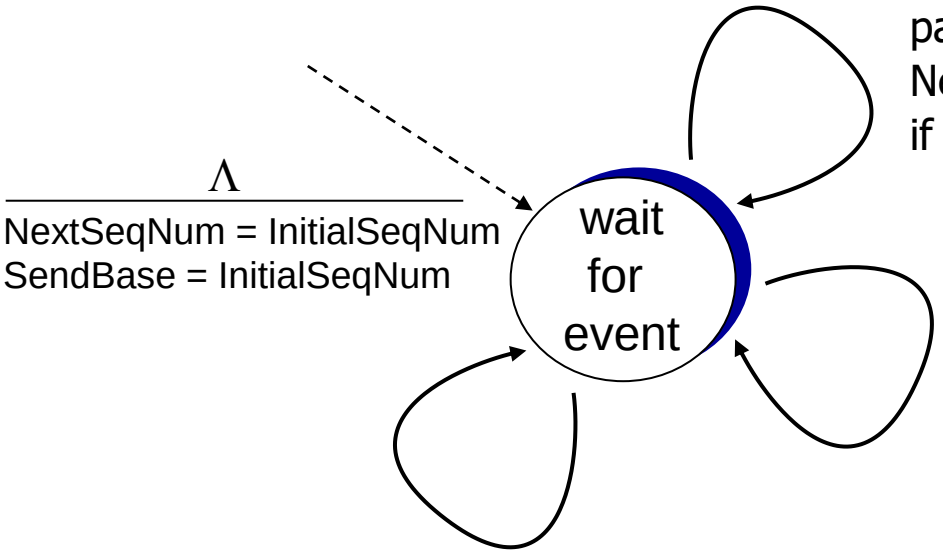


- GBN is simpler, but can lead to large number of unnecessary retransmission

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 (simplified)



data received from application above

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

timeout

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

ACK received, with ACK field value y

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

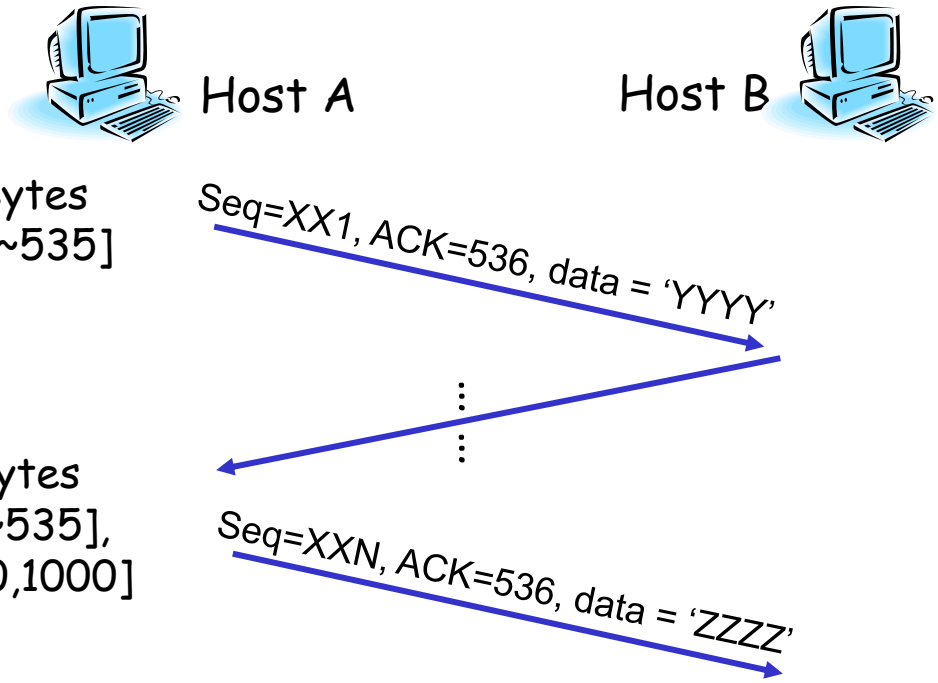
TCP Seq. #'s and ACKs

Seq. #'s:

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

ACKs:

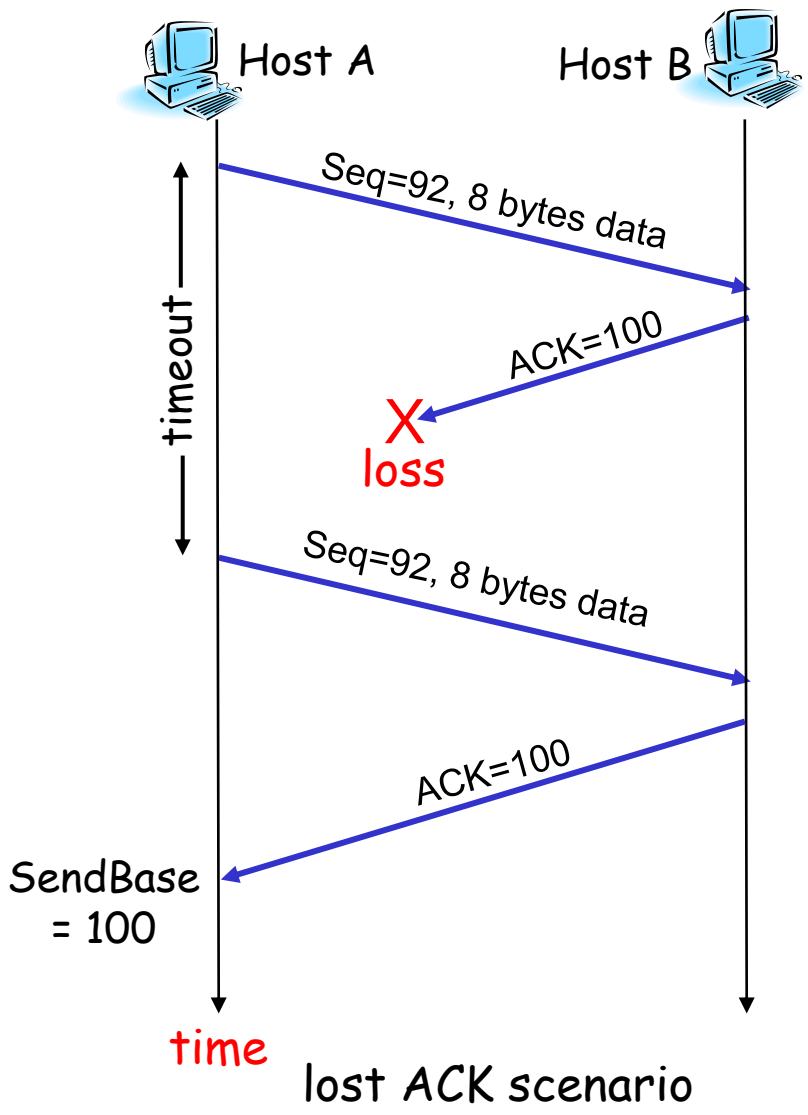
- Seq # of next byte expected from other side
- Cumulative ACK: TCP only acknowledge bytes up to the first missing byte in the stream



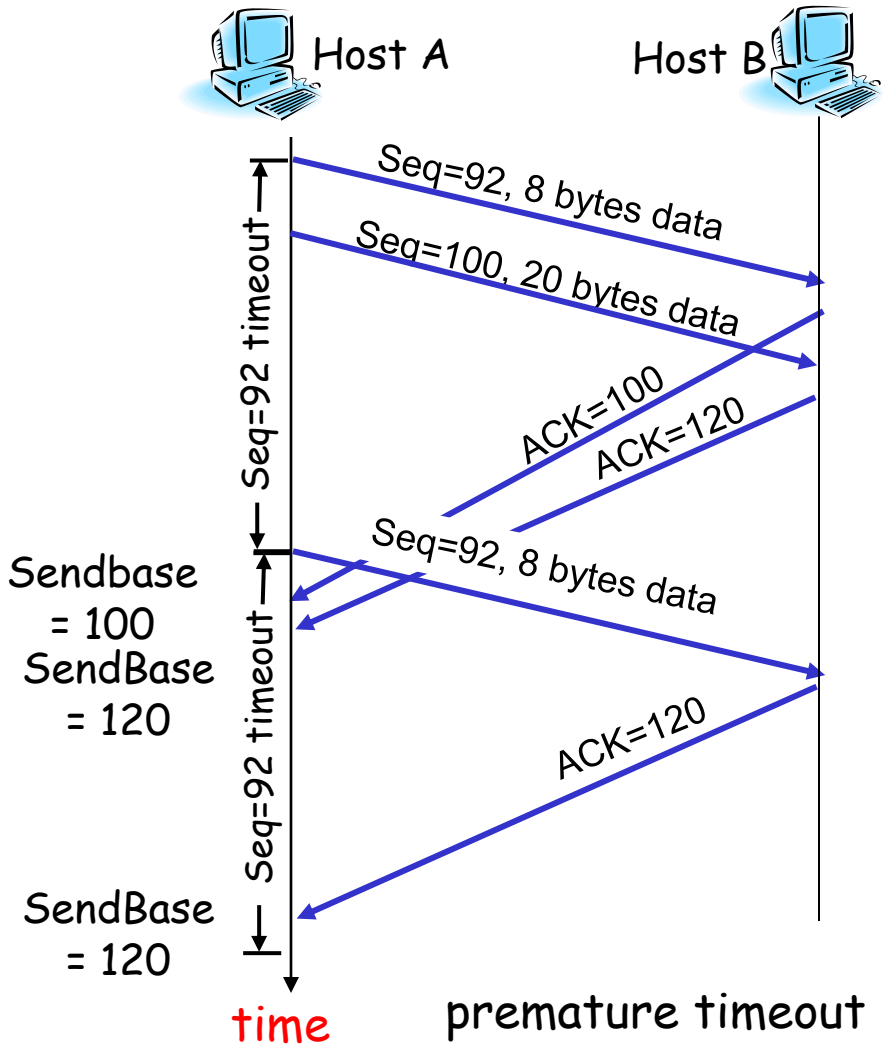
Q: how receiver handles out-of-order segments?

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

TCP: Retransmission Scenarios

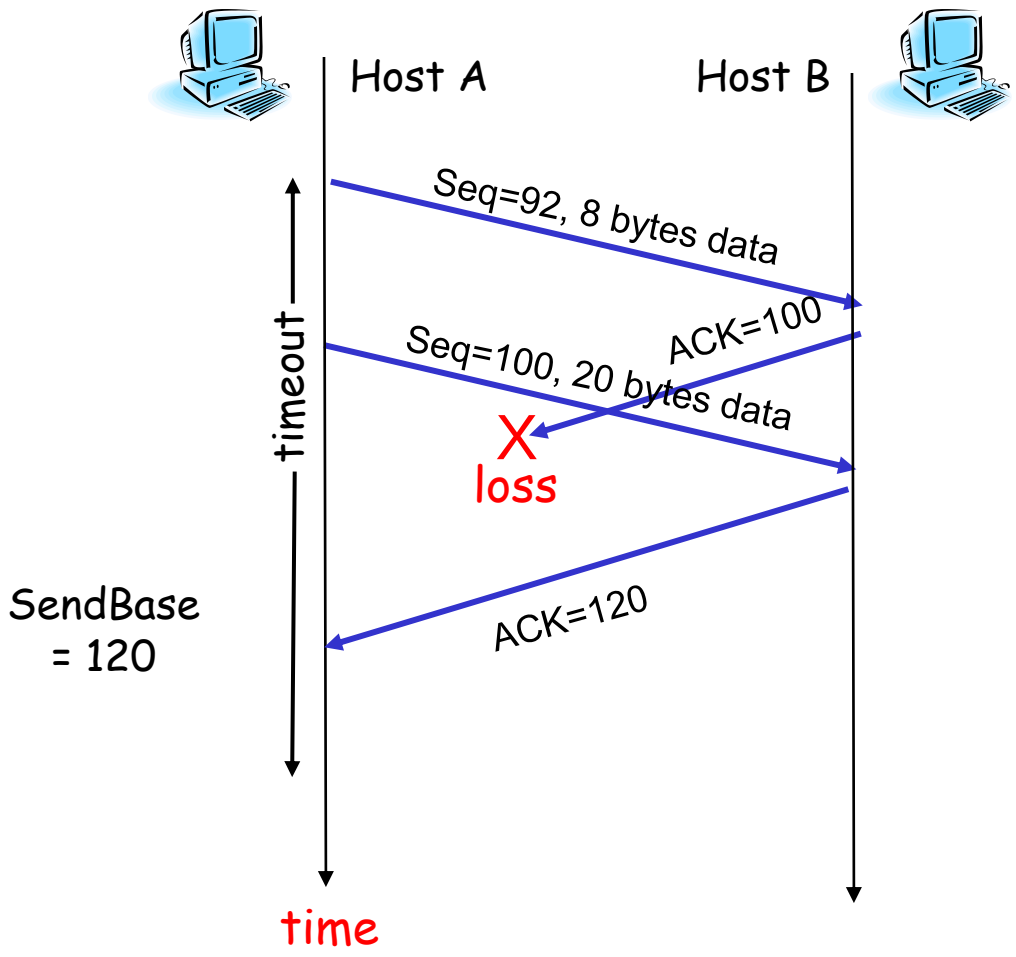


lost ACK scenario



premature timeout

TCP Retransmission Scenarios (more)



Cumulative ACK scenario

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

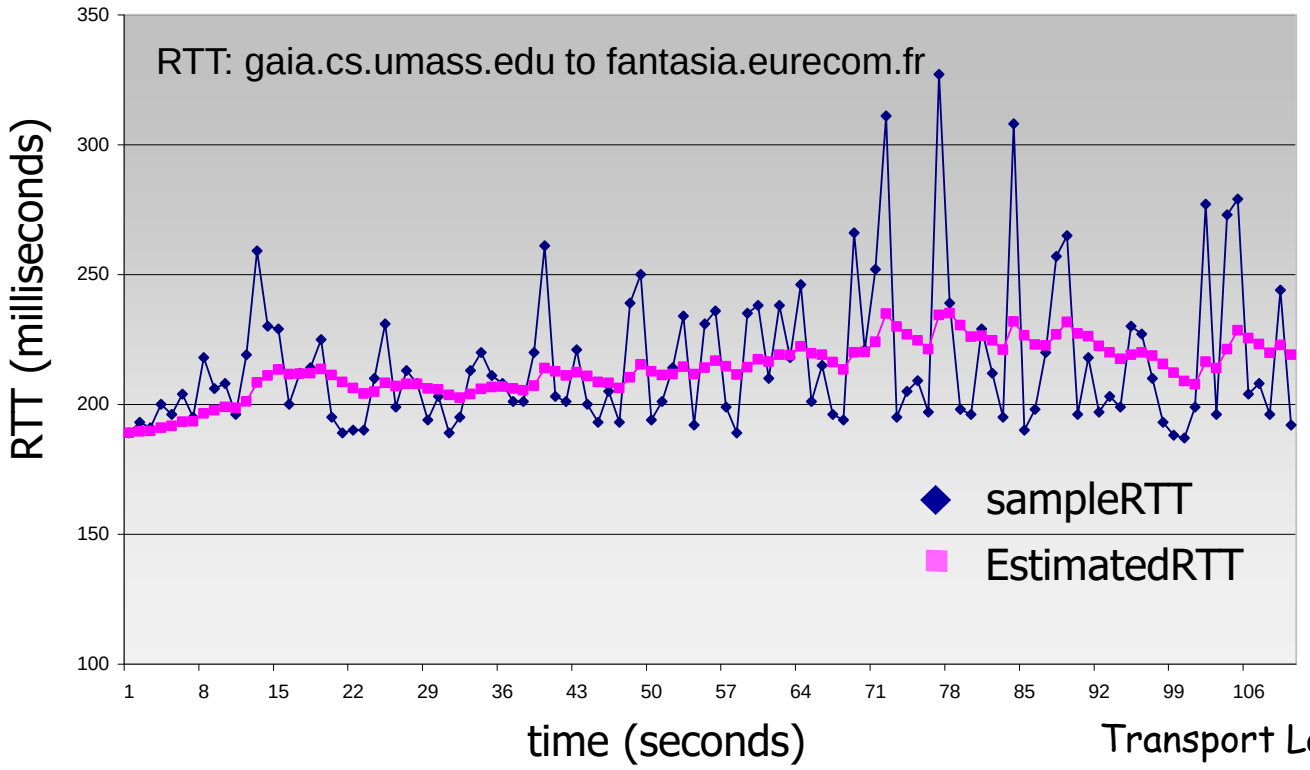
Q: how to estimate RTT?

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

TCP round trip time, timeout

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

- Exponential Weighted Moving Average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

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

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

`(typically, β = 0.25)`

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

estimated RTT

"safety margin"

More on Sender Policies

- Doubling the Timeout Interval
  - Used by most TCP implementations
  - If timeout occurs then, after retransmisison, Timeout Interval is doubled
  - Intervals grow exponentially with each consecutive timeout
- When Timer restarted because of (i) new data from above or (ii) ACK received, then Timeout Interval is reset as described previously using Estimated RTT and DevRTT.
- Limited form of Congestion Control

TCP fast retransmit

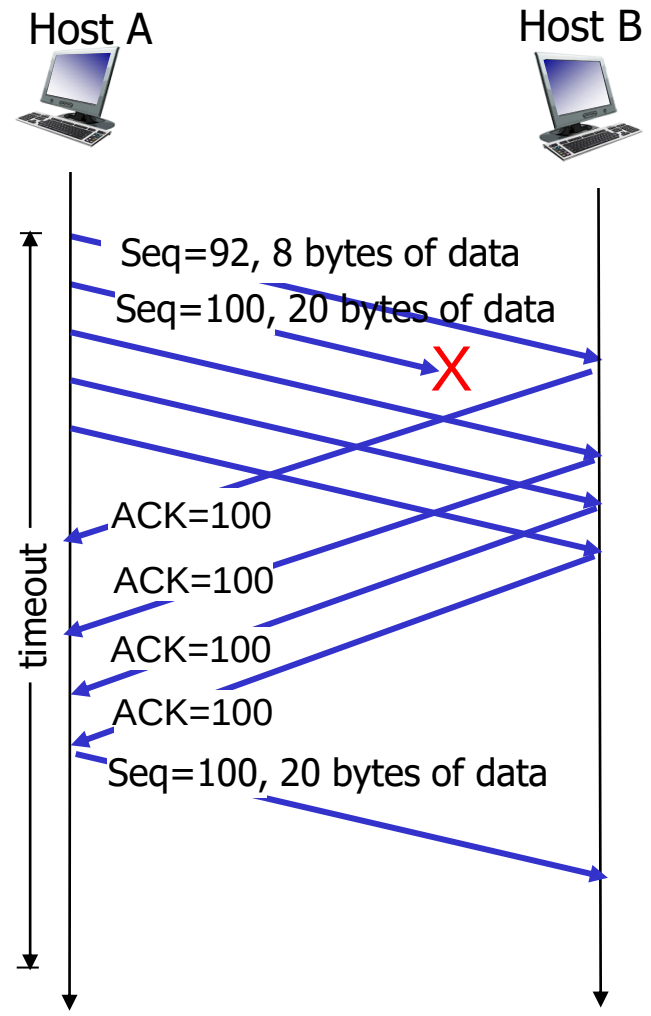
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives "triple duplicate ACKs" for same data, resend unacked segment with smallest seq #

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

TCP fast retransmit



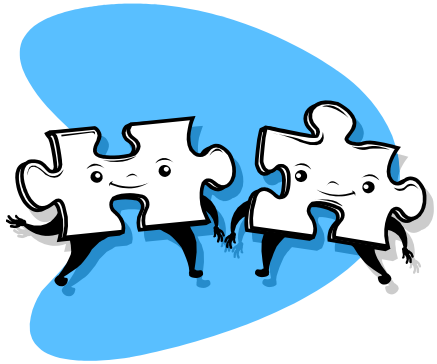
fast retransmit after sender receipt of triple duplicate ACK

TCP ACK Generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

TCP: GBN or Selective Repeat?

- Basic TCP looks a lot like GBN
- Many TCP implementations will buffer received out-of-order segments and then ACK them all after filling in the range
  - This looks a lot like Selective Repeat
- TCP is a hybrid



Exercises-3

1. A TCP connection has been established between host A and host B. host A sends three consecutive TCP segments to host B, including 300 bytes, 400 bytes and 500 bytes payload respectively. The sequence number of the third segment is 900. If host B receives only the first and third segments correctly, the confirmation sequence number sent by host B to host A is( )

(1)300

(2)500

(3)1200

(4)1400

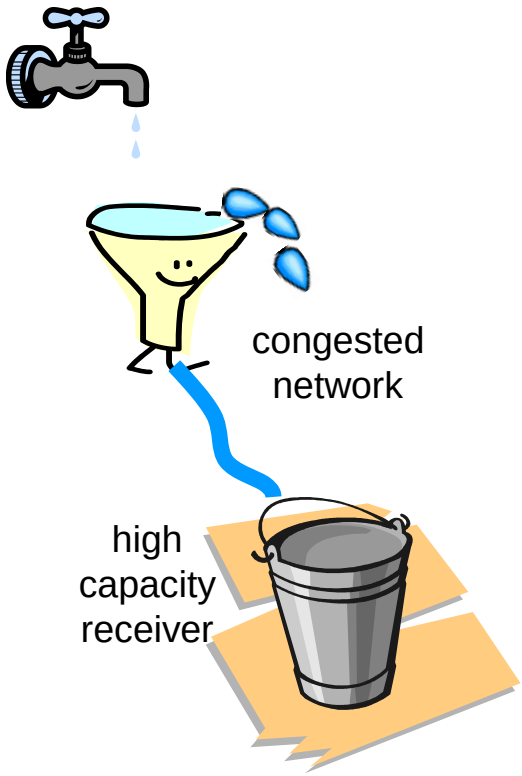
Chapter 6 Outline

- 6.1 Transport-layer services
- 6.2 Multiplexing and demultiplexing
- 6.3 Connectionless transport: UDP
- 6.4 Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Reliable data transfer
- 6.5 TCP congestion control

Principles of Congestion Control

Congestion:

- Informally: "too many sources sending too much data too fast for *network* to handle"
- Different from flow control!
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)
- A top-10 problem!



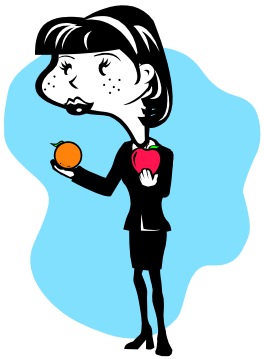
congestion control problem

Approaches towards Congestion Control

Two broad approaches towards congestion control:

End-end congestion control:

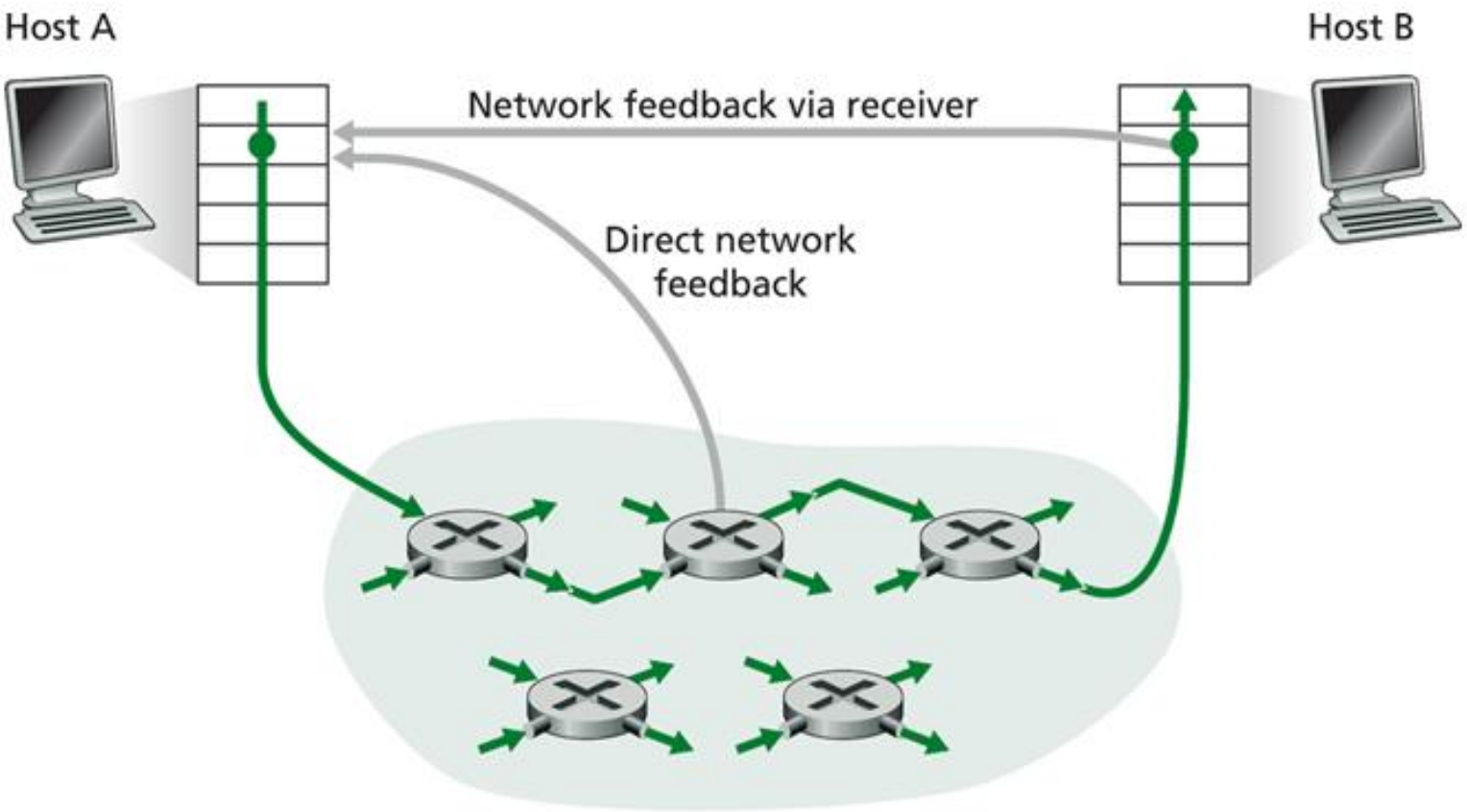
- No explicit feedback from network
- Congestion inferred from end-system observed loss, delay
- Approach taken by TCP



Network-assisted congestion control:

- Routers provide feedback to end systems
  - Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Explicit rate sender should send at

Two Ways for Providing Feedback



TCP Congestion Control

- End-to-end control (no network assistance)
- Transmission rate limited by congestion window size, **Congwin**, over segments
  - Congwin dynamically modified to reflect perceived congestion
- Tools are "similar" to flow control

  sender limits transmission using:

  LastByteSent-LastByteAcked ≤ CongWin

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate ACKs
- TCP sender reduces rate (**CongWin**) after loss event

TCP Congestion Control Algorithms

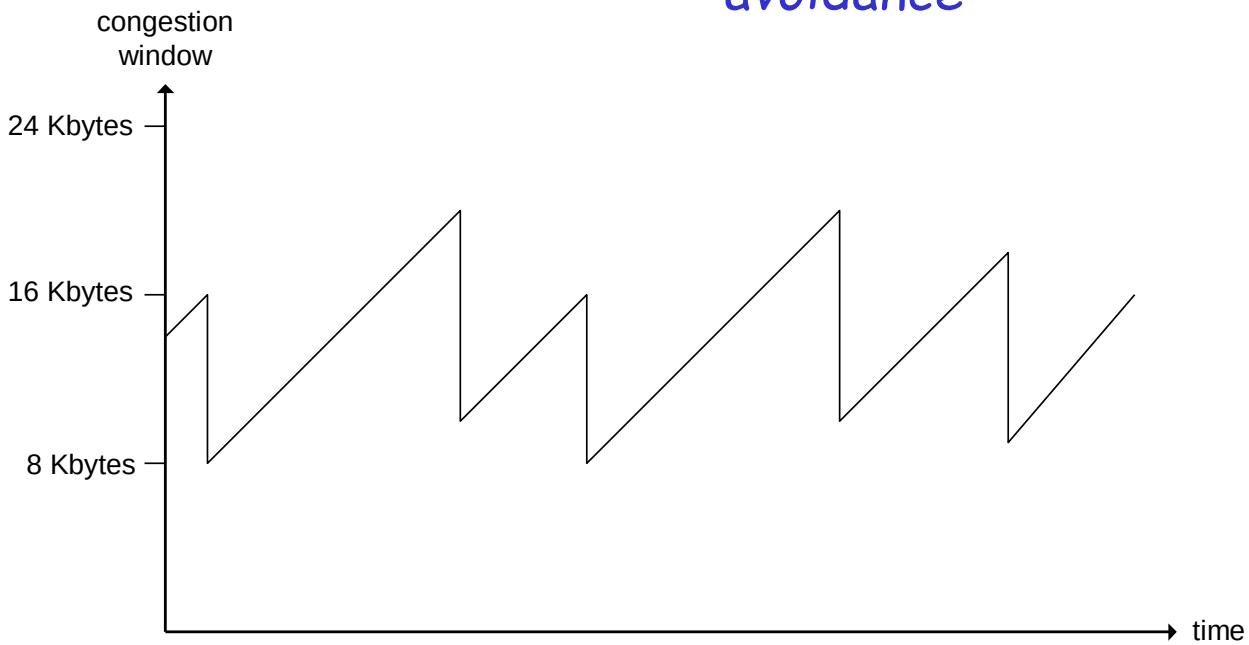
Three mechanisms:

- AIMD = *Additive Increase Multiplicative Decrease*
- Slow start = `CongWin` set to 1 and then grows exponentially
- Conservative after timeout events

TCP AIMD

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

Additive increase: increase **CongWin** by 1 MSS every RTT in the absence of loss events: *probing* also known as *congestion avoidance*



Long-lived TCP connection

TCP Slow Start

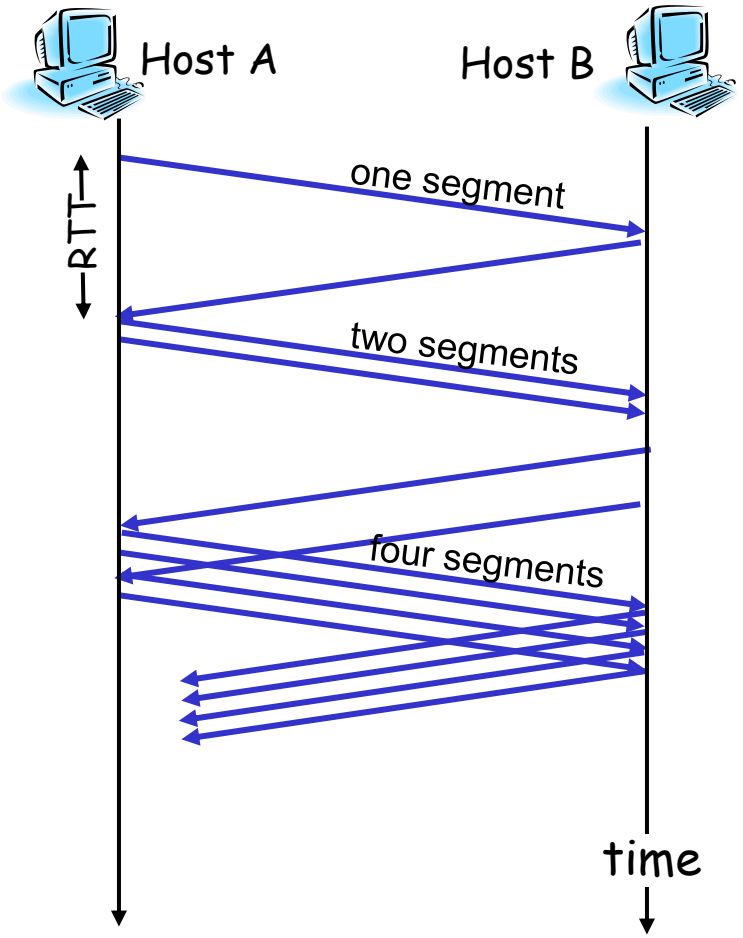
- When connection begins, **CongWin** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial rate = 20 kbps
- Available bandwidth may be >> MSS/RTT
  - Desirable to quickly ramp up to respectable rate

- When connection begins, increase rate exponentially fast until first loss event

Slow Start is NOT really Slow!

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - Double **CongWin** every RTT
  - Done by incrementing **CongWin** for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



- So Far
  - Slow-Start: ramps up exponentially
  - Followed by AIMD: sawtooth pattern
  - Q: When should the exponential increase switch to linear?

- Reality (TCP Reno)
  - Introduce new variable **threshold:** initially very large
  - Slow-Start exponential growth stops when reaches **threshold** and then switches to AIMD
  - Two different types of loss events
    - 3 dup ACKS: cut **CongWin** in half and set **threshold=CongWin** (now in standard AIMD)
    - Timeout: set **threshold=CongWin/2**, **CongWin=1** and switch to Slow-Start

- Reason for treating 3 dup ACKS differently than timeout is that:
  - 3 dup ACKs indicates network capable of delivering some segments while
  - Timeout before 3 dup ACKs is "more alarming"

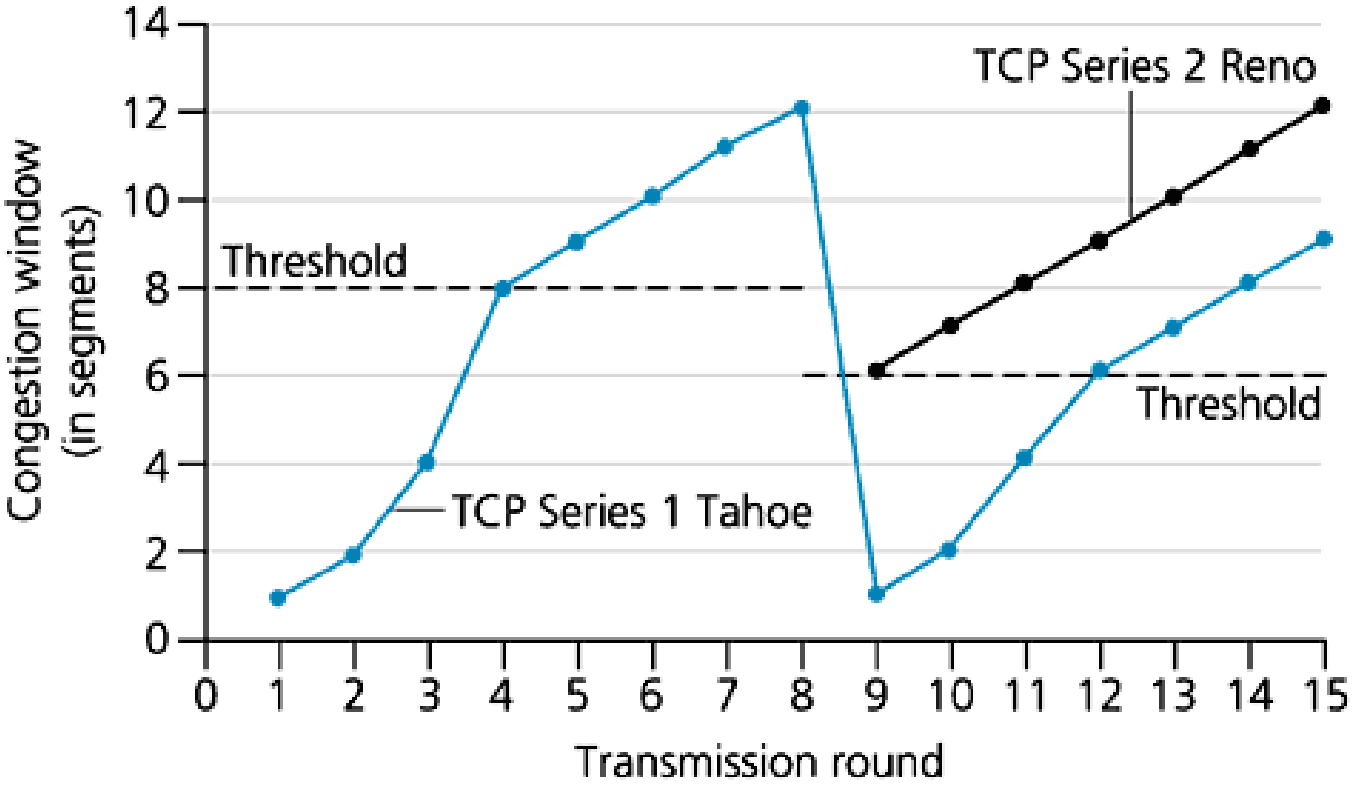
- Note that older protocol, TCP Tahoe, treated both types of loss events the same and **always** goes to slow-start with Congwin=1 after a loss event

- TCP Reno's skipping of the slow start for a 3-DUP-ACK loss event is known as fast-recovery

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** *(only in TCP Reno)*
- When timeout occurs, **Threshold** set to **CongWin/2** and **CongWin** is set to 1 MSS *(TCP Tahoe does this for 3 Dup Acks as well)*

The Big Picture



TCP Sender Congestion Control

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

Exercises-4

1. A TCP connection always sends TCP segments with 1KB MSS. The sender has enough data to send. A timeout event occurs when the congestion window is 16KB. If the transmission of TCP segments in the next four RTTS is successful, when all TCP segments sent in the fourth RTT receive positive responses, the size of the congestion window is( )

A. 7KB
B. 8KB
C. 9KB
D. 16KB

Chapter 6: Summary

- Principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Congestion control
- Instantiation and implementation in the Internet
  - UDP
  - TCP

Homework

- P538:14,15
- P322:27
- P323:28
- P326:42,43