

Chapter 3

Transport Layer

A note on the use of these ppt slides:

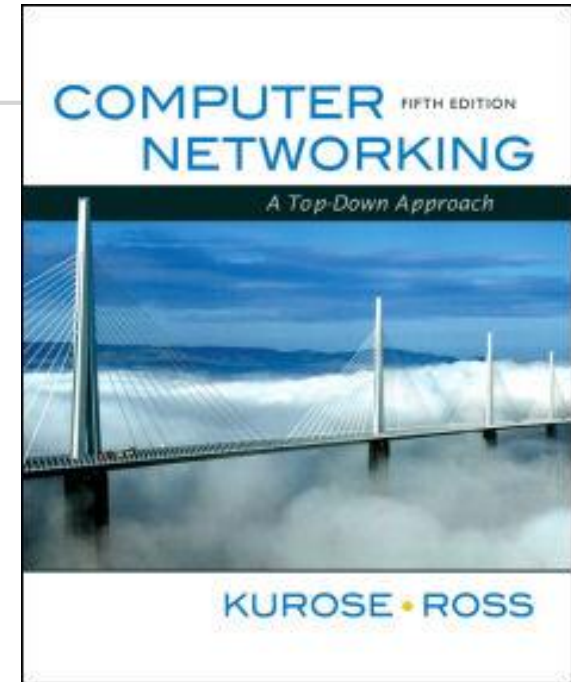
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- 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.

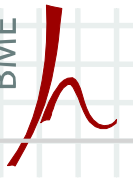
Thanks and enjoy! JFK/KWR

All material copyright 1996-2010

J.F Kurose and K.W. Ross, All Rights Reserved



**Computer Networking: A
Top Down Approach
Featuring the Internet,
5th edition.
Jim Kurose, Keith Ross
Pearson Addison-Wesley,
2009.**



Chapter 3: Transport layer

Our goals:

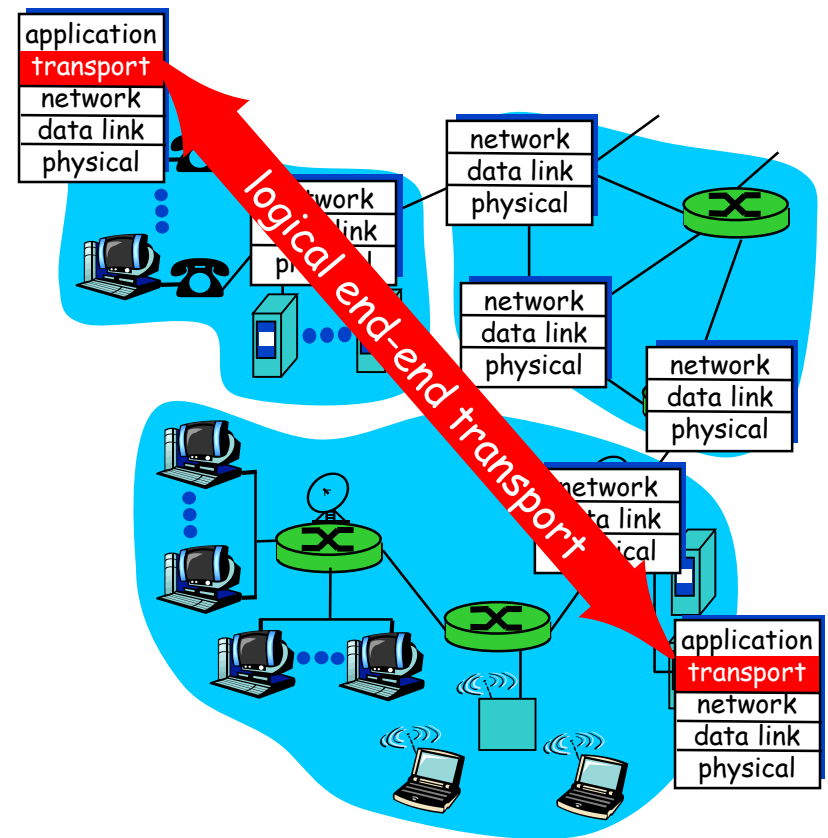
- Understand principles behind transport layer services:
 - multiplexing/demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- Learn about transport layer protocols in the Internet:
 - UDP: connectionless transport
 - TCP: connection-oriented transport
 - TCP congestion control

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

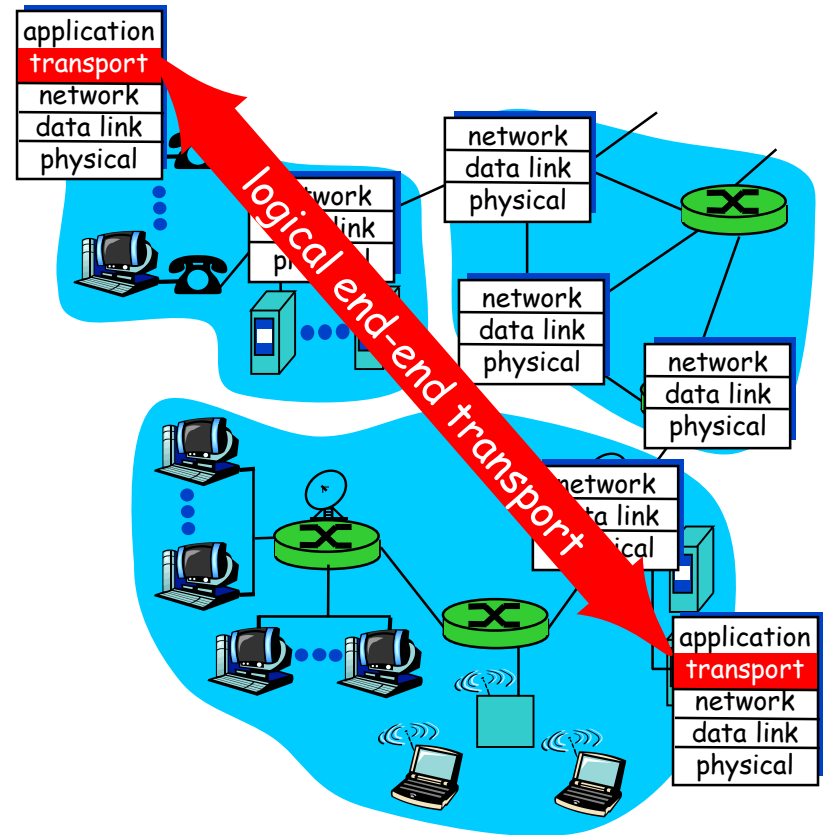
Household analogy:

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: controlling the flow of data when congestion has actually occurred
 - flow control: to prevent a fast sender from outrunning a slow receiver
 - 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

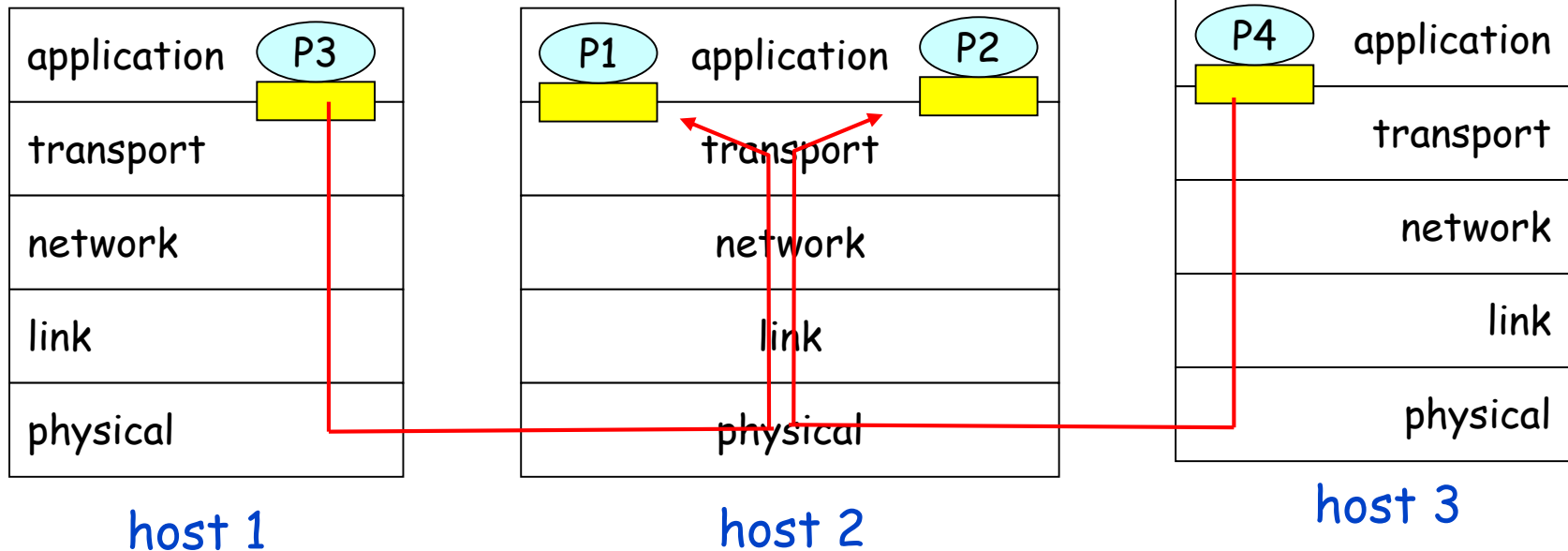
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

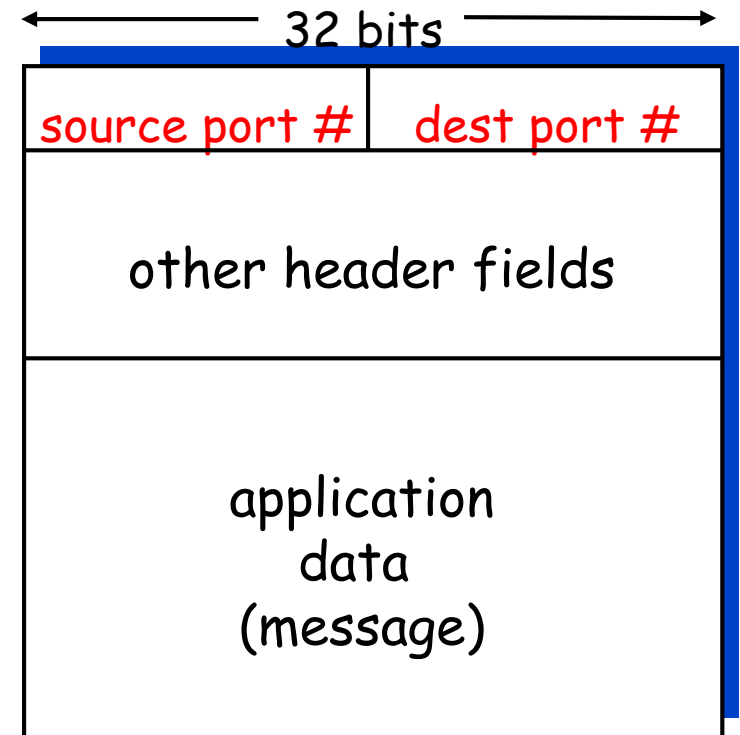
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

 = socket  = process

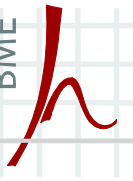


How demultiplexing works

- 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
- Host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format



Connectionless demultiplexing

- Create sockets with port numbers:

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

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

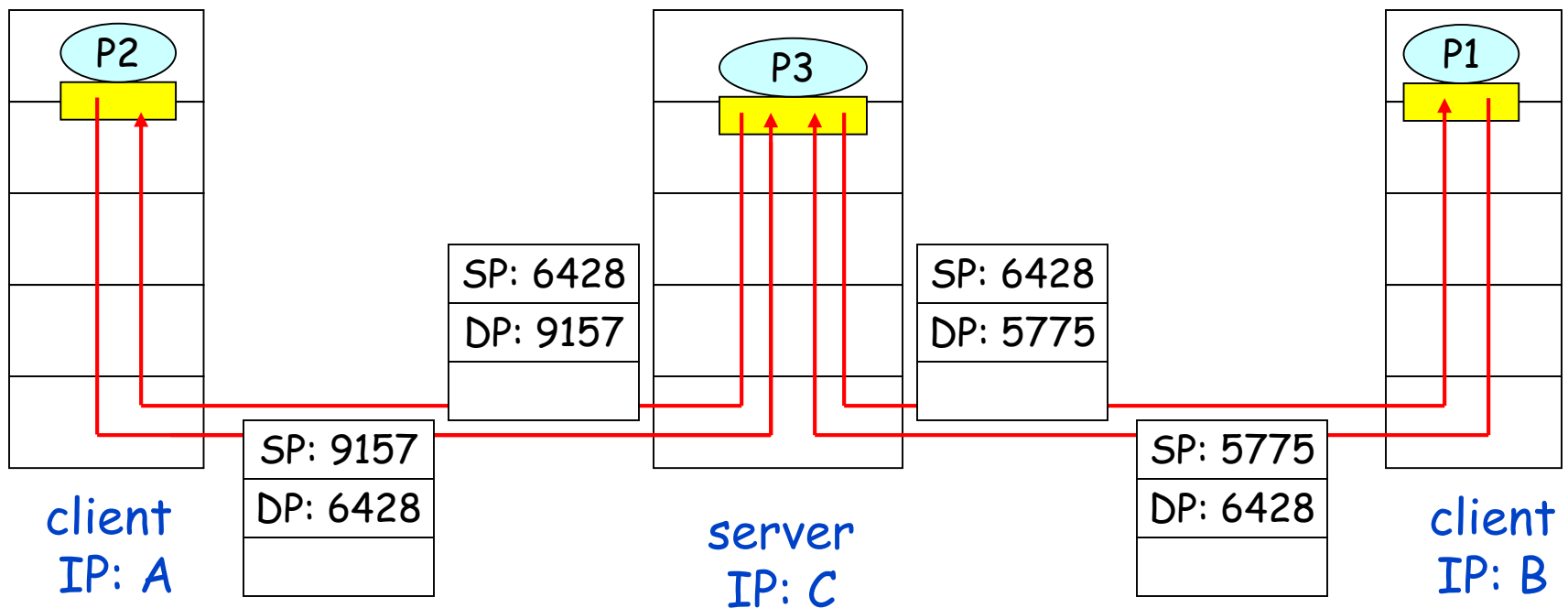
- UDP socket identified by two-tuple:

(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'd)

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

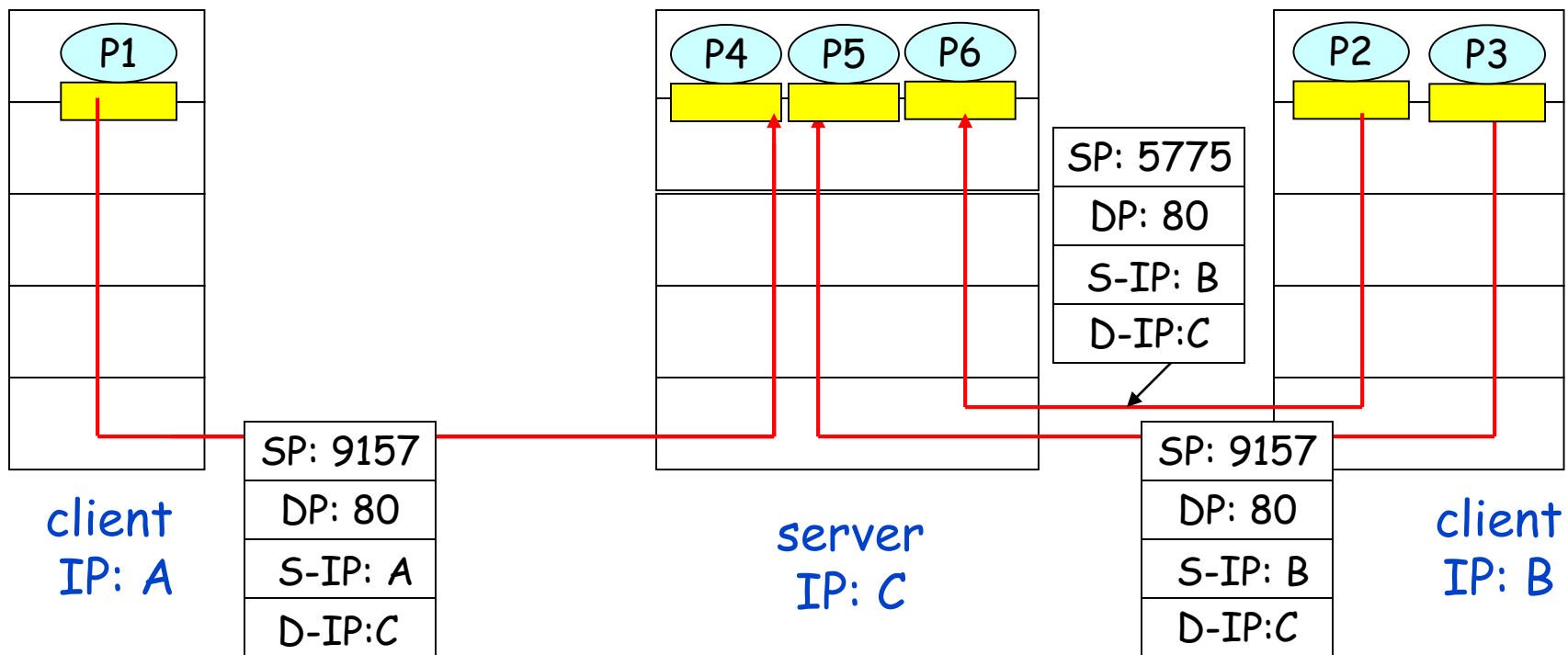


SP provides "return address"

Connection-oriented demux

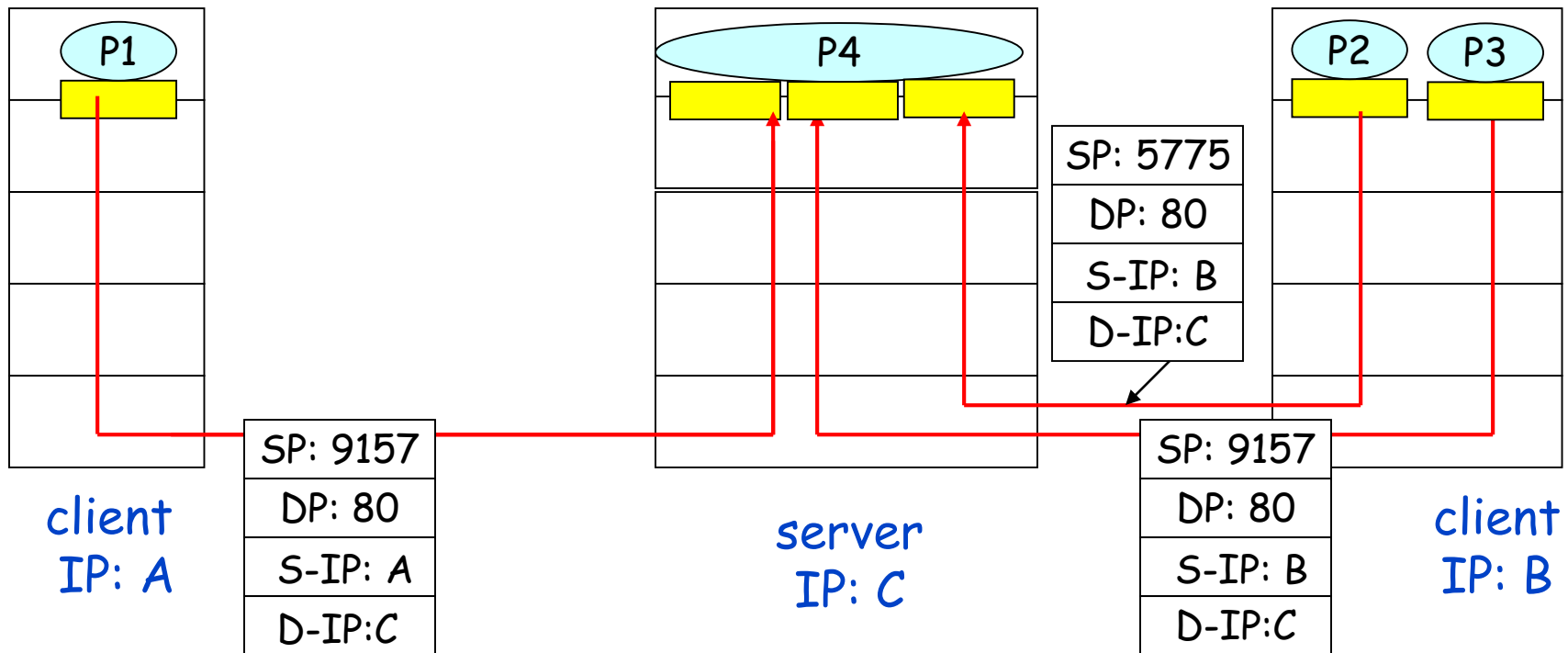
- TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- Recv host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets
 - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
 - non-persistent HTTP will have different socket for each request

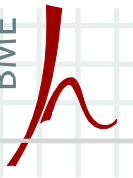
Connection-oriented demux (cont'd)



Threaded web server

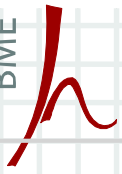
- There is not always a one-to-one correspondence between sockets and processes
 - one process may create a new thread (a „lightweight subprocess”) with a new connection socket for new clients





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]

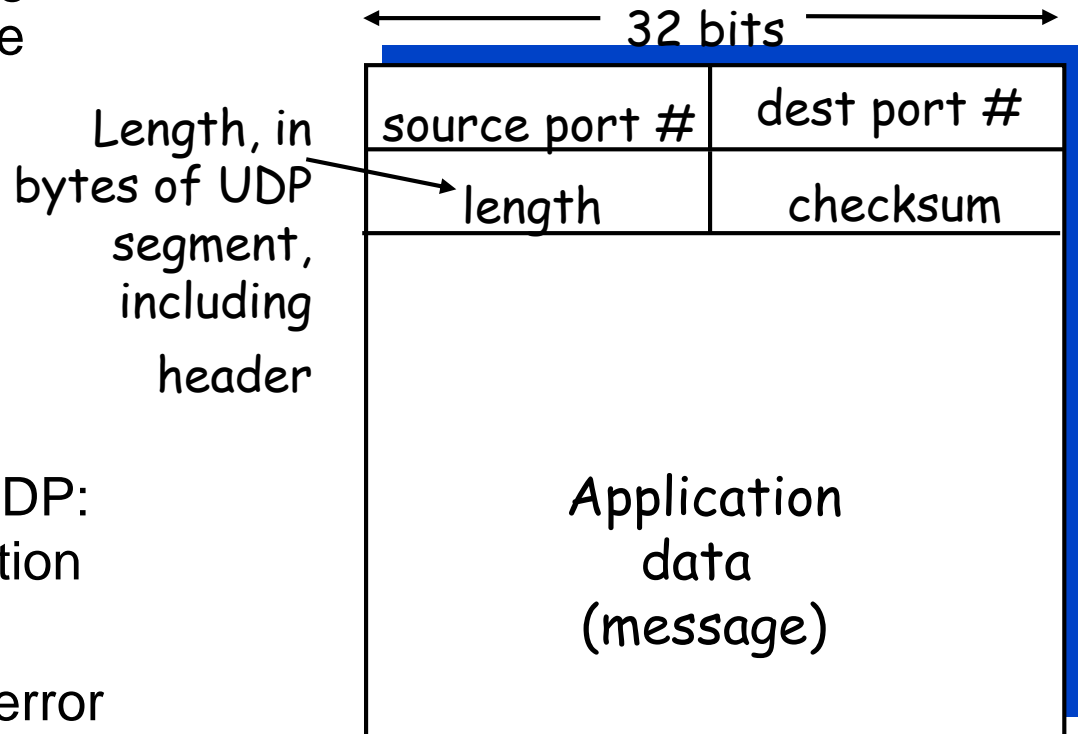
- “No frills,” “bare bones” Internet transport protocol
- “Best effort” service, UDP segments may be:
 - lost
 - delivered out of order to app
- *Connectionless*
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others

Why is there a UDP?

- 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 that are
 - 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 (one’s complement* sum) of segment contents
- Sender puts checksum value into UDP checksum field

*the value obtained by inverting all the bits in the binary representation of the number

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?

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

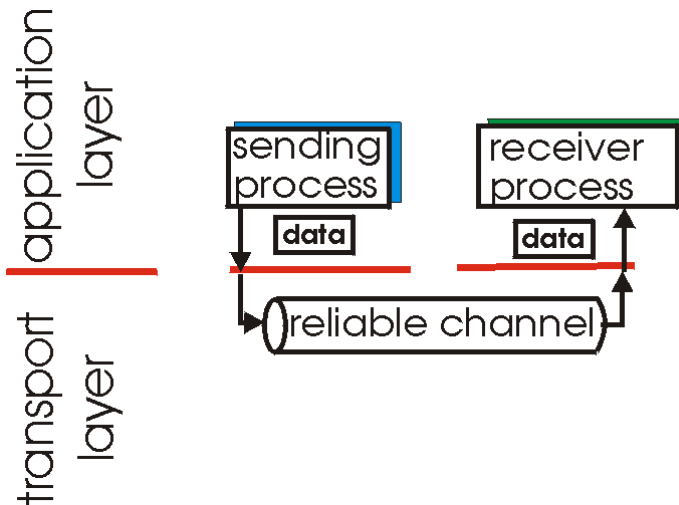
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																	
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
<hr/>																	
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0	
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1	

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!

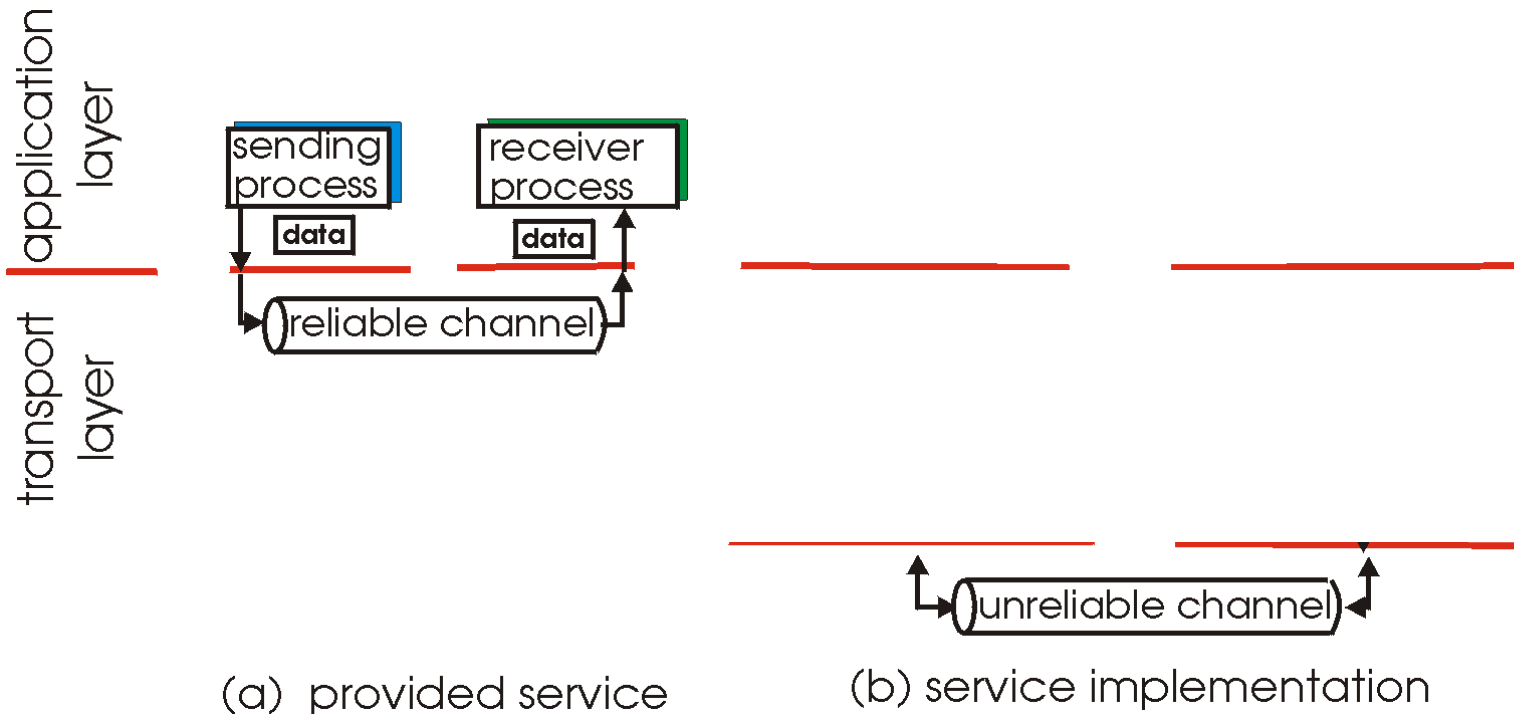


(a) provided service

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

Principles of reliable data transfer

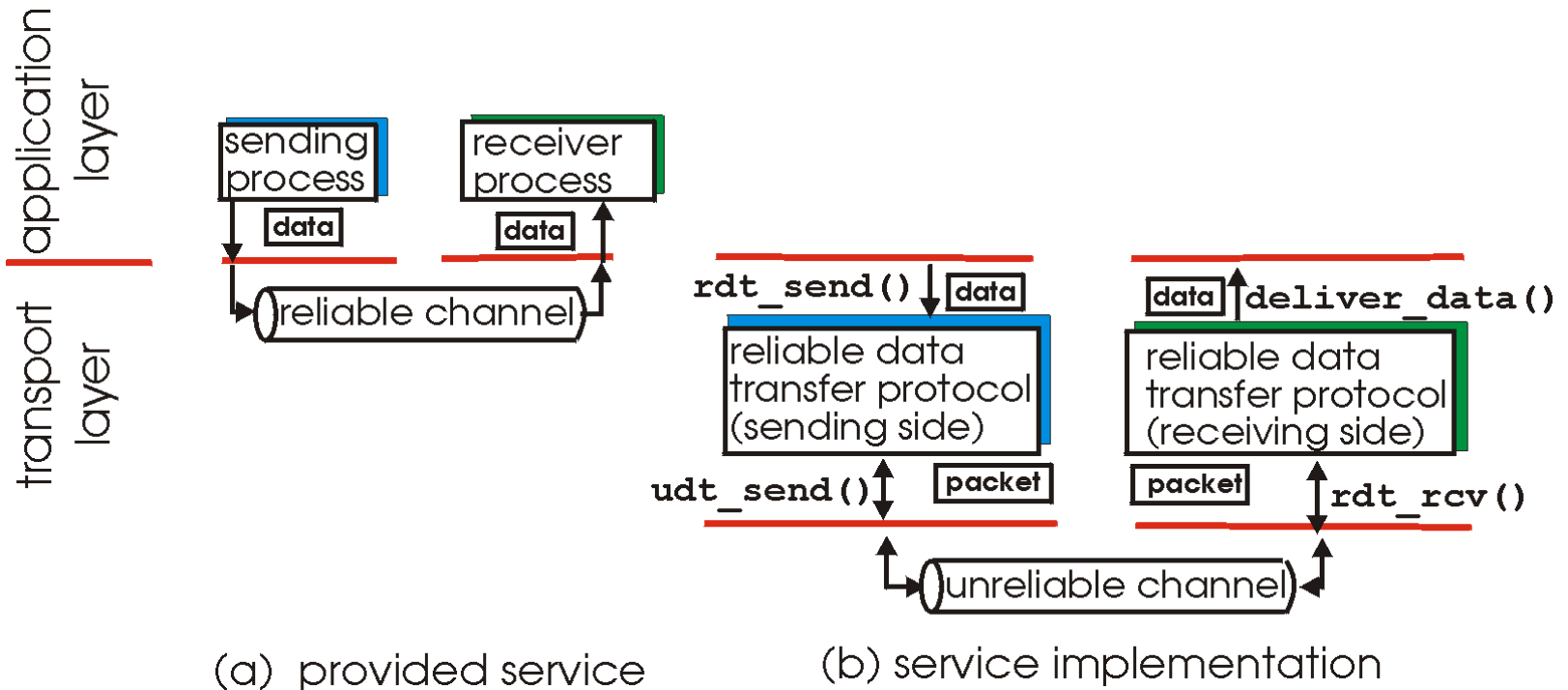
- important in app., transport, link layers
- top-10 list of important networking topics!



- 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!

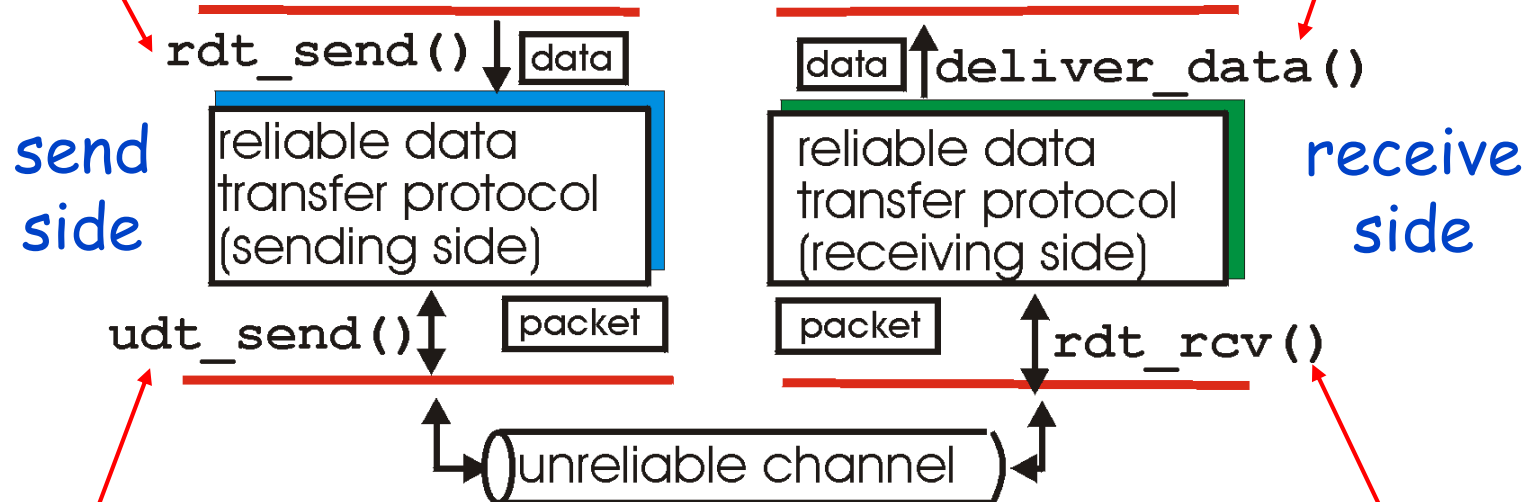


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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



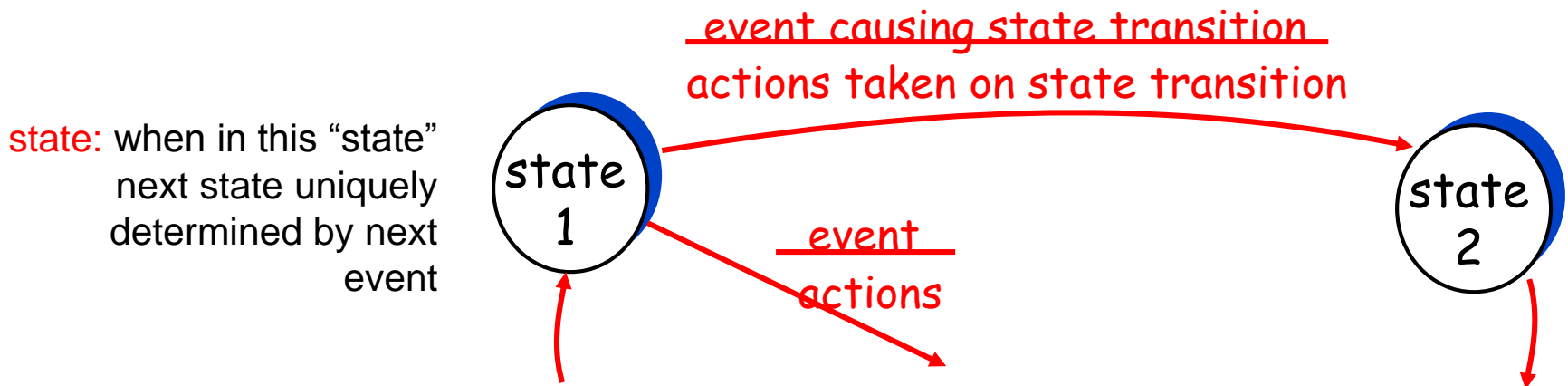
udt_send() : called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: Getting started

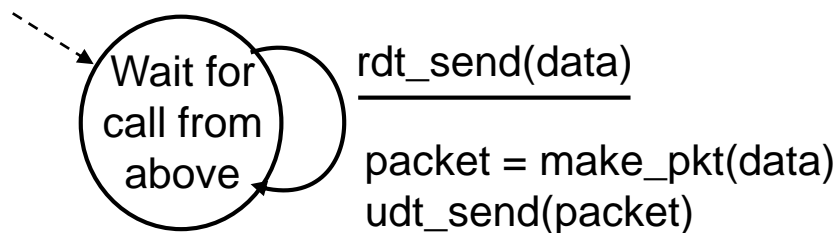
We'll:

- 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

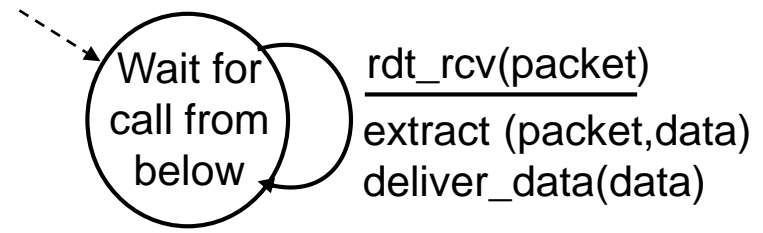


Rdt1.0: Reliable transfer over a reliable channel

- 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

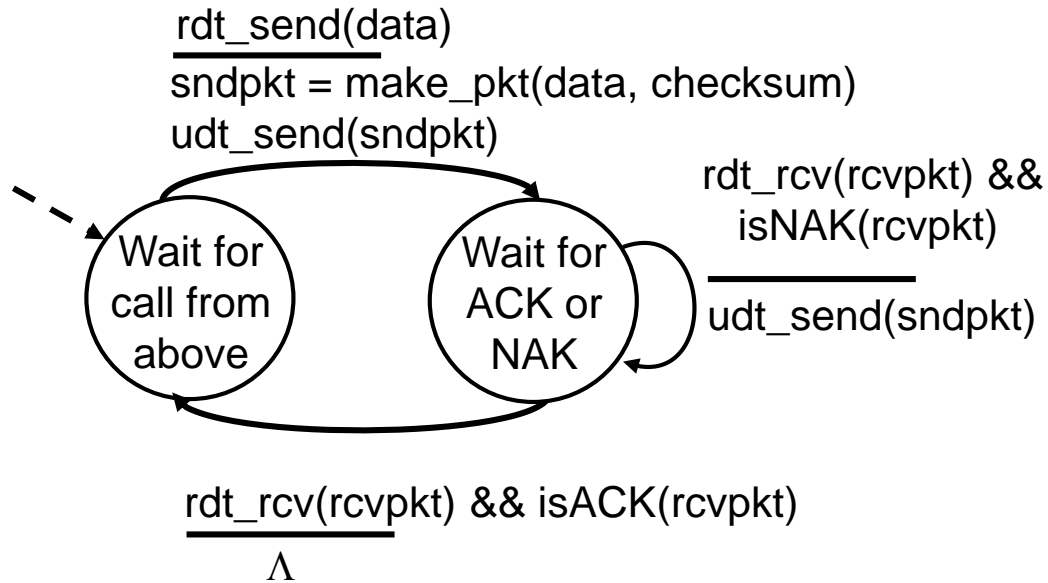


receiver

Rdt2.0: Channel with bit errors

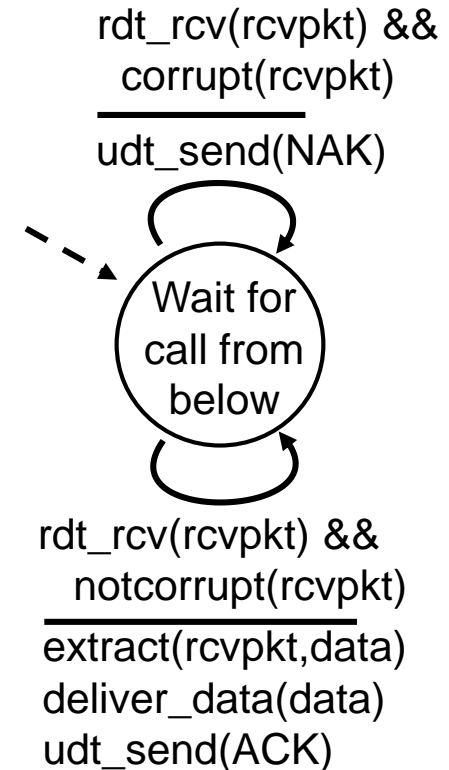
- Underlying channel may flip bits in packet
 - Checksum to detect bit errors
- *The question: how to recover from errors?*
 - *Acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
 - *Negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
 - 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

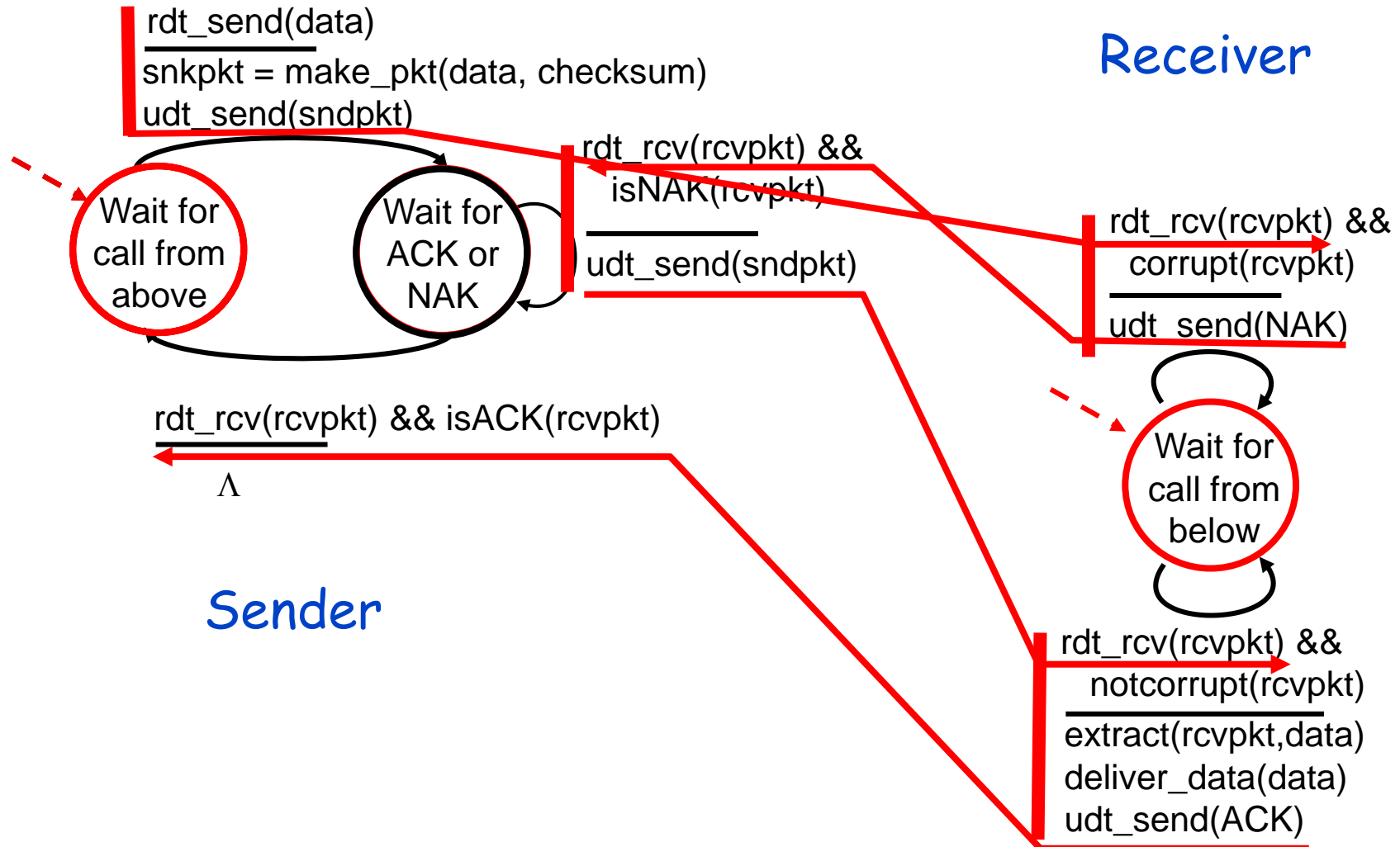


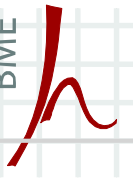
Sender

Receiver



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

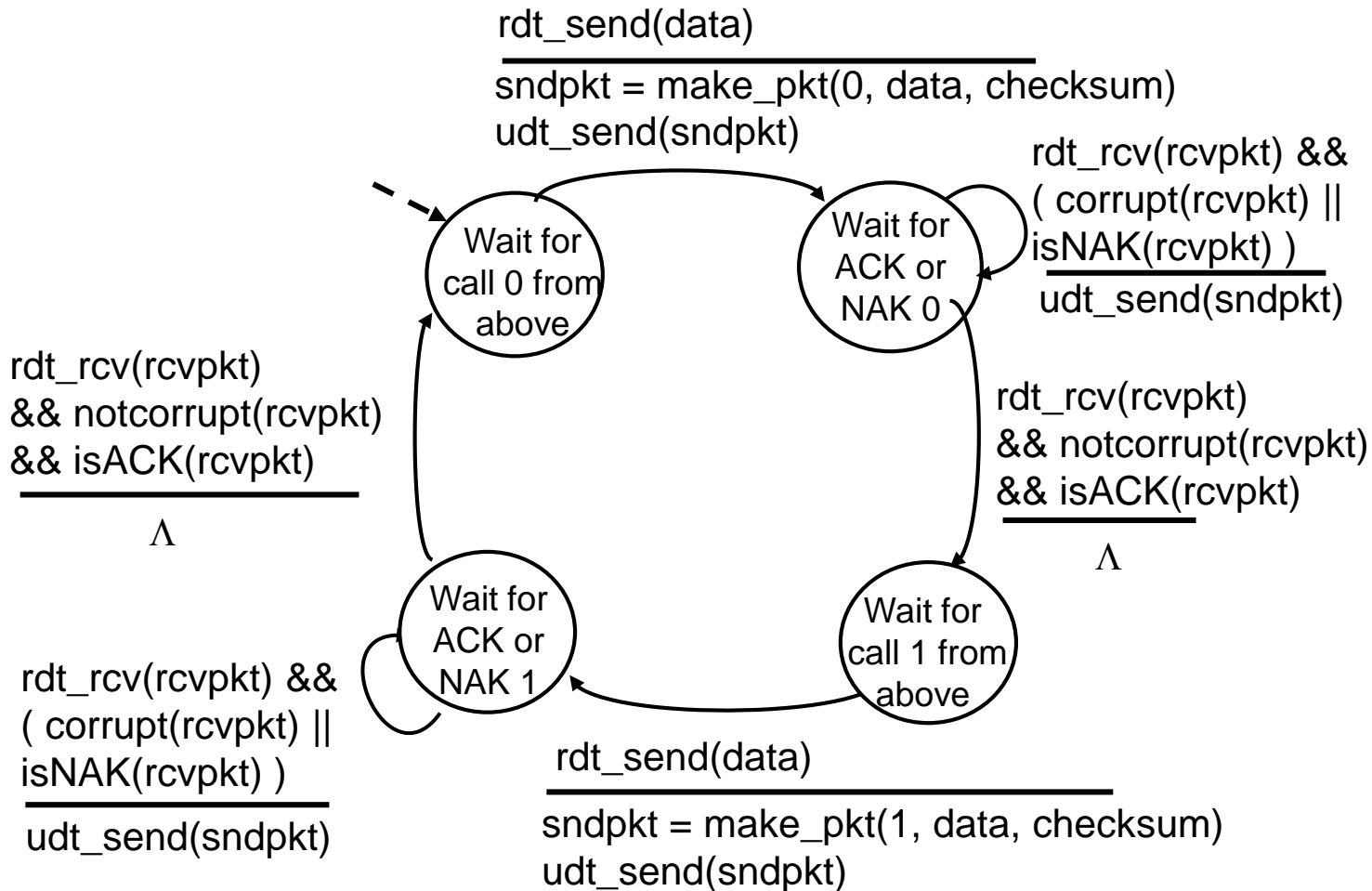
Handling duplicates

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

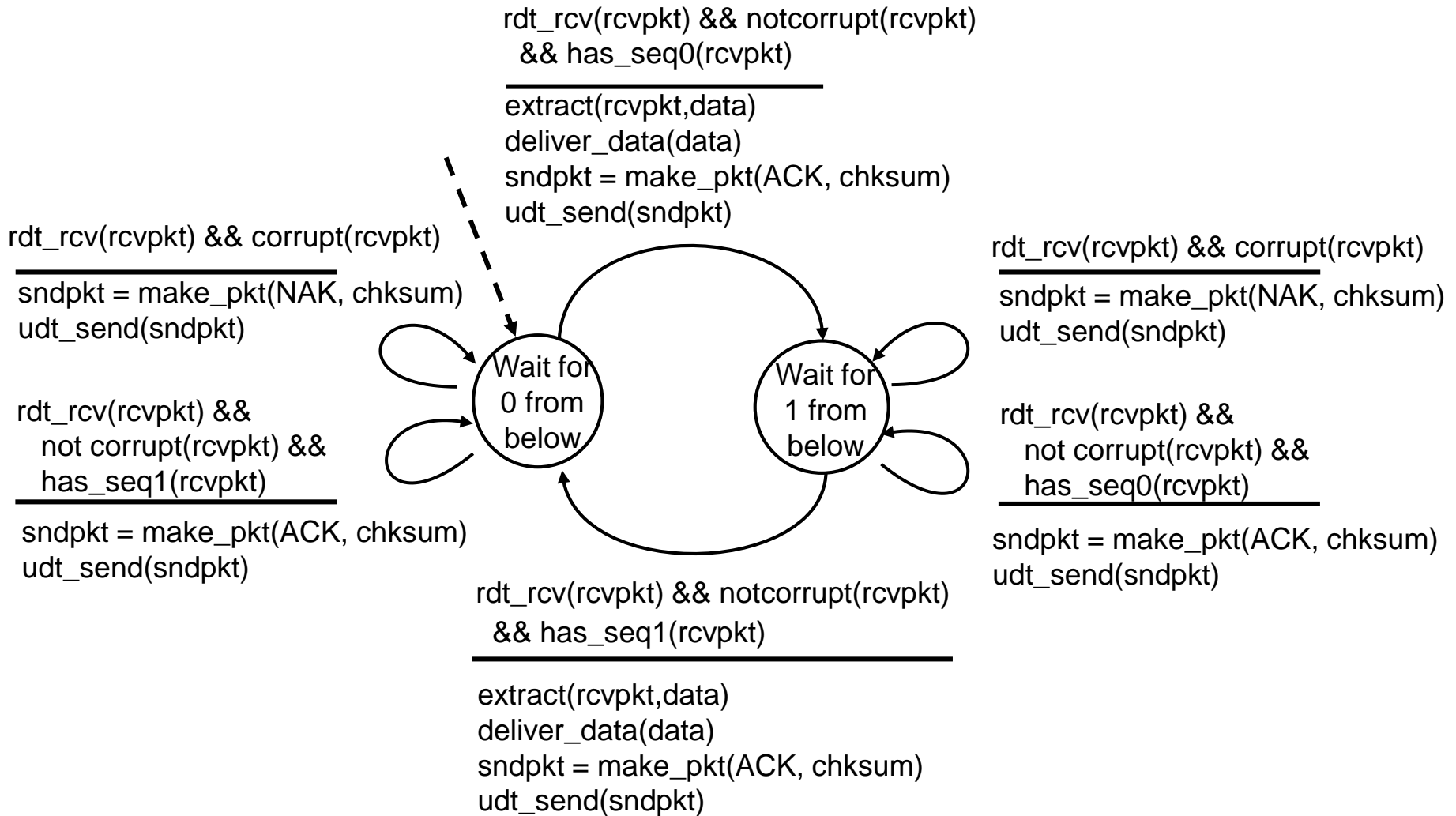
stop and wait

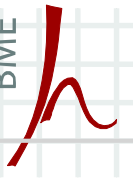
Sender sends one packet, then waits for receiver response

Rdt2.1: Sender, handles garbled ACK/NAKs



Rdt2.1: Receiver, handles garbled ACK/NAKs





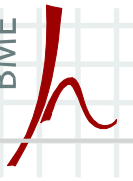
Rdt2.1: Discussion

Sender:

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

Receiver:

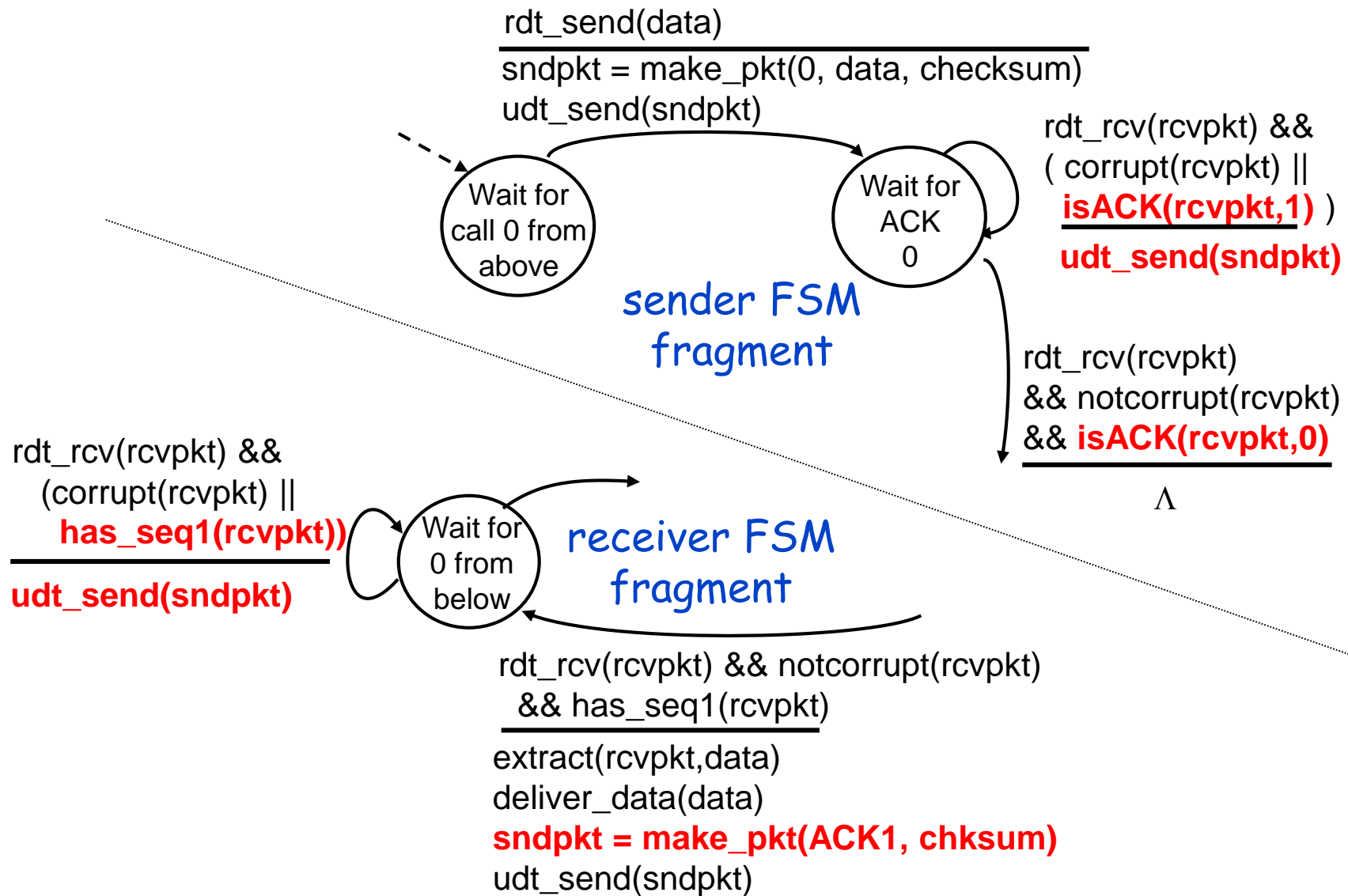
- Must check if received packet is duplicate
 - state indicates whether 0 or 1 is expected pkt seq #
- Note: receiver can *not* know if its last ACK/NAK received OK at sender

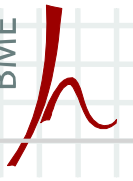


Rdt2.2: A NAK-free protocol

- 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

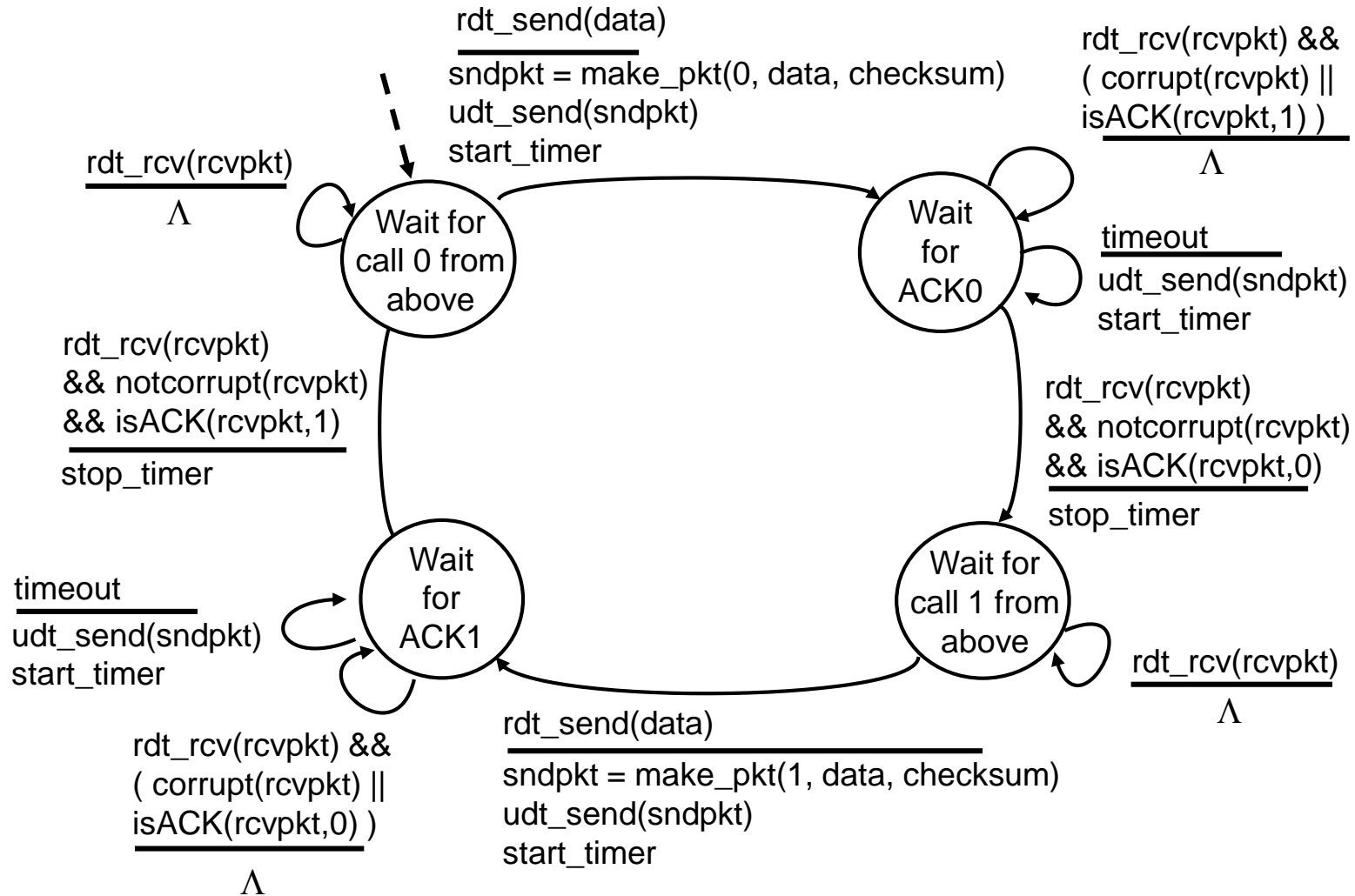
New assumption: underlying channel can also lose packets (data or ACKs)

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

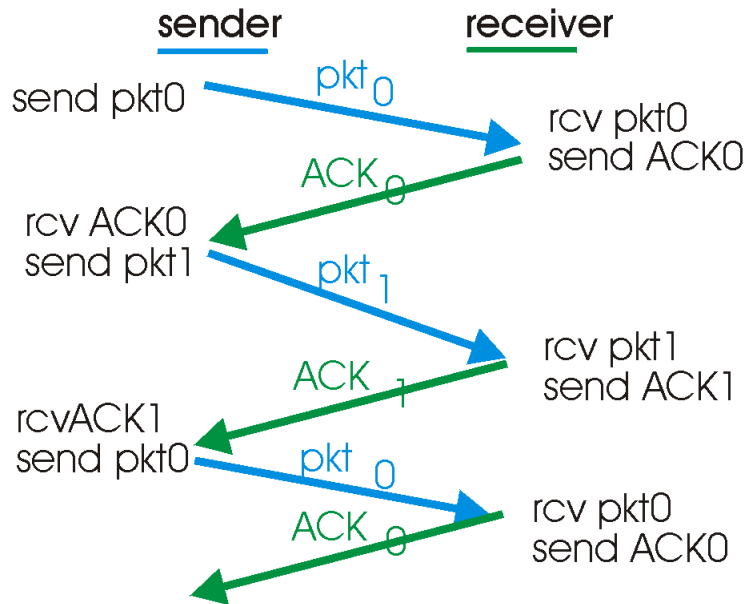
Approach: sender waits “reasonable” amount of time for ACK

- 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

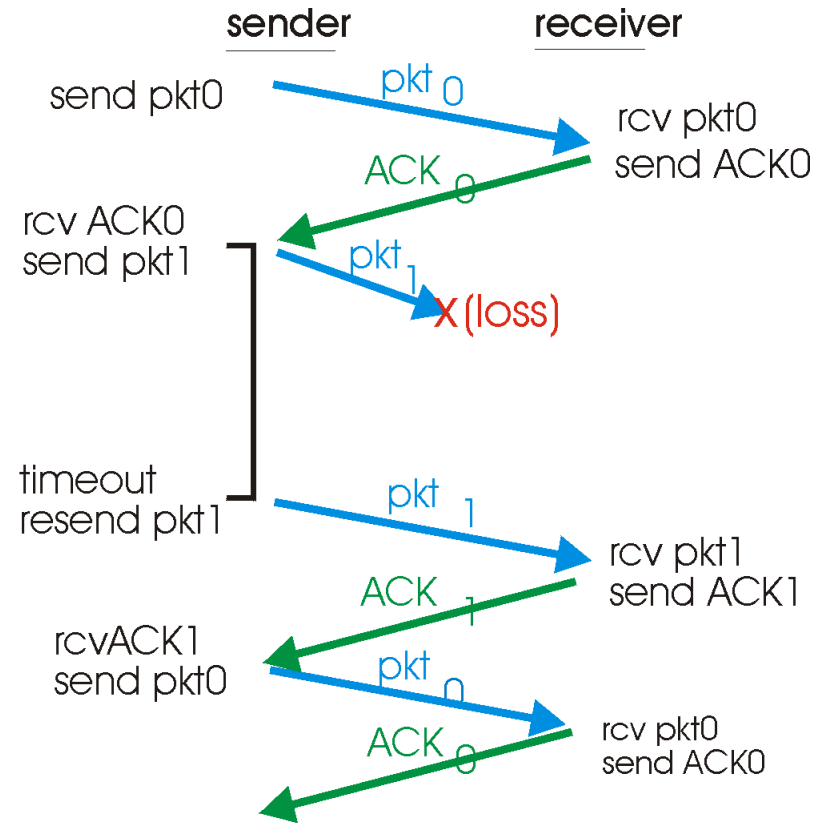
Rdt3.0 sender



Rdt3.0 in action

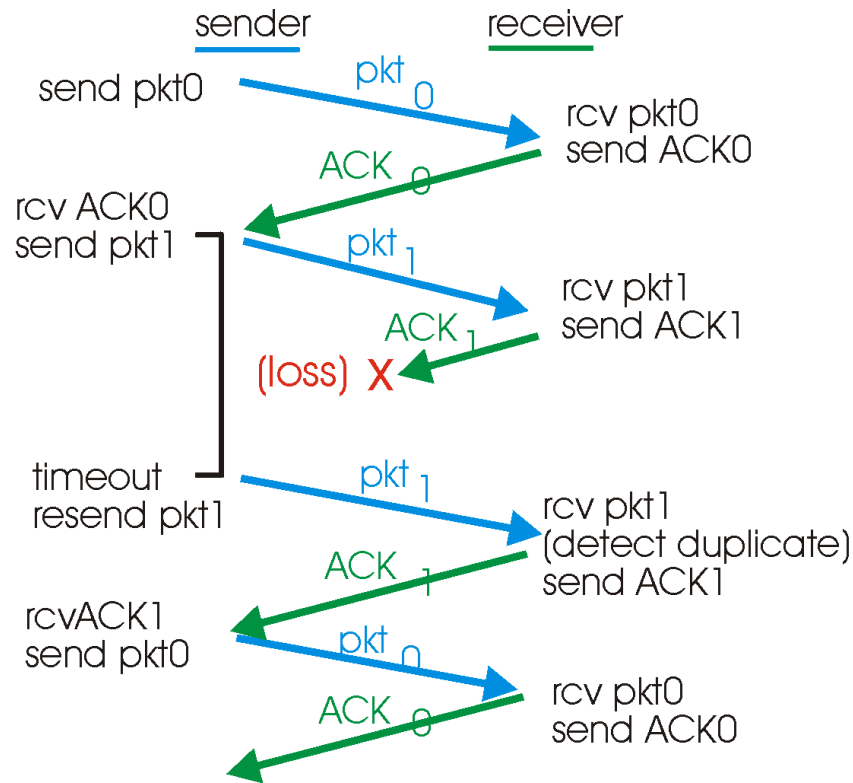


(a) operation with no loss

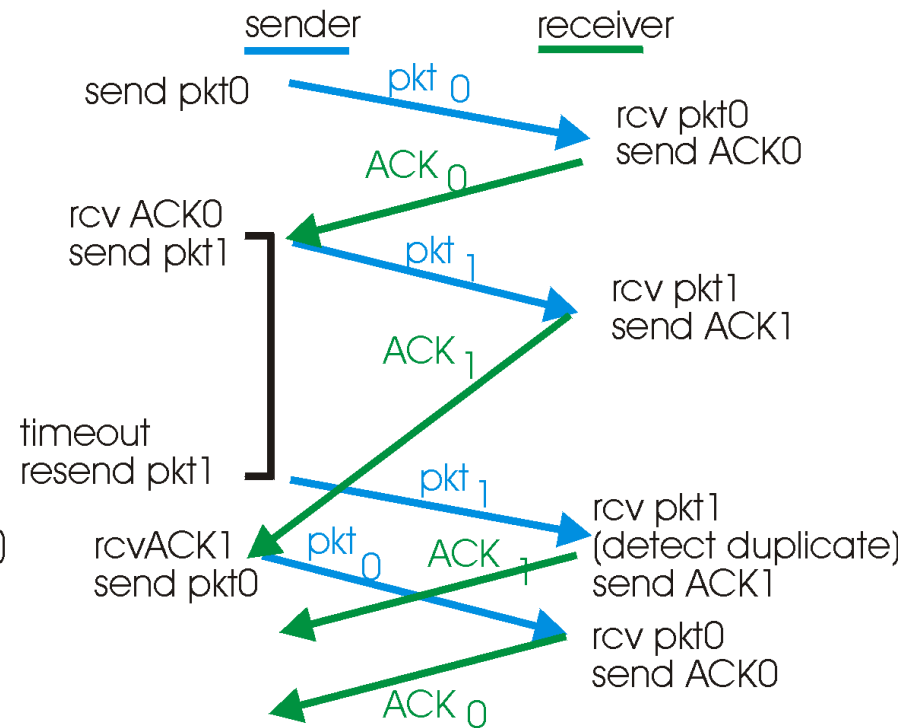


(b) lost packet

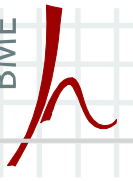
Rdt3.0 in action



(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:

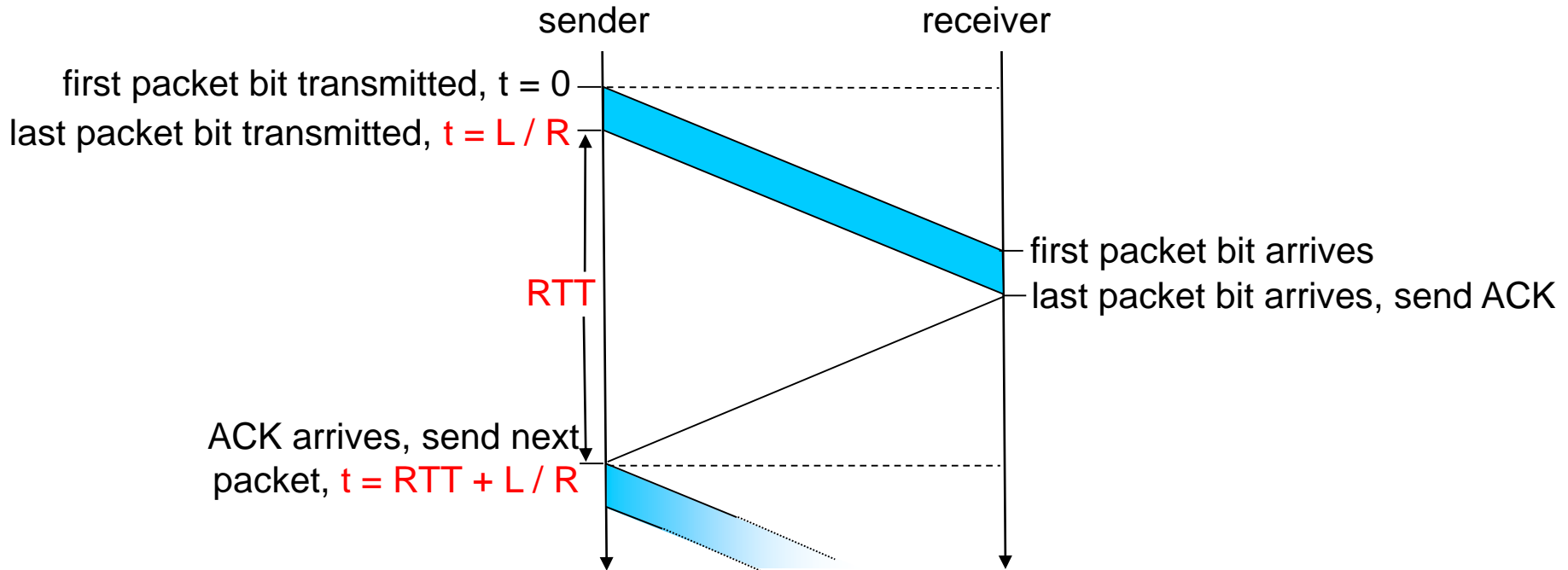
$$T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

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

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

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

Rdt3.0: Stop-and-wait operation

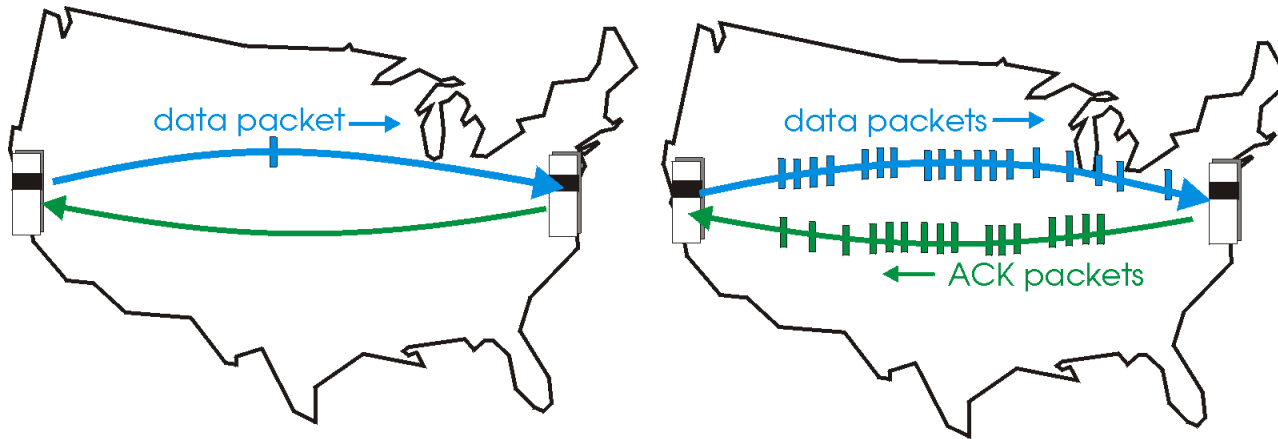


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

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

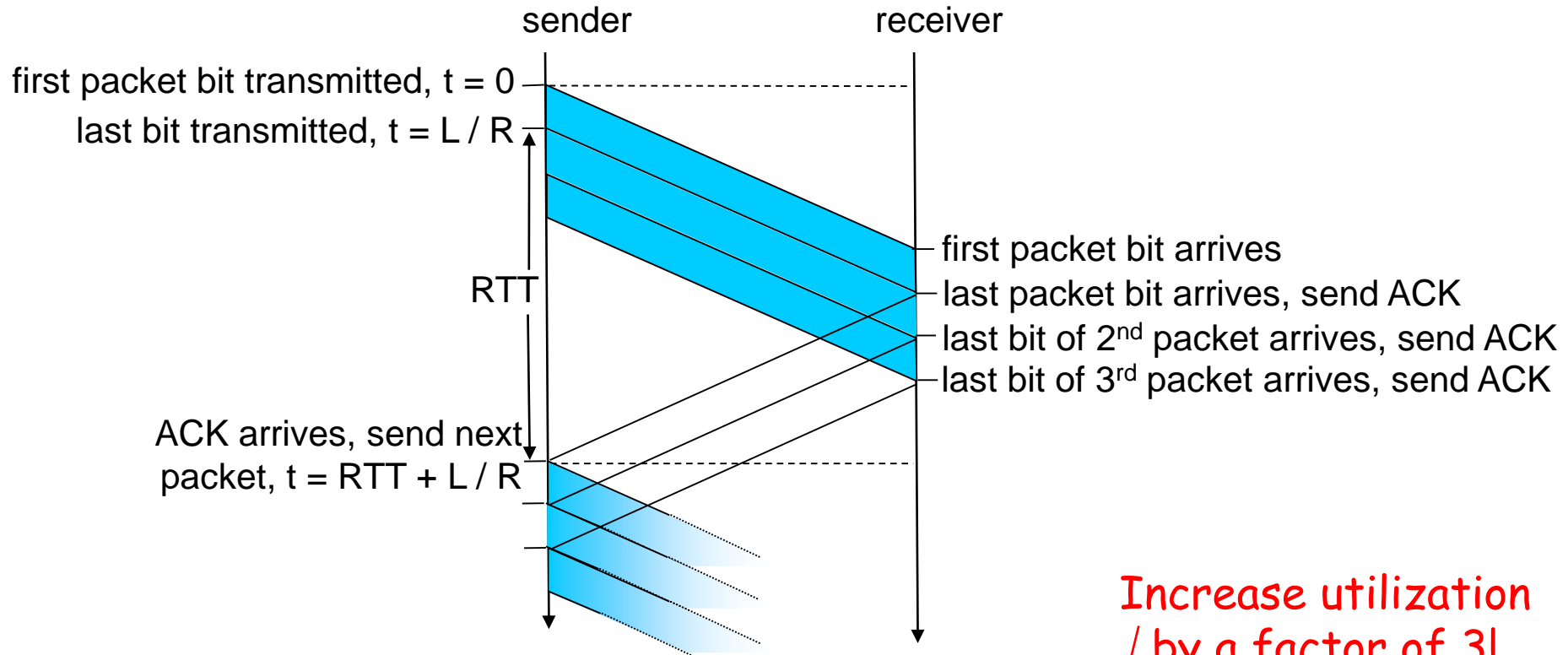


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: Increased utilization

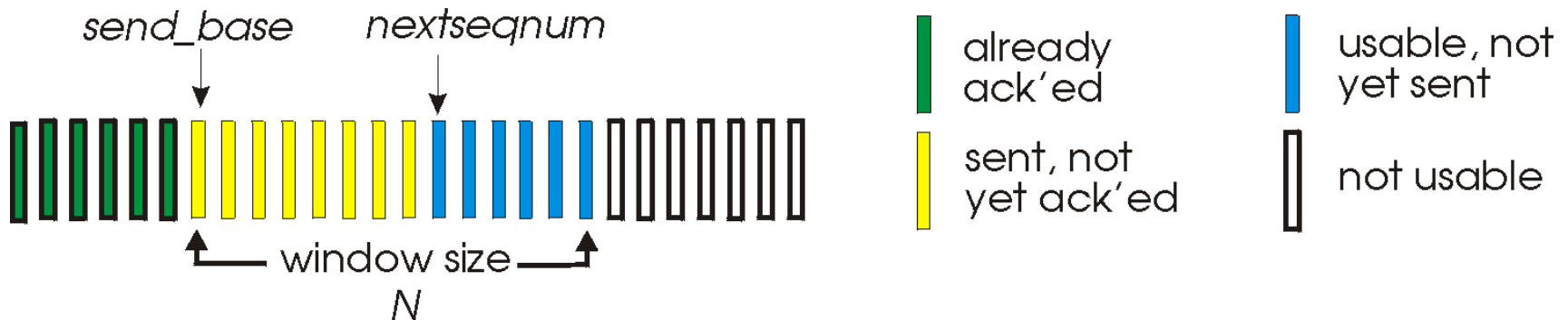


Increase utilization
by a factor of 3!

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

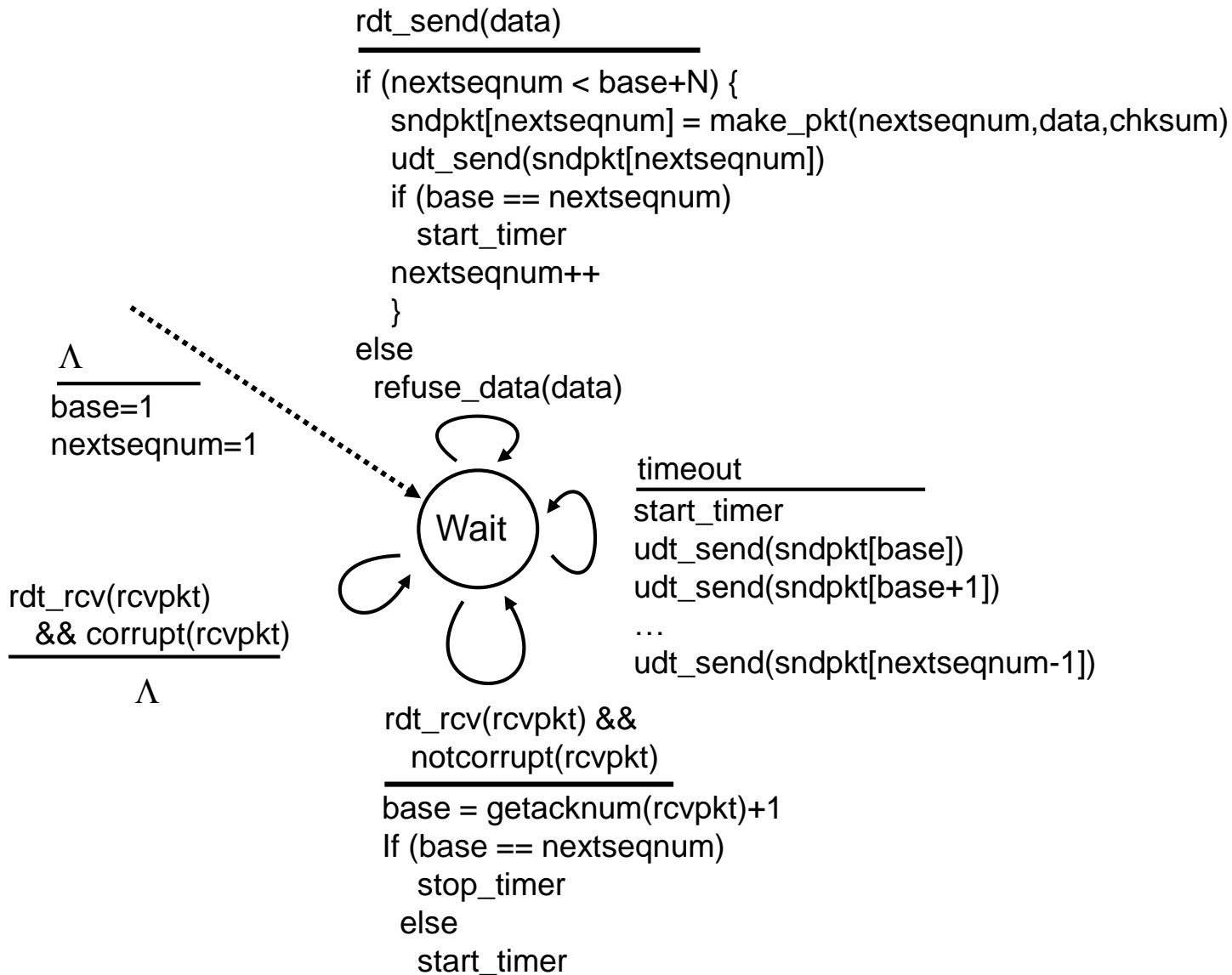
Sender:

- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed

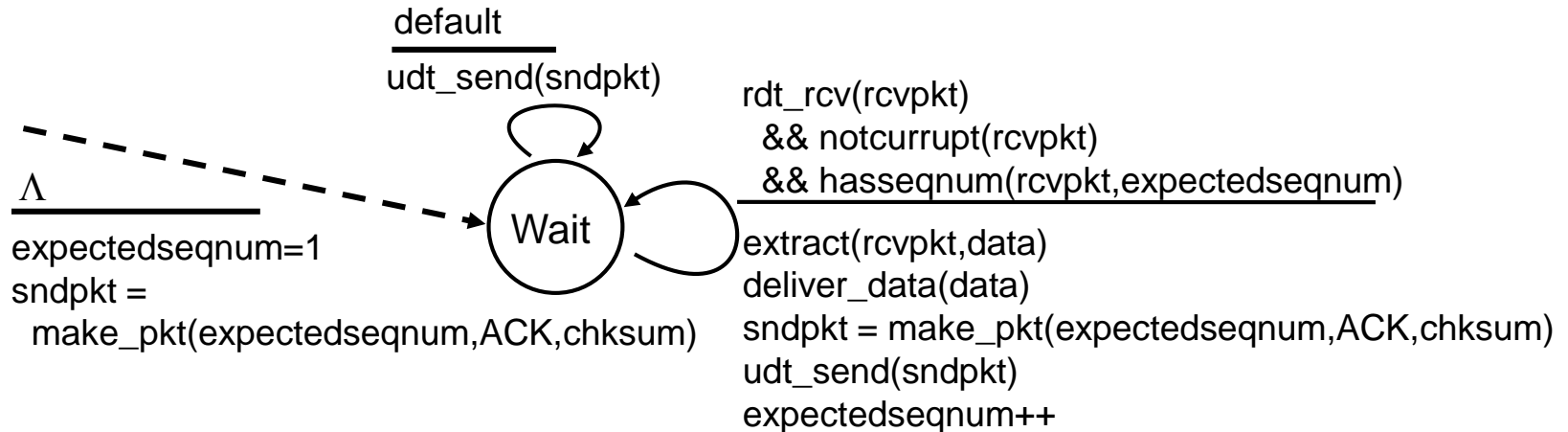


- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
 - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

GBN: Sender extended FSM



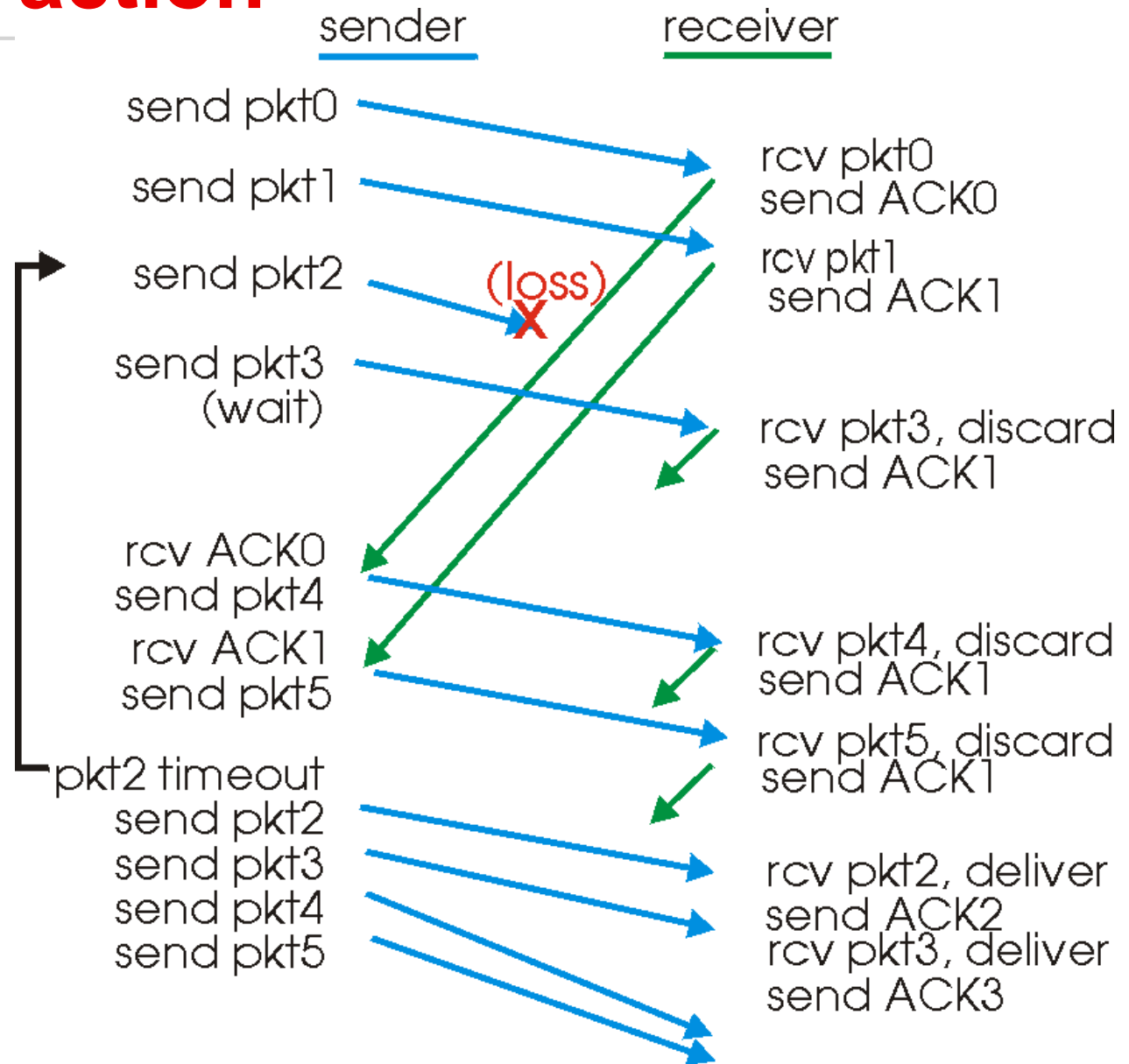
GBN: Receiver extended FSM

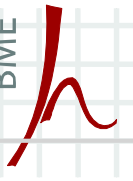


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**
- Out-of-order pkt:
 - discard (don't buffer) -> **no receiver buffering!**
 - re-ACK pkt with highest in-order seq #

GBN in action

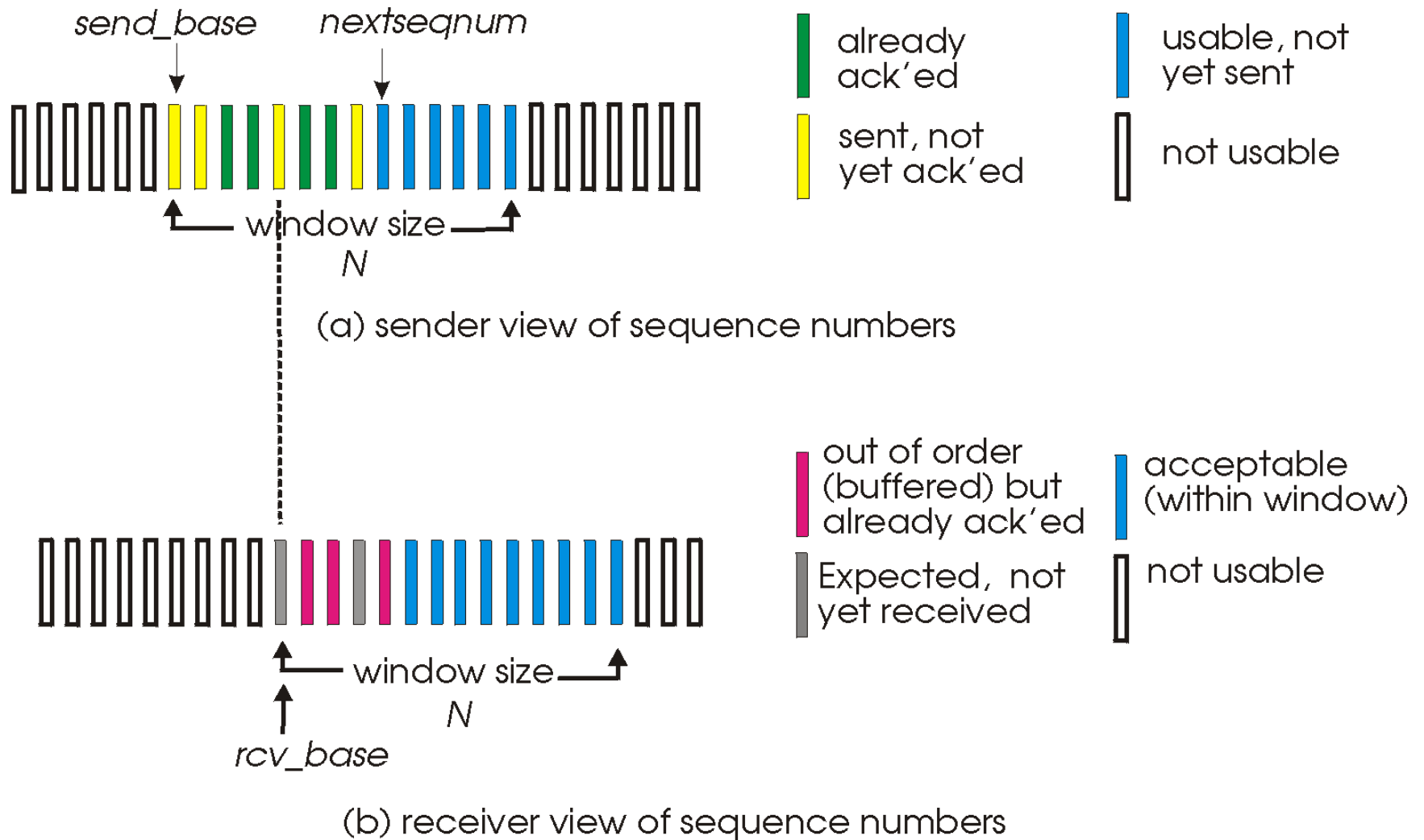




Selective repeat

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

Selective repeat: Sender, receiver windows



Selective repeat: Sender, receiver

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)
- Pkt is out-of-order: buffer it!
- Pkt is in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt!

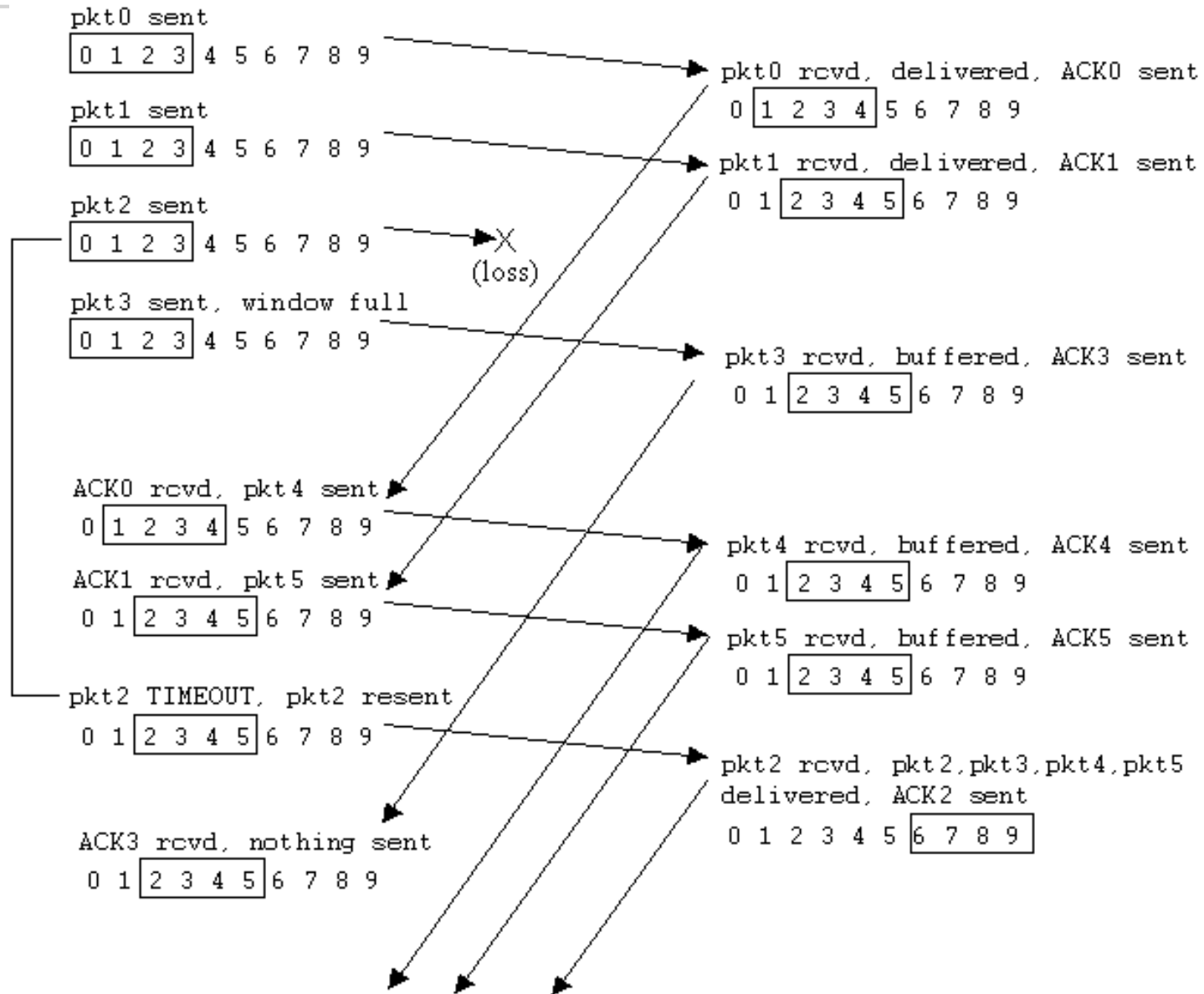
Pkt n in [rcvbase-N, rcvbase-1]

- ACK(n), even though this is a packet previously ACK-ed

Otherwise

- Ignore the packet

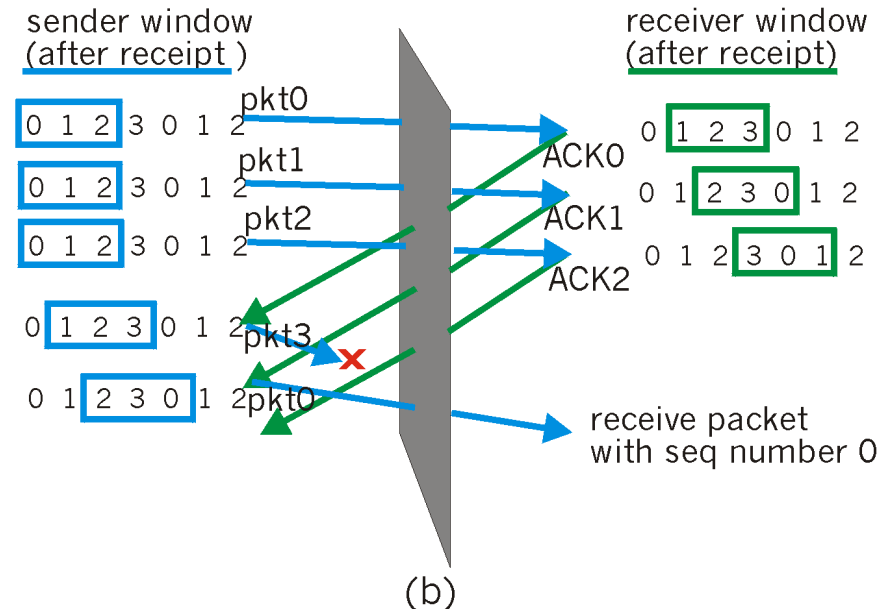
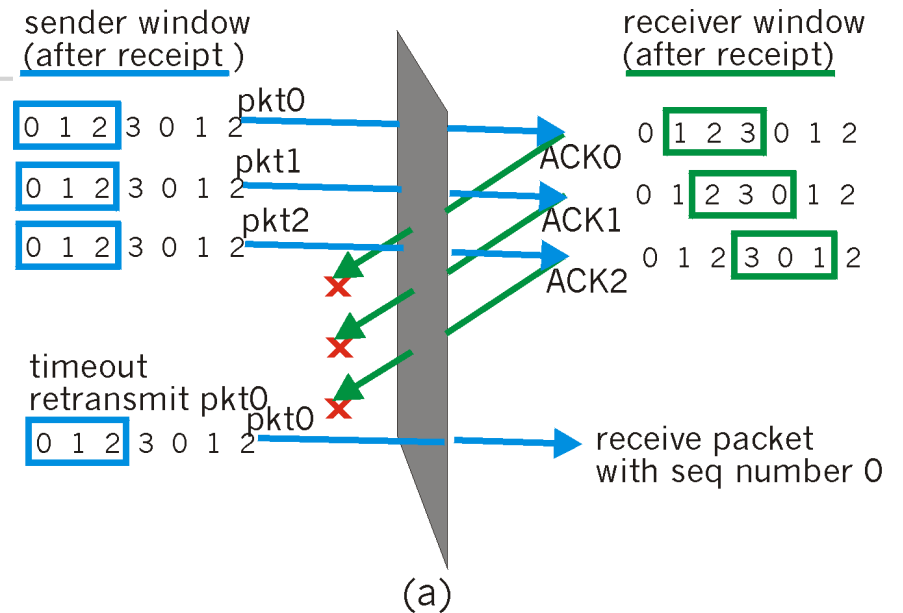
Selective repeat in action



Dilemma

Example:

- Seq #'s: 0, 1, 2, 3
- Window size=3
- Receiver sees no difference in the two scenarios!
- Incorrectly passes duplicate data as new in (a)
- No way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet -> the windows size must be less than or equal to half the size of the sequence number space for SR schemes!



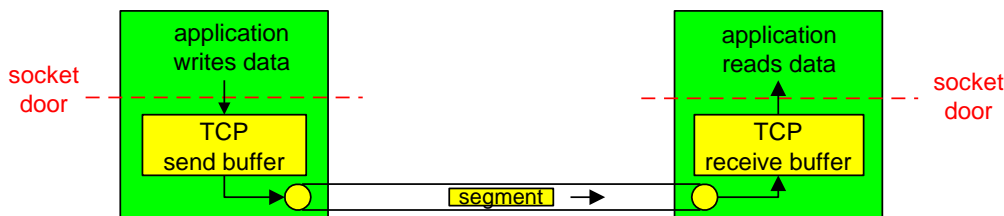
Chapter 3 outline

- 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

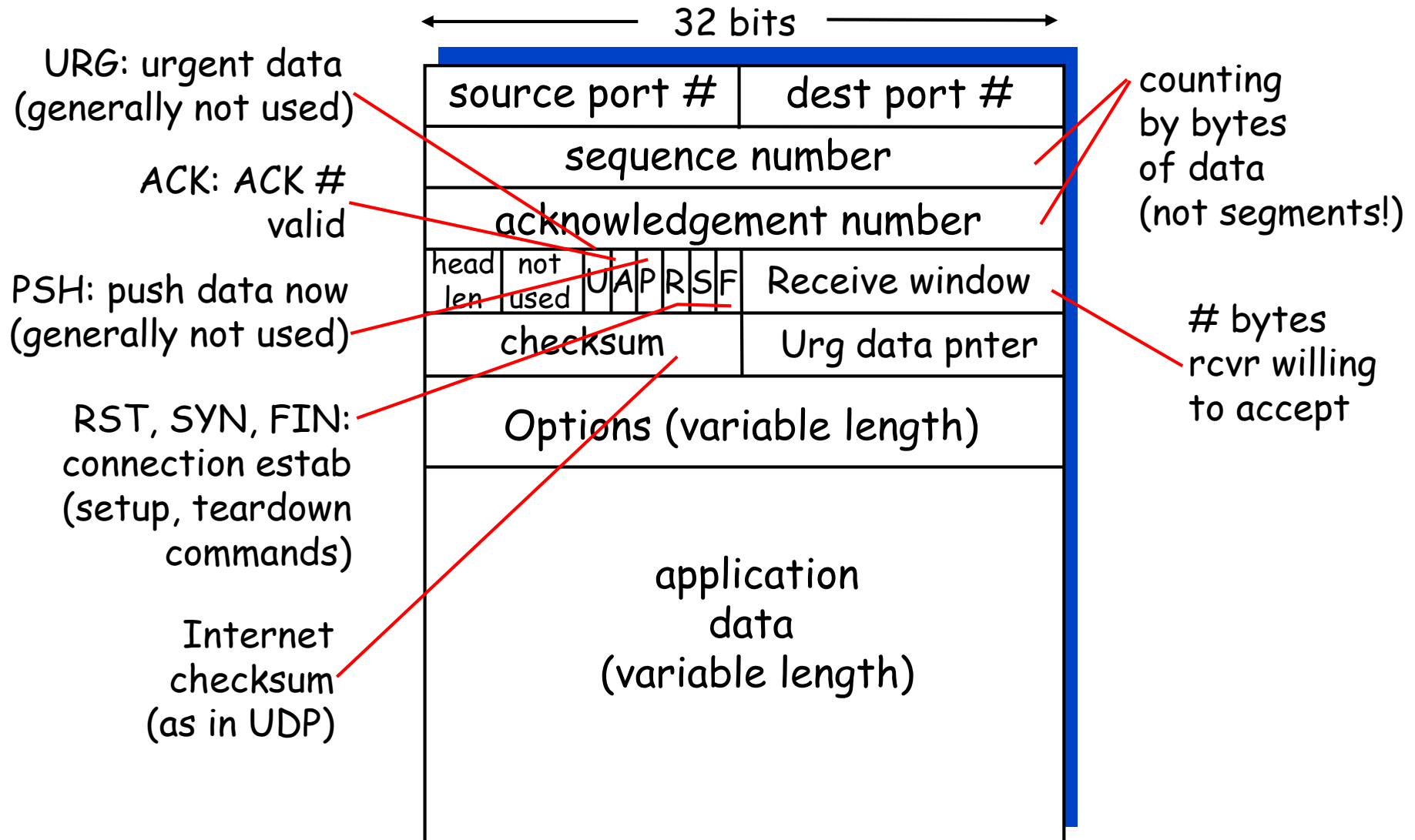
TCP: Overview

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 msgs) 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:

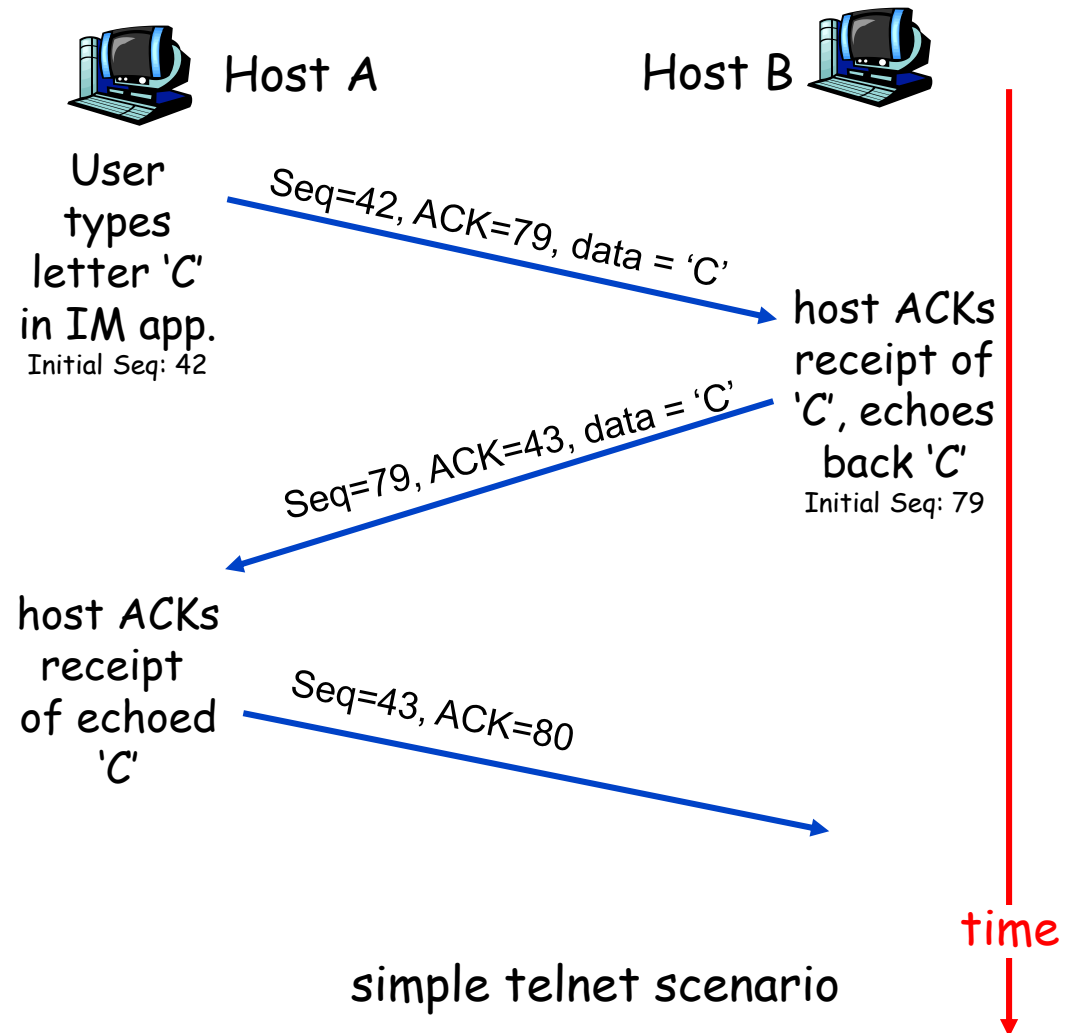
- byte stream
“number” of first byte in segment's data

ACKs:

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

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

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



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**

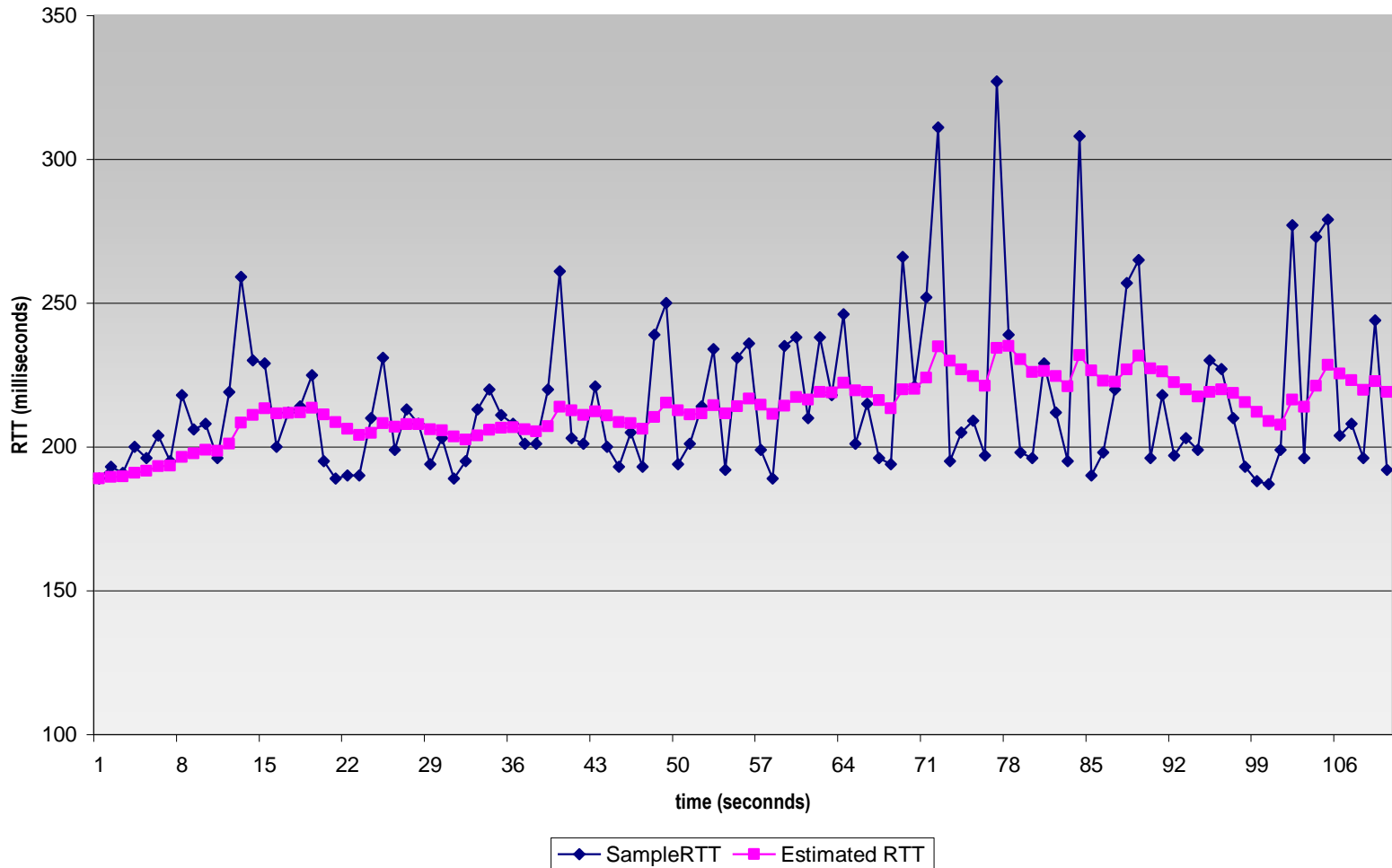
TCP Round Trip Time and Timeout

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

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

Example RTT estimation

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr





TCP Round Trip Time and Timeout

Setting the timeout

- **EstimatedRTT** plus “safety margin”
 - large variation in **EstimatedRTT** → larger safety margin
- First estimate of how much **SampleRTT** deviates from **EstimatedRTT**:

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

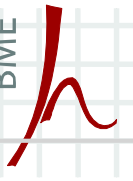
(typically, $\beta = 0.25$)

Then set timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

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:

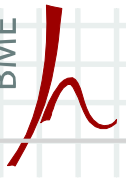
- Create segment with seq #
- Seq # is byte-stream number of first data byte in segment
- Start timer if not already running (think of timer as for oldest unACKed segment)
- Expiration interval:
`TimeoutInterval`

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)

Comment:

- `SendBase`: seq. of the oldest unACK-ed byte
- `SendBase-1`: last cumulatively ack'ed byte

Example:

- `SendBase-1 = 71`;
`y=73`, so the rcvr wants 73+ ;
`y > SendBase`, so that new data is acked

```
NextSeqNum = InitialSeqNum
```

```
SendBase = InitialSeqNum
```

```
loop (forever) {
    switch(event)
```

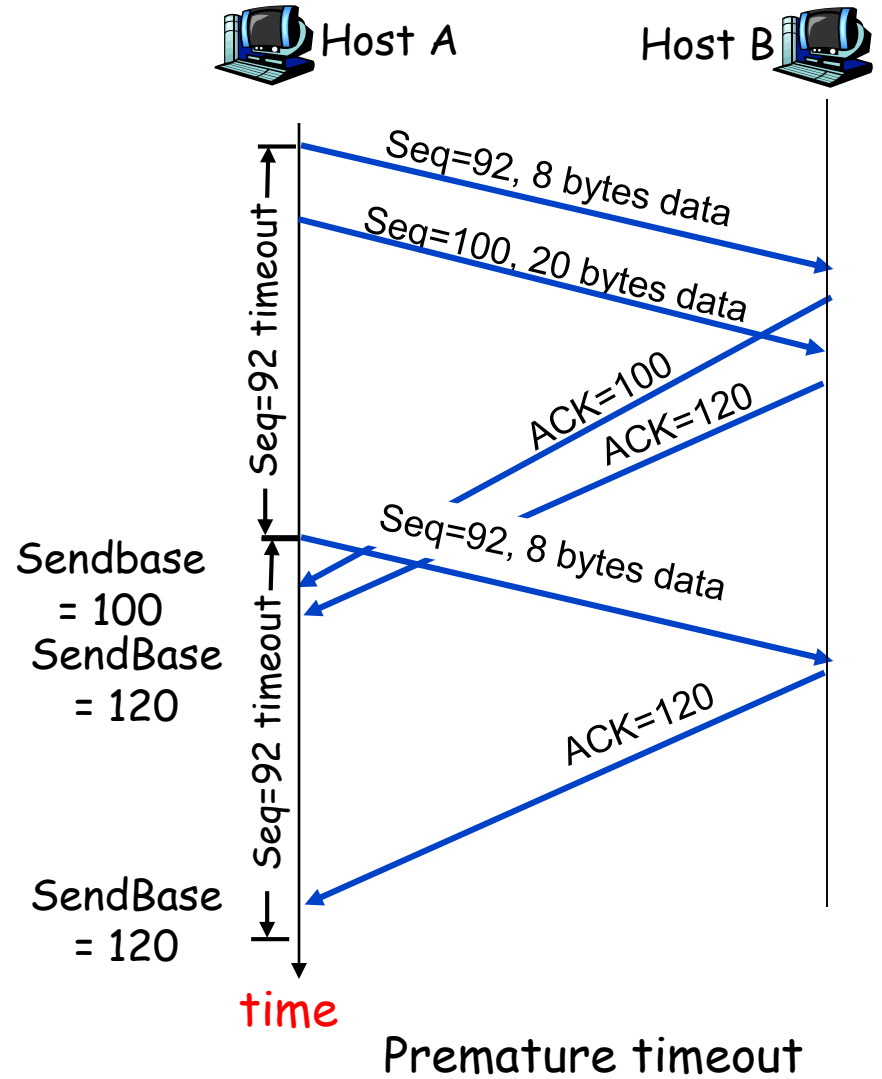
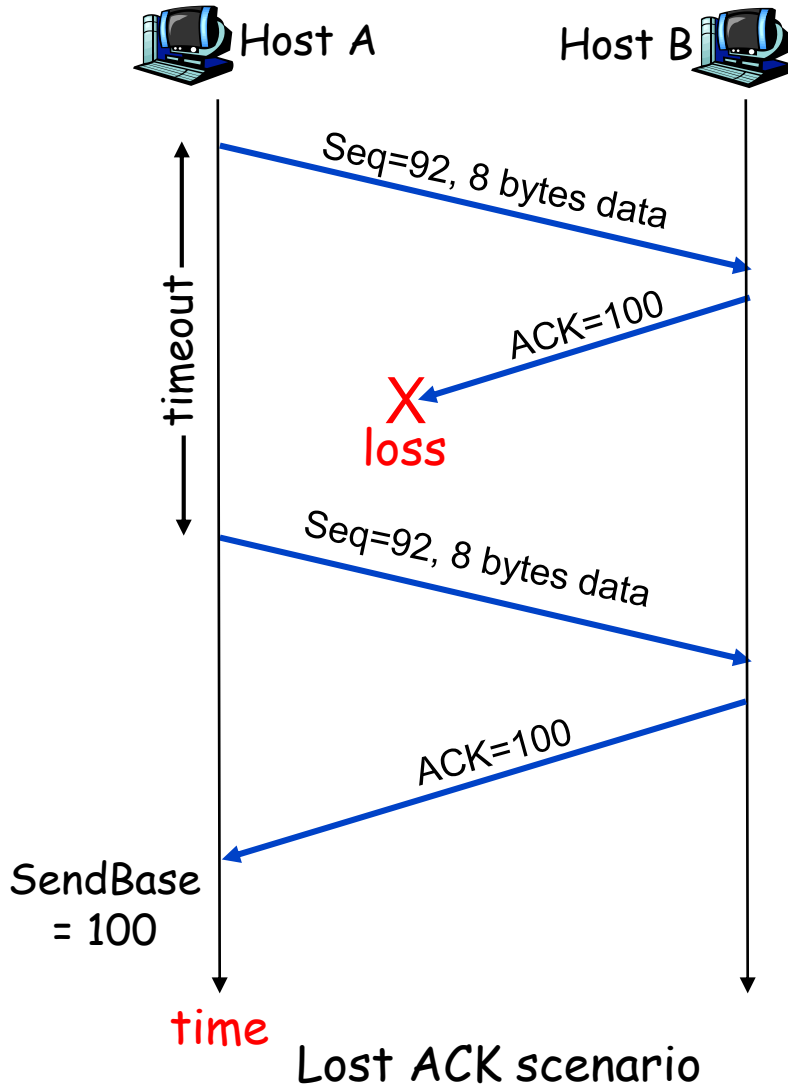
```
    event: data received from application above
           create TCP segment with sequence number NextSeqNum
           if (timer currently not running)
               start timer
           pass segment to IP
           NextSeqNum = NextSeqNum + length(data)
```

```
    event: timer timeout
           retransmit not-yet-acknowledged segment with
           smallest sequence number
           start timer /*for the retransmitted data*/
```

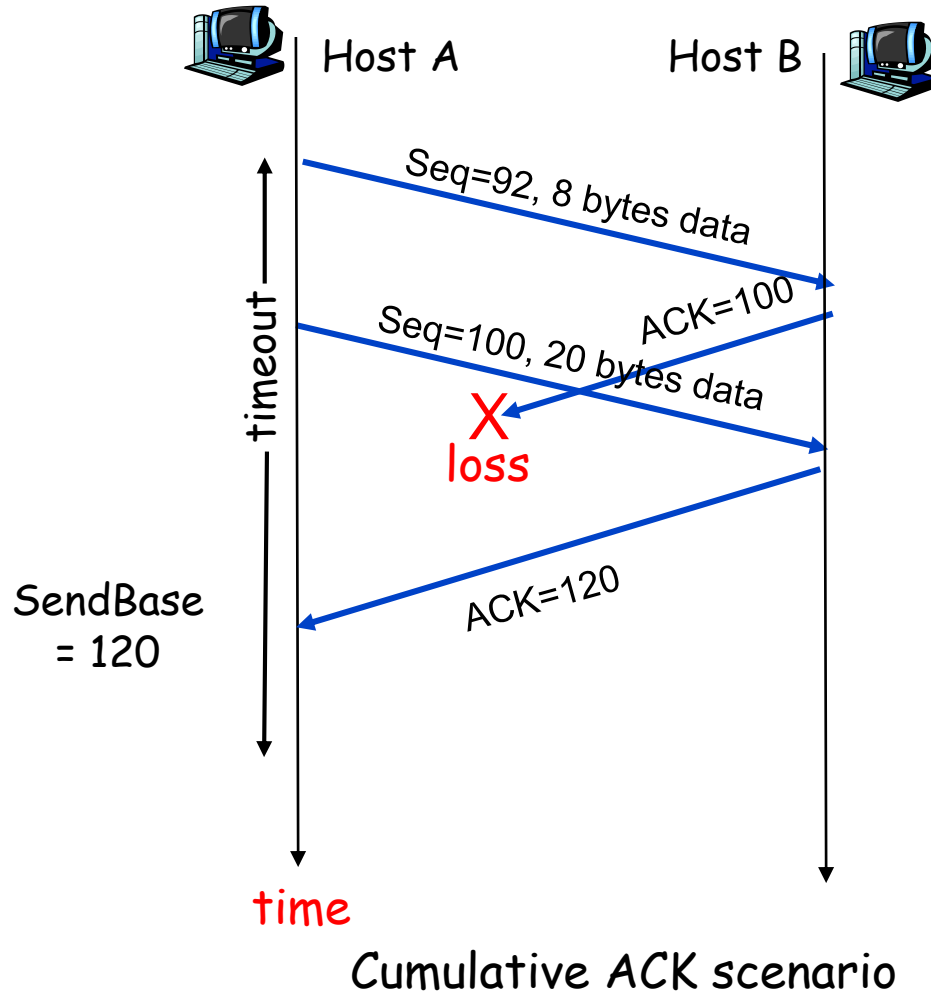
```
    event: ACK received, with ACK field value of y
           if (y > SendBase) { /*ACK acknowledges one or more
                               previously unACK-ed segments */
               SendBase = y
               if (there are currently not-yet-acknowledged segments)
                   start timer /*for them*/
           }
```

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

TCP: Retransmission scenarios



TCP retransmission scenarios (more)



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

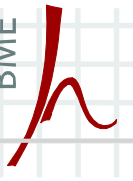
Immediately send single cumulative ACK, ACKing both in-order segments

Arrival of out-of-order segment higher-than-expected 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

- Timeout 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 ACKs for the same data, it supposes that segment after ACKed data was lost
 - Fast retransmit: resend segment before timer expires

Fast retransmit algorithm

```
event: ACK received, with ACK field value of y
    if (y > SendBase) { /*ACK acknowledges one or more
                        previously unACK-ed segments*/
        SendBase = y
        if (there are currently not-yet-acknowledged segments)
            start timer /*for them*/
    }
    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

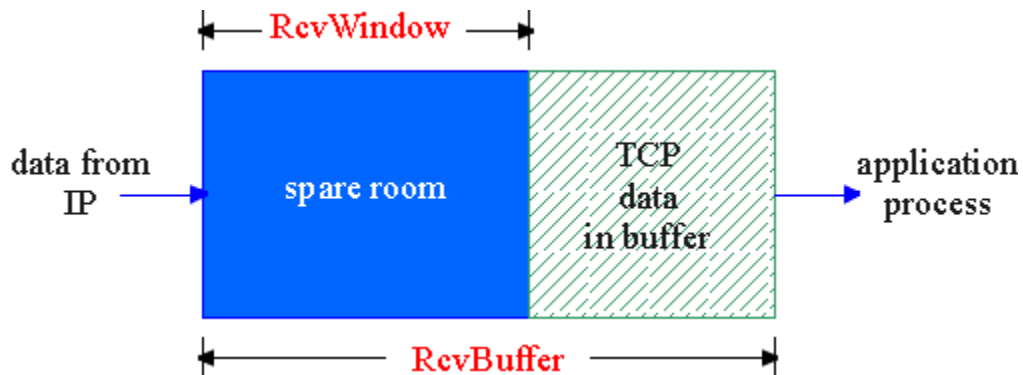
fast retransmit

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

- Receive side of TCP connection has a receive buffer:

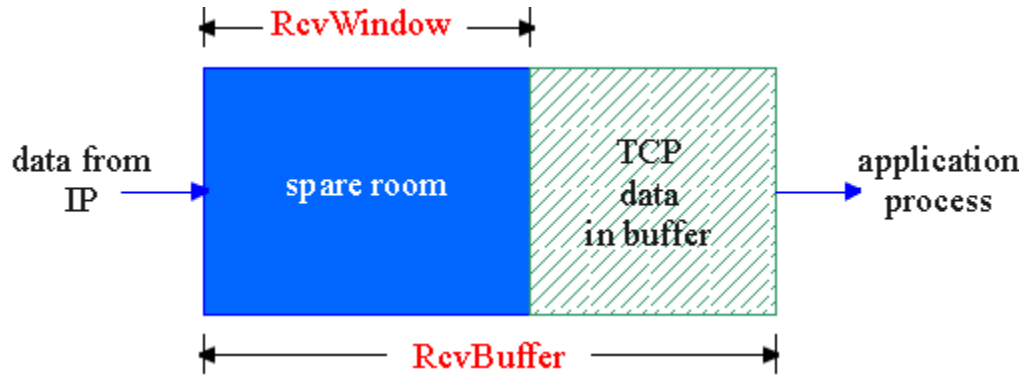


- App process may be slow at reading from buffer

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?



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

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

- Spare room in buffer
 - = **RcvWindow**
 - = **RcvBuffer** - [**LastByteRcvd** - **LastByteRead**]

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
- *Server:* contacted by client

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'd)

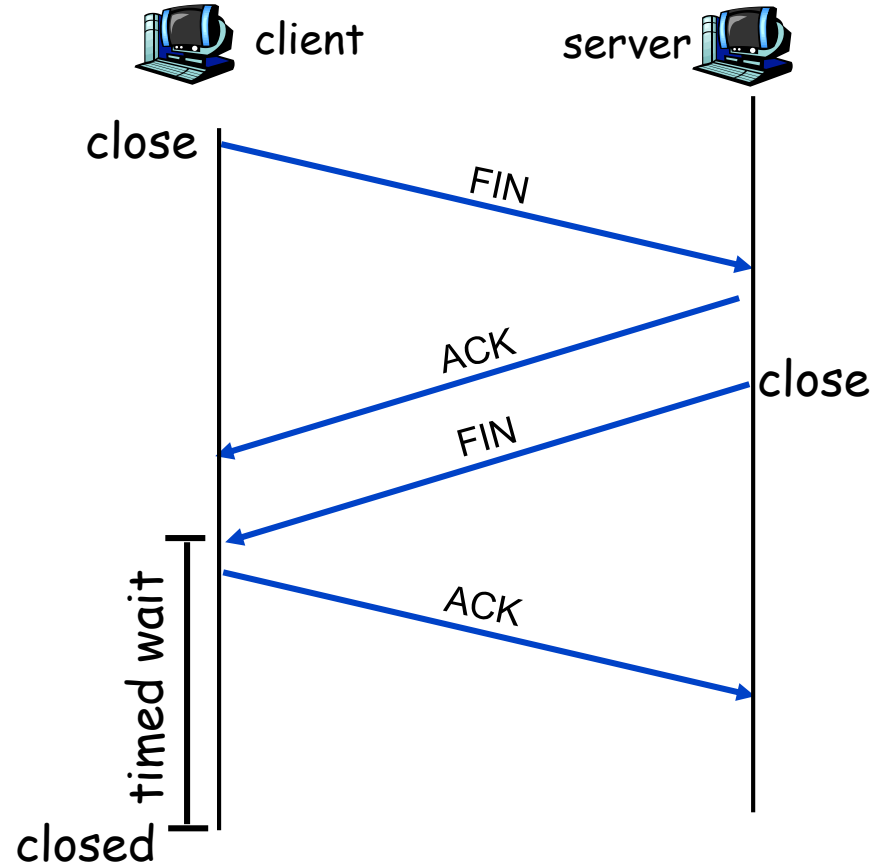
Closing a connection:

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



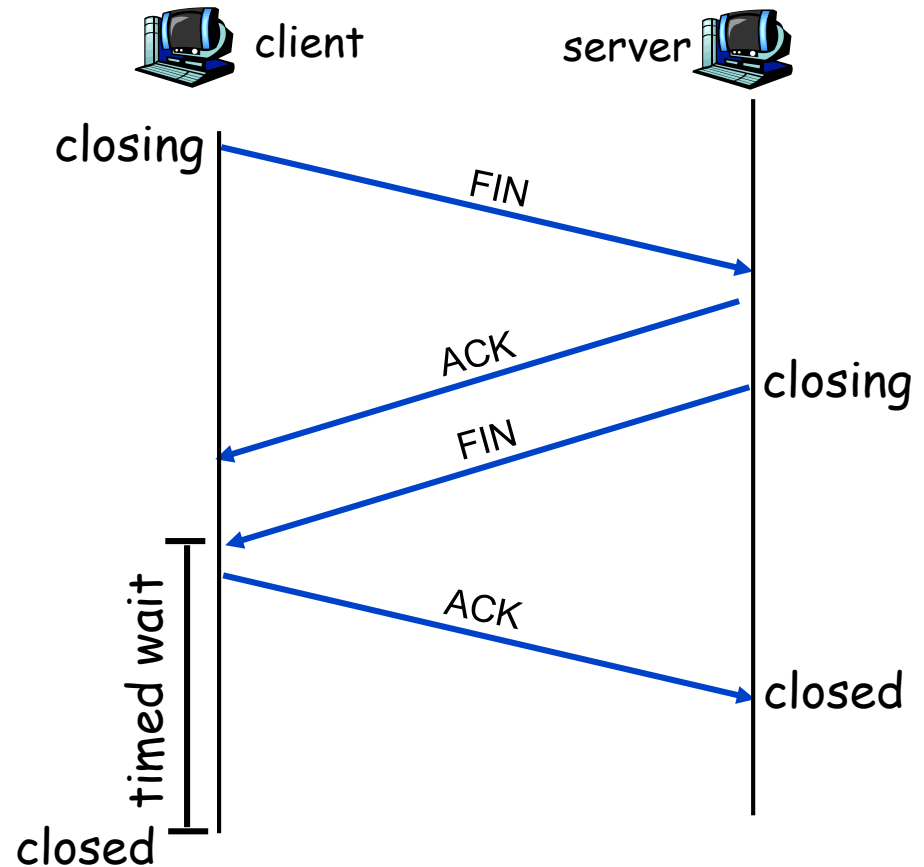
TCP connection management (cont'd)

Step 3: Client receives FIN, replies with ACK

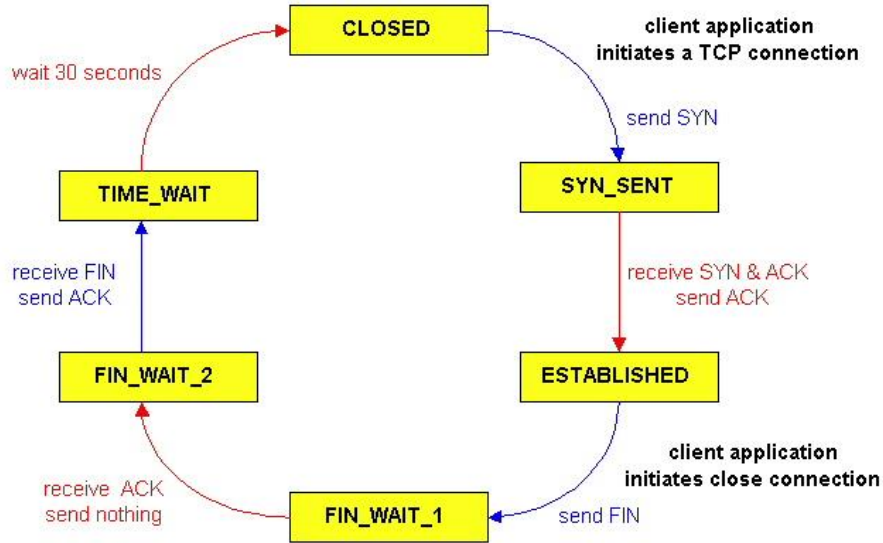
- Enters “timed wait” - will respond with ACK to received FINs

Step 4: Server receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

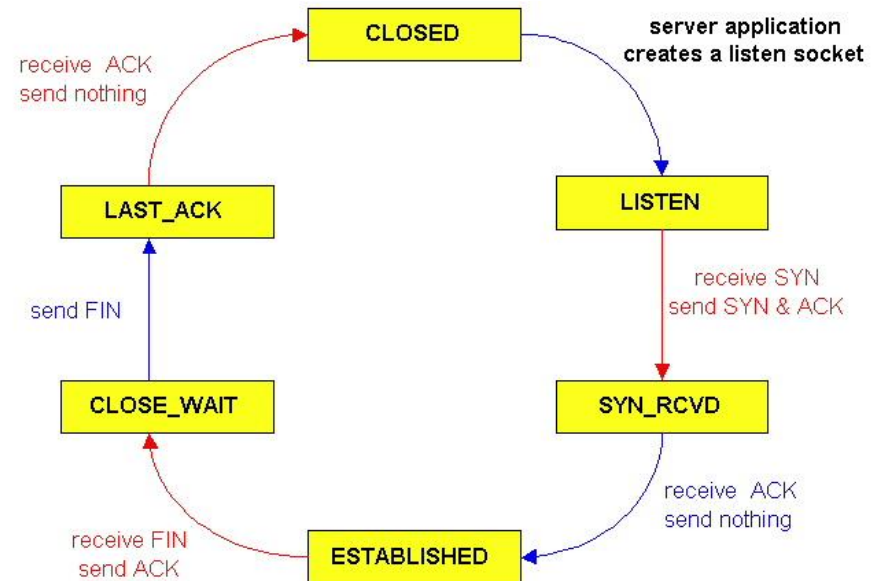


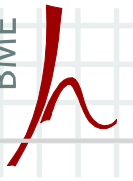
TCP connection management (cont'd)



TCP client lifecycle

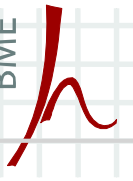
TCP server lifecycle





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



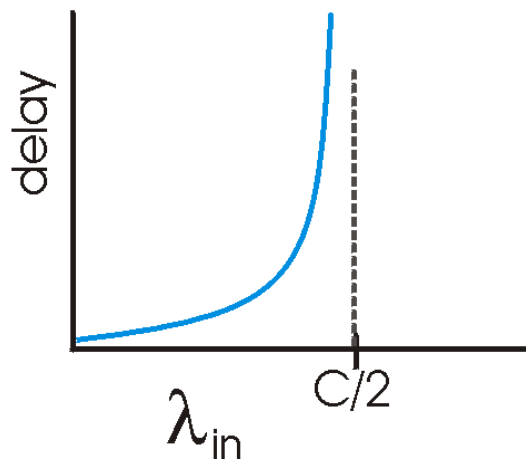
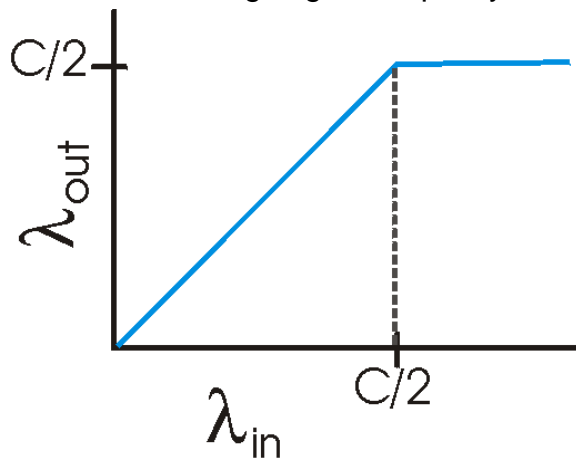
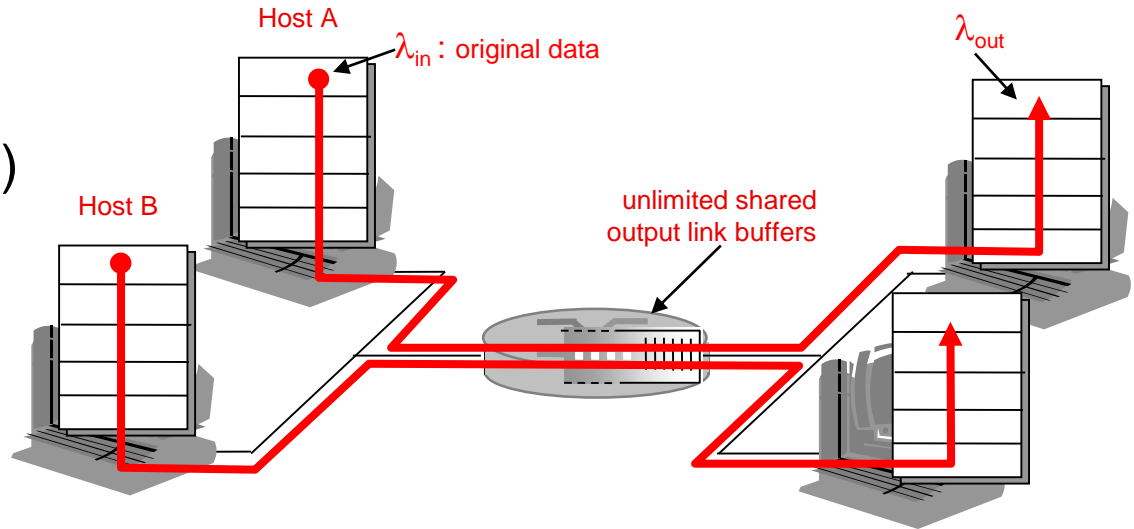
Principles of congestion control

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 (queueing in router buffers)
- A top-10 problem!

Causes/costs of congestion: Scenario 1

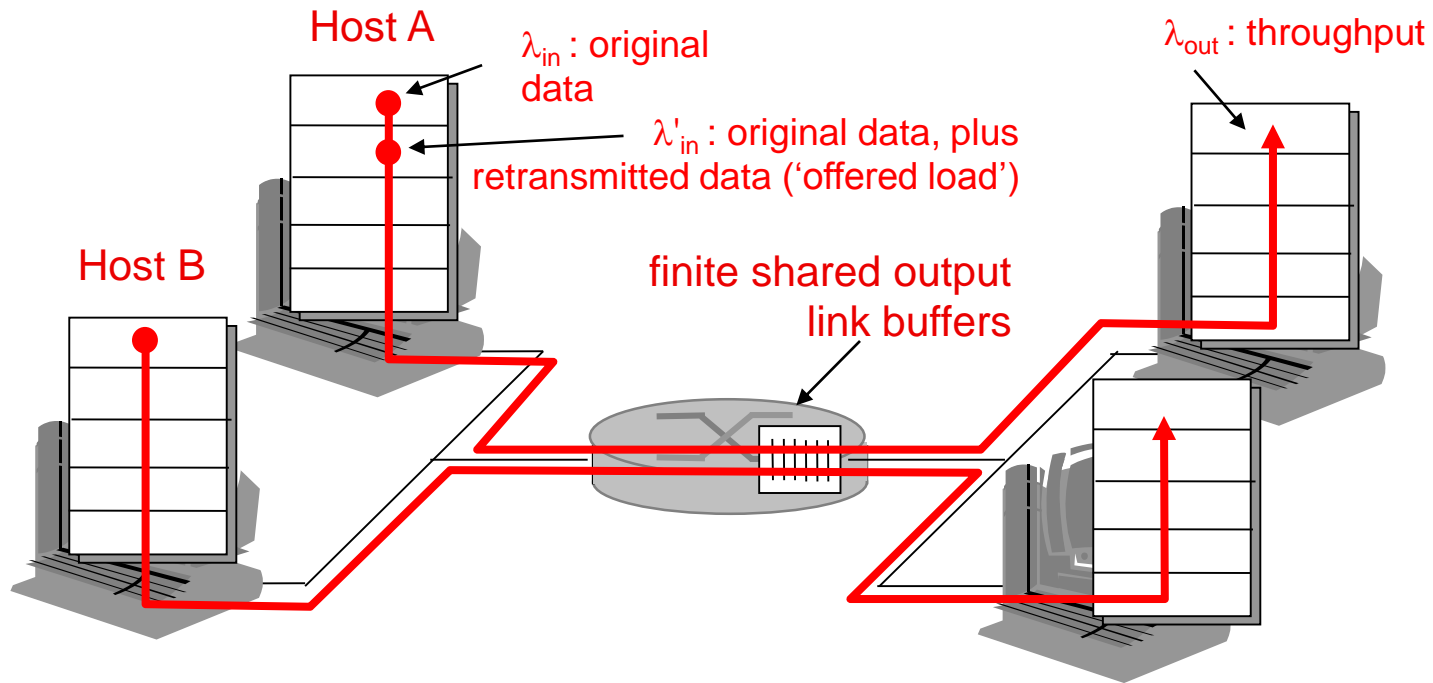
- Two senders (rate λ_{in})
- Two receivers (rate λ_{out})
- One router, infinite buffers
- No retransmission
- Legend:
 - Rate in/out: λ bytes/s
 - Outgoing link capacity of router: C



- Large delays when congested
- Maximum achievable throughput

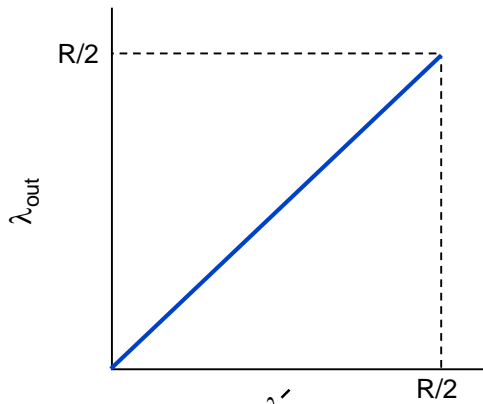
Causes/costs of congestion: Scenario 2

- One router, *finite* buffers
- Sender retransmission of lost packet

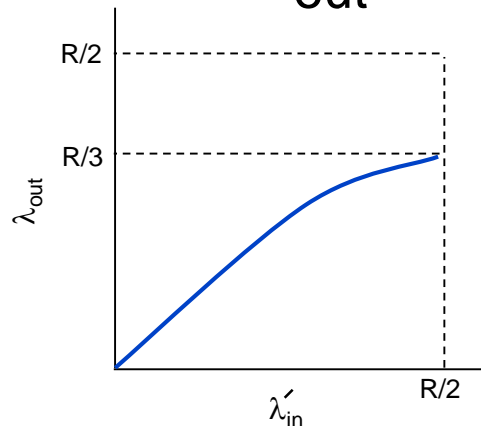


Causes/costs of congestion: Scenario 2

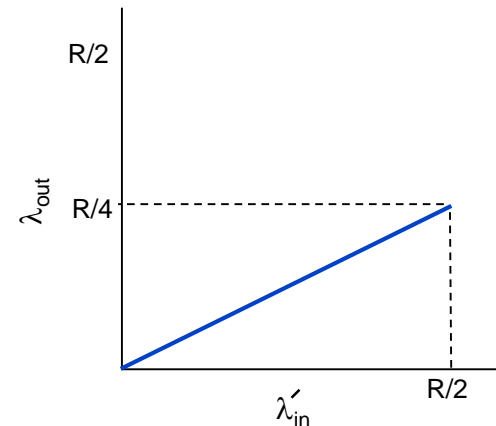
- a.) Always: $\lambda_{in} = \lambda_{out}$ (goodput)
- b.) “Perfect” retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- c.) Retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



- Host A „magically” determines whether or not the buffer is free
- No loss



- Sender retransmits only when a packet is known for certain to be lost
- Still ideal!
- The rate at which receiver drains packets by the application is R/3



- Here both original packet and its retransmission can reach the receiver
- Each packet is assumed to be forwarded on average twice by the router

“Costs” of congestion:

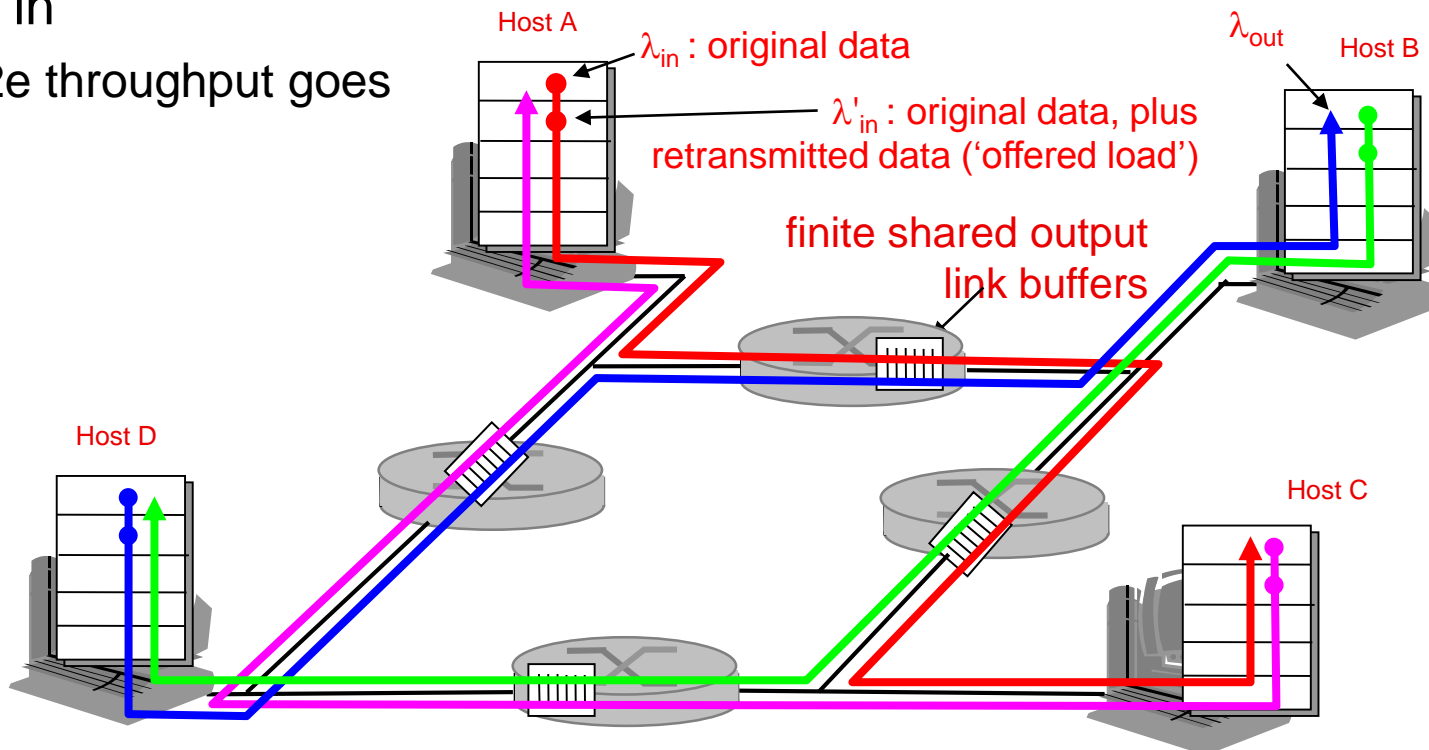
- Sender must compensate for lost packets due to buffer overflow: more work (retrans) for given “goodput”
- Unneeded retransmissions: link carries multiple copies of pkt

Causes/costs of congestion: Scenario 3

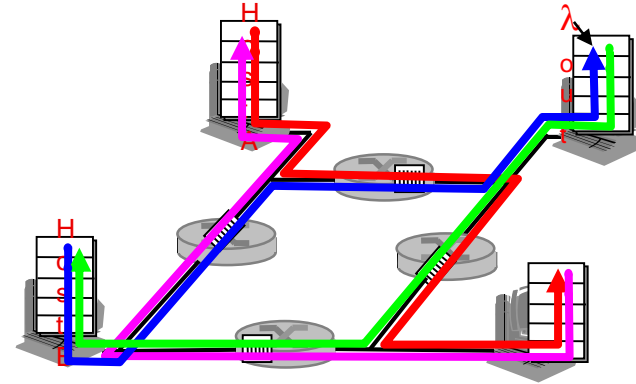
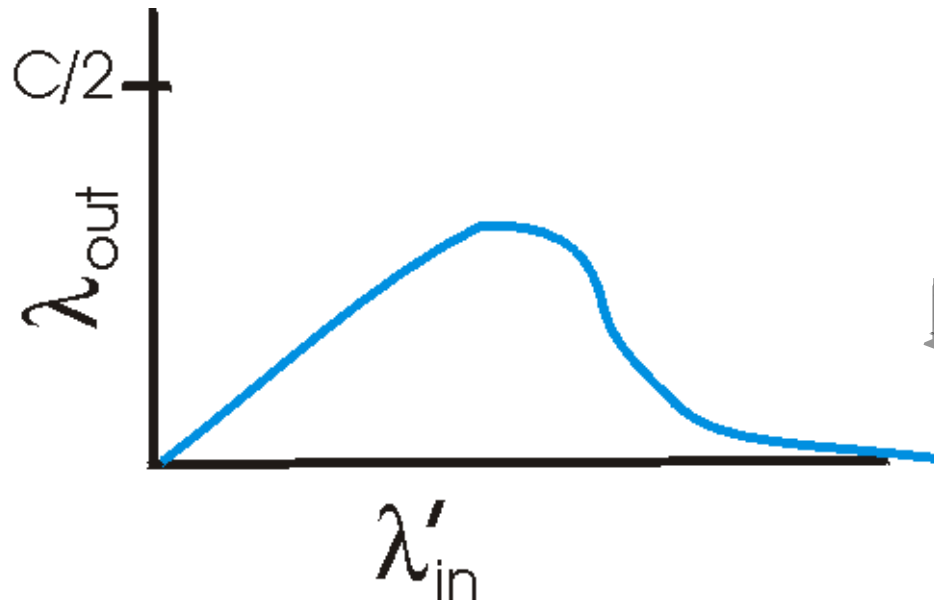
- Parameters:
 - Four senders
 - Overlapping two-hop multihop paths
 - Timeout/retransmit
- For small values of λ_{in} an increase in λ_{in} results in increase of the throughput

Q: What happens as λ_{in} and λ'_{in} increase?

A: A-C e2e throughput goes zero!

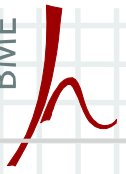


Causes/costs of congestion: Scenario 3



Another “cost” of congestion:

- When packet dropped (e.g. in a second hop router), any “upstream” transmission capacity (e.g. in a first hop router) used for that packet was wasted!



Approaches towards congestion control

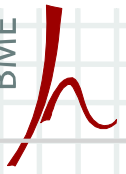
Two broad approaches towards congestion control:

End-end congestion control:

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

Network-assisted congestion control:

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



Case study: ATM ABR congestion control

- Asynchronous Transfer Mode (ATM): a standard switching technique designed to unify telecommunication and computer networks.
- Introduces both circuit switched and packet switched networking paradigms
 - It uses asynchronous time-division multiplexing, and it encodes data into small, fixed-sized cells.
 - Creates and maintains Virtual Circuits (VCs) to track the behavior of senders
- Good but expensive technology

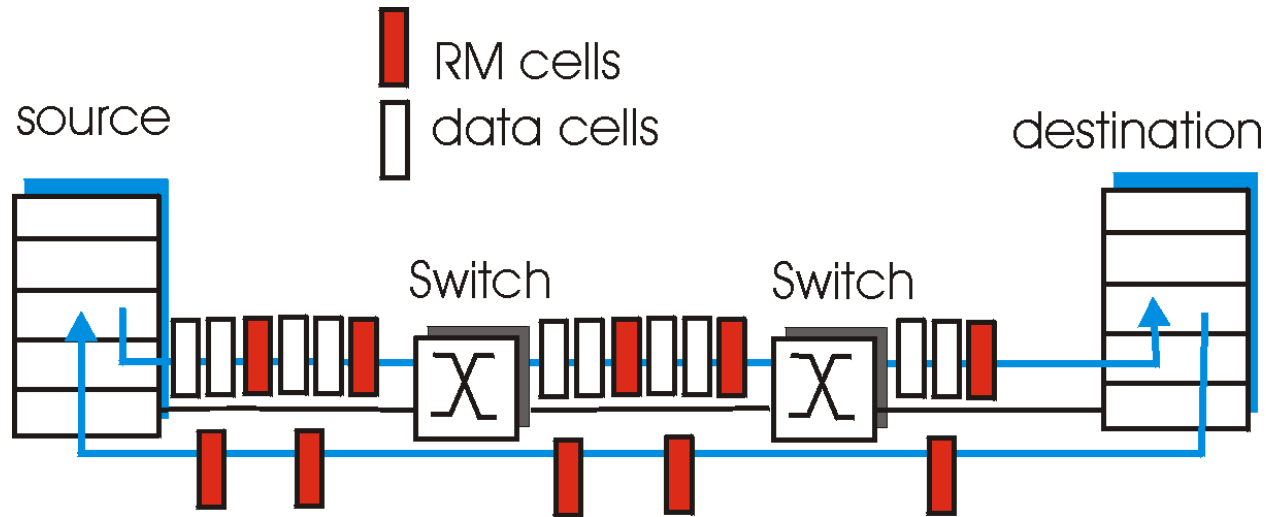
ATM ABR: available bit rate

- “Elastic service”
- If sender’s path “underloaded”:
 - sender should use available bandwidth
- If sender’s path congested:
 - sender throttled to minimum guaranteed rate

RM (resource management) cells:

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

Case study: ATM ABR congestion control



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



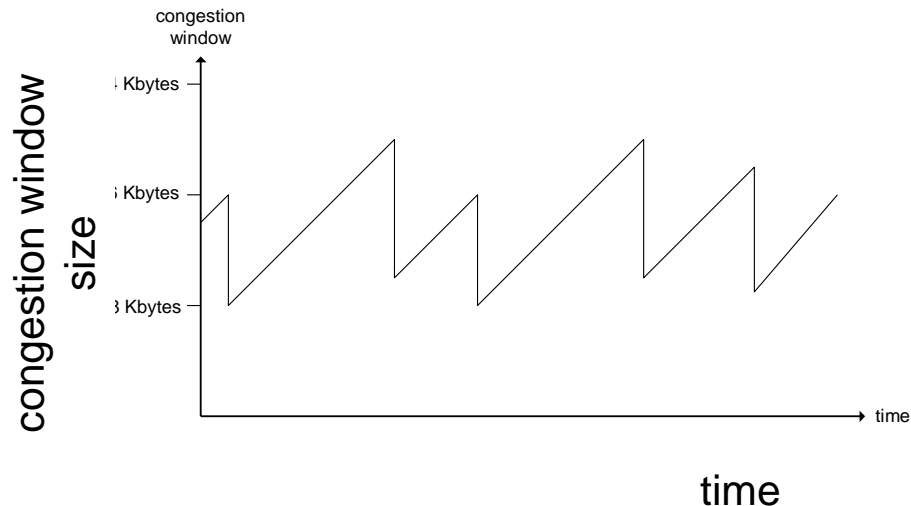
Chapter 3 outline

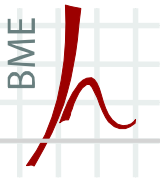
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 Principles of congestion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

TCP congestion control

- *Approach*: increase transmission rate (window size), probing for usable bandwidth, until loss occurs
 - Sender keeps track of an additional variable: congestion window (a constraint on the sending rate) -> the amount of unACKed data at a sender may not exceed the minimum of congestion window and receiver window
 - *Additive increase*: increase **CongWin** by 1 MSS (Max. Segment Size) every RTT until loss detected
 - *Multiplicative decrease*: cut **CongWin** in half after loss
 - The name of this scheme is AIMD: Additive increase/multiplicative decrease

Saw tooth
behavior: probing
for bandwidth





TCP congestion control: Details

- Assume that receiver's TCP buffer is so large that rwnd constraint can be ignored

- Sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$$

- Roughly,

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

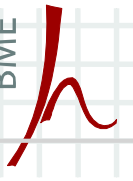
- CongWin** is dynamic, function of perceived network congestion

How does sender perceive congestion?

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

Four mechanisms:

- AIMD
- Slow start
- Conservative after timeout events
- Fast retransmission and recovery

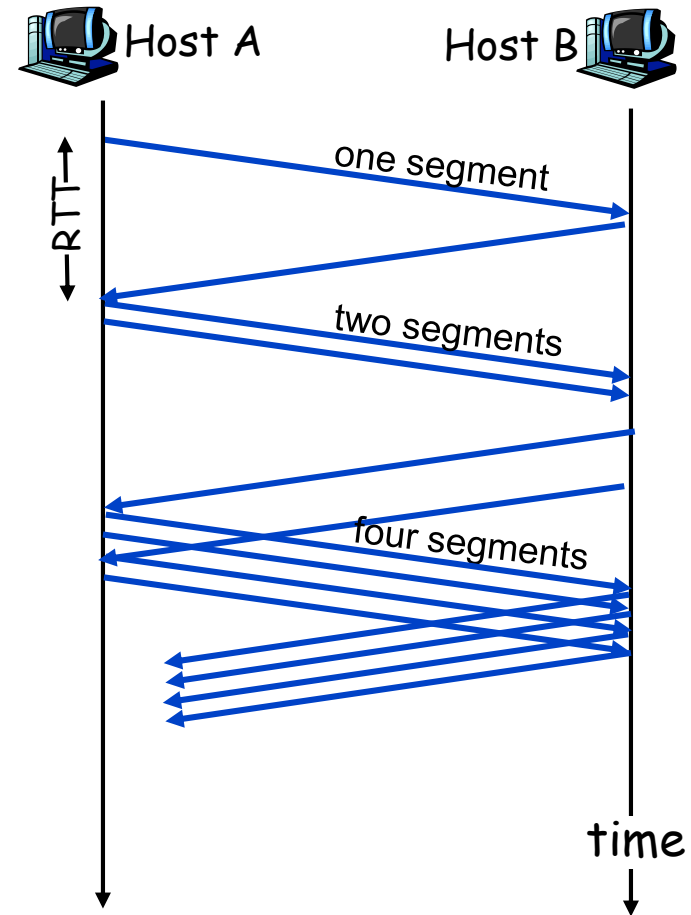


TCP slow start

- When connection begins, **CongWin** = 1 MSS
 - Example: MSS = 500 bytes & RTT = 200 msec
 - Initial rate = 20 kbps
- Available bandwidth may be much greater than 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 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



Refinement

Q: When should the exponential increase switch to linear?

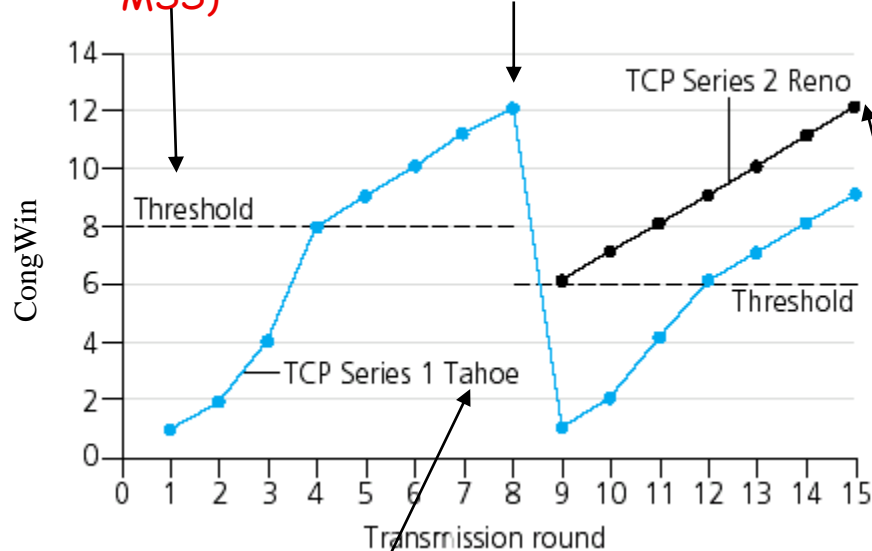
A: When **CongWin** gets to 1/2 of its value before timeout. This is the so called congestion avoidance („congestion is just around the corner”)

Implementation:

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

Threshold: half the value of cwnd when congestion was last detected (here the initial value is 8 MSS)

cwnd is 12 MSS when the loss occurs in transmission round 8



TCP Tahoe is an early version of TCP: unconditionally cuts its cwnd to 1 MSS and enters slow-start phase in case of timeout indicated or triple dupACK indicated loss event

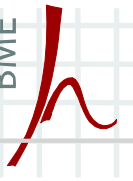
TCP Reno is a newer version of TCP: incorporates fast recovery -> sets cwnd to the threshold value (6 MSS) and grows linearly

Fast retransmission and recovery

- 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 until a threshold,
 - then grows linearly

Philosophy:

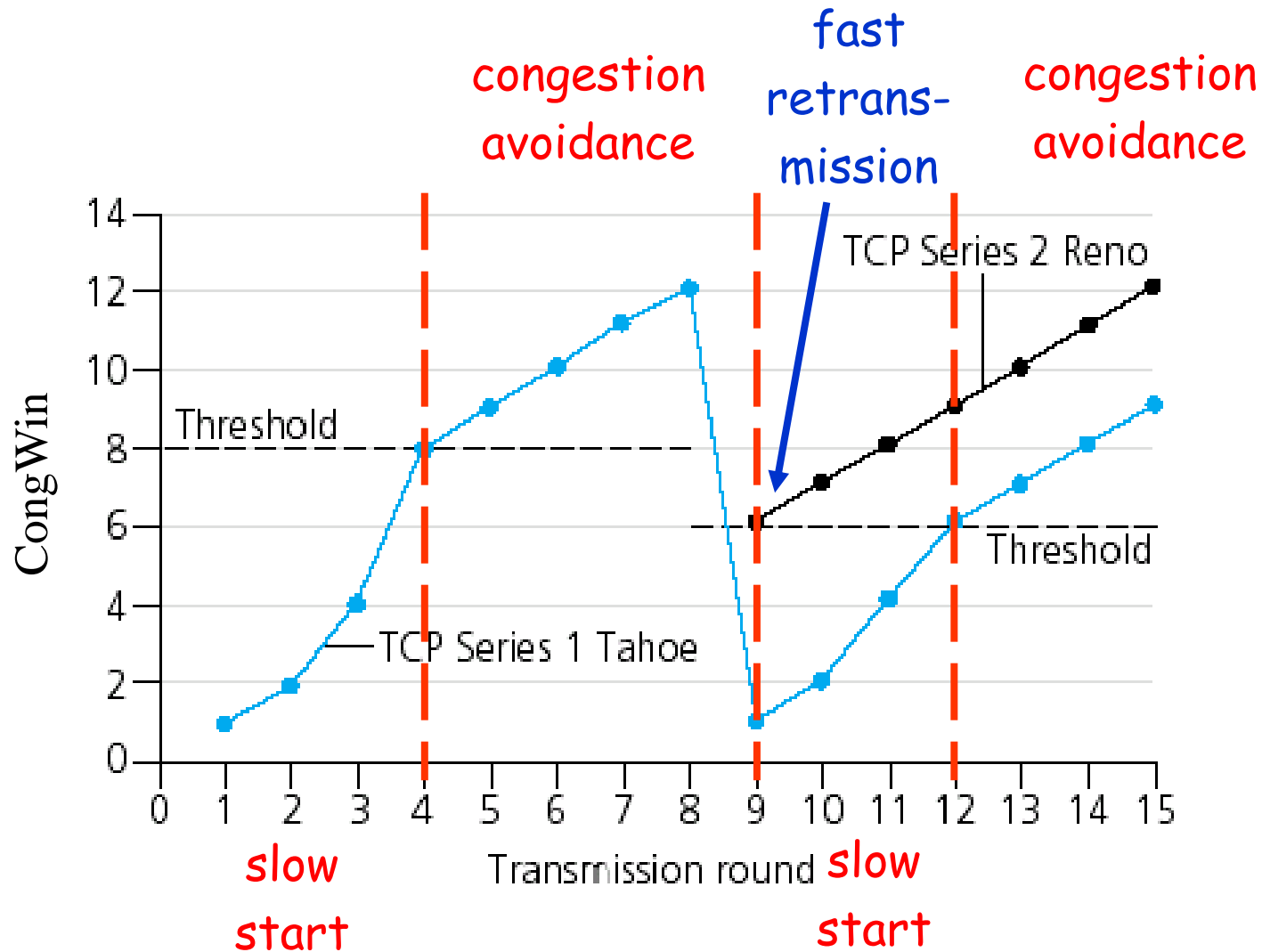
- ❑ 3 dup ACKs indicates network capable of delivering some segments
- ❑ timeout indicates a "more alarming" congestion scenario



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.

Summary: TCP congestion control

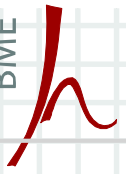


TCP sender congestion control

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

- Given the saw-toothed behavior of TCP the question arises: what is the average throughput 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 $W/2RTT$.
- Assuming that RTT and W are constant during the connection, the TCP transmission rate ranges from $W/2RTT$ to W/RTT
- Therefore a simplified macroscopic model for the average TCP throughput: $0.75 W/RTT$



TCP futures: TCP over “long, fat pipes”

- Example: a TCP connection with 1500 byte segments, 100ms RTT, and will use a 10 Gbps link (i.e. we want a throughput of 10 Gbps)
- Based on the previous formula, it requires window size $W = 83,333$ in-flight segments -> lot of segments, one might be lost
- What fraction of the segments could be lost to achieve the 10 Gbps rate when using the learned TCP congestion-control?
 - Average throughput in terms of loss rate (L) in basic TCP:

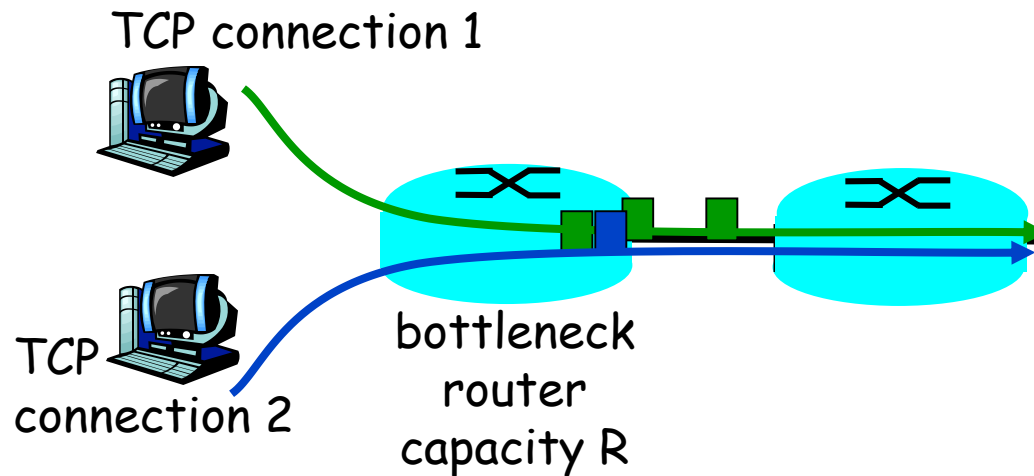
$$\frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

- $\rightarrow L = 2 \cdot 10^{-10}$ *WOW -> one loss event for every 5,000,000,000 segments!*

- New versions of TCP for high-speed needed!

TCP fairness

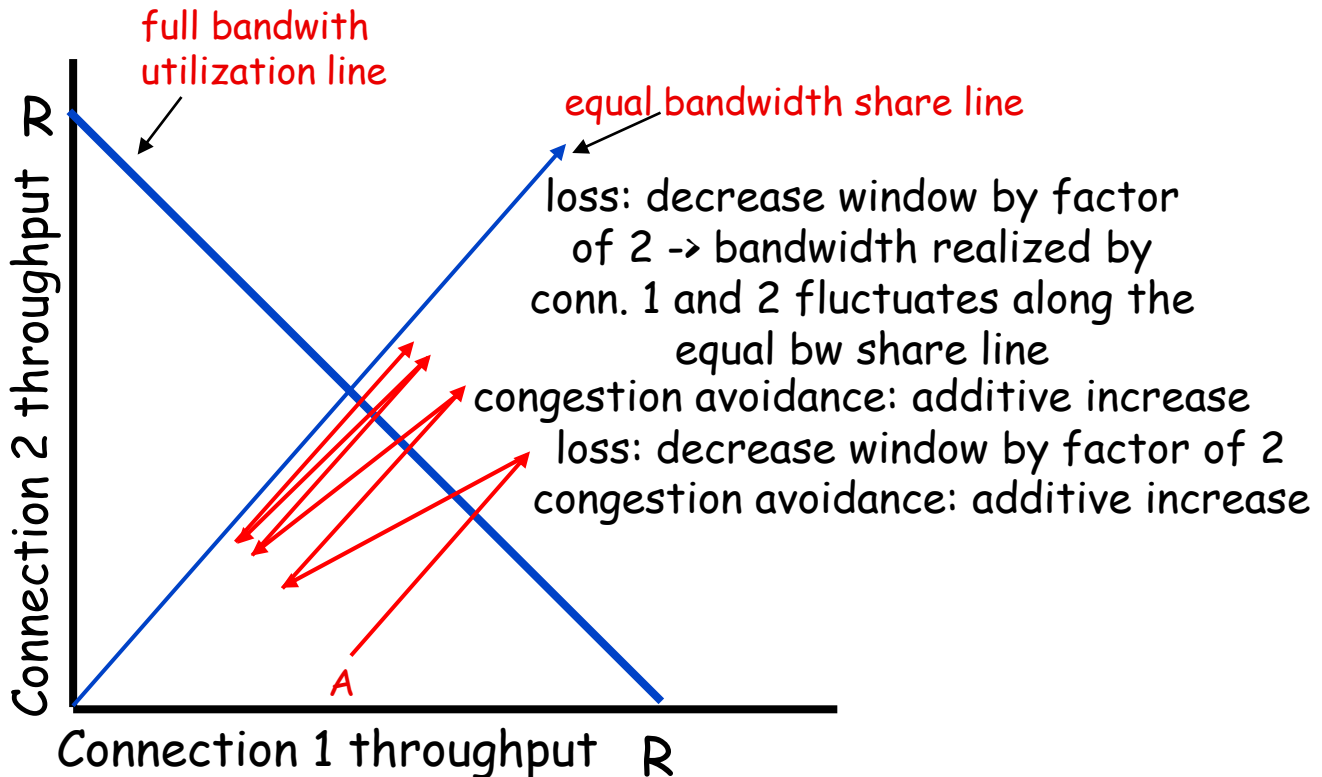
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K

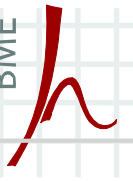


Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughput increases
- Multiplicative decrease decreases throughput proportionally
- Assume that at a given point in time, connection 1 and 2 realize throughputs indicated by point **A**





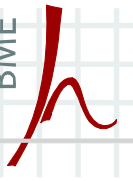
Fairness (more)

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 friendliness

Fairness and parallel TCP connections

- Nothing prevents app from opening parallel connections between 2 hosts
- Web browsers do this
- Example: link of rate R supporting 9 connections
 - New app asks for 1 TCP, gets rate $R/10$
 - New app asks for 11 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”