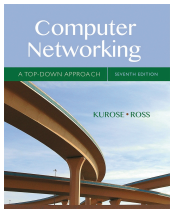
# Chapter 3: Transport Layer

Our goals:

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




*Computer Networking: A Top Down Approach*
7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

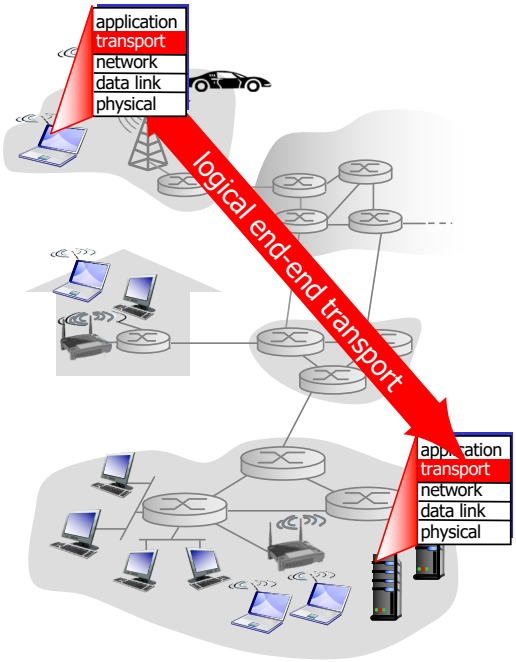


# Chapter 3 outline

- 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 application processes running on different hosts
- transport protocols run in end systems
  - send side: breaks application messages into segments, passes to network layer
  - receive side: reassembles segments into messages, passes to application layer
- more than one transport protocol available to applications
  - Internet: TCP and UDP






# Transport vs. network layer

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

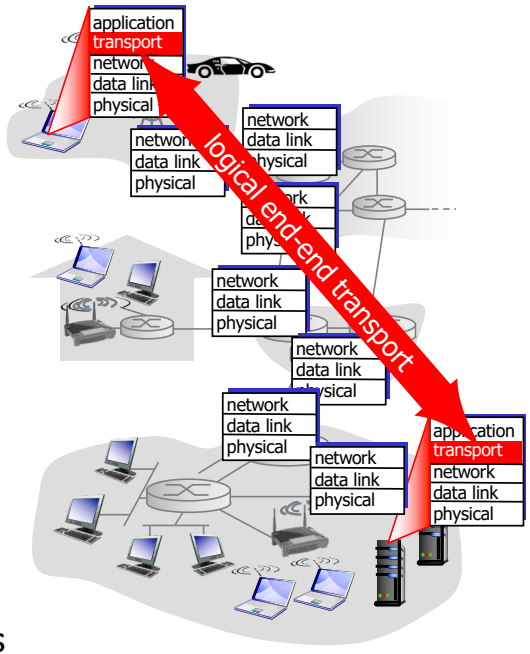
Household analogy:

*12 kids sending letters to 12 kids*

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport layer protocol = Ann and Bill
- network layer protocol = postal service (including mail carriers)

# Internet transport layer protocols

- UDP: unreliable, unordered delivery
  - no-frills extension of "best-effort" IP
- TCP: reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- services not available:
  - delay and bandwidth guarantee
- Other service: Encryption for confidentiality, SSL (Secure Sockets Layer)



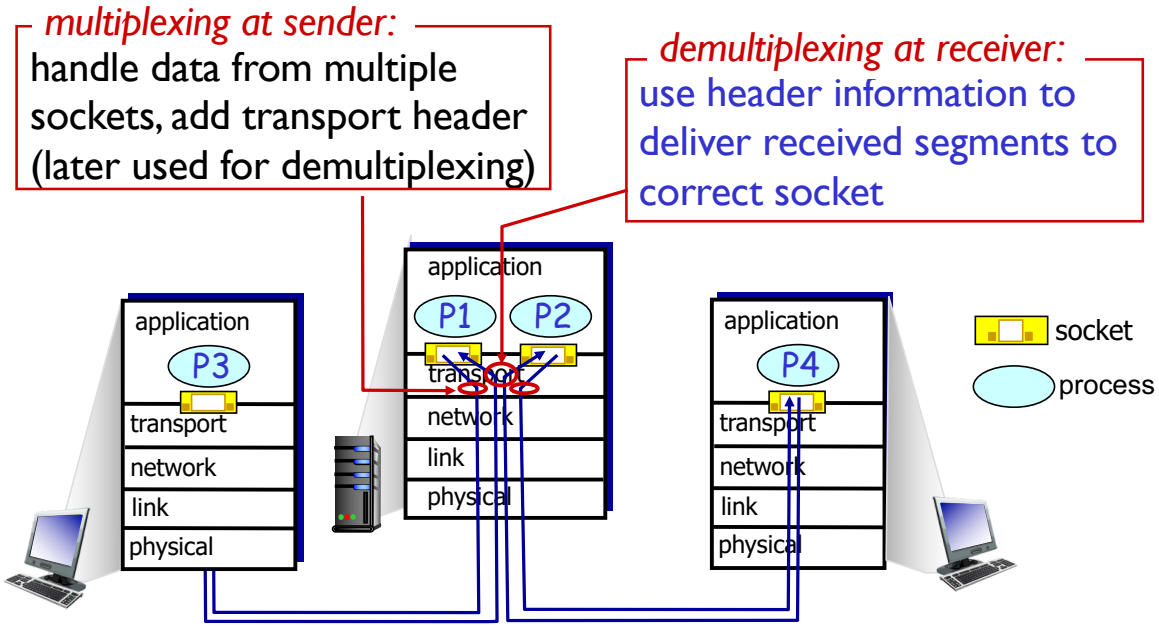



# Chapter 3 outline

- 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

- Extending host-to-host delivery to process-to-process delivery.






# Port Number

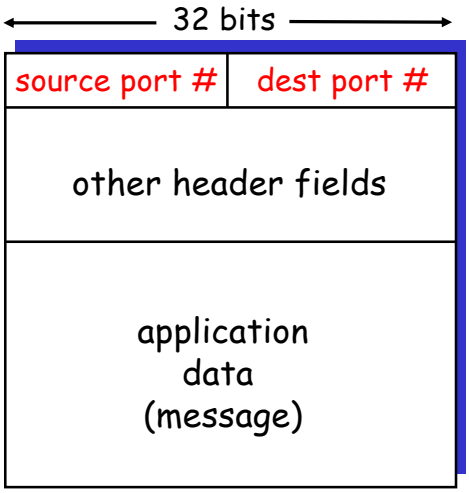
- IANA: Internet Assigned Number Authority.

  http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

- Well-known ports (0 – 1023): Assigned and controlled by IANA.
  - HTTP: 80, FTP: 21, SMTP: 25, DNS: 53
- Registered ports (1024 – 49,151): They are not assigned or controlled by IANA. They can only be registered with IANA to prevent duplication.
  - Microsoft 2000 SQL server: 1434
- Dynamic ports (49,152 – 65,535): Neither controlled nor registered. They can be used by any process.

## How demultiplexing works

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




TCP/UDP segment format



## Connectionless Demultiplexing (UDP)

- *recall:* created socket has host-local port #:

```
clientSocket =
socket(AF_INET, SOCK_DGRAM)
clientSocket.bind(('', 19157))
```

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

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

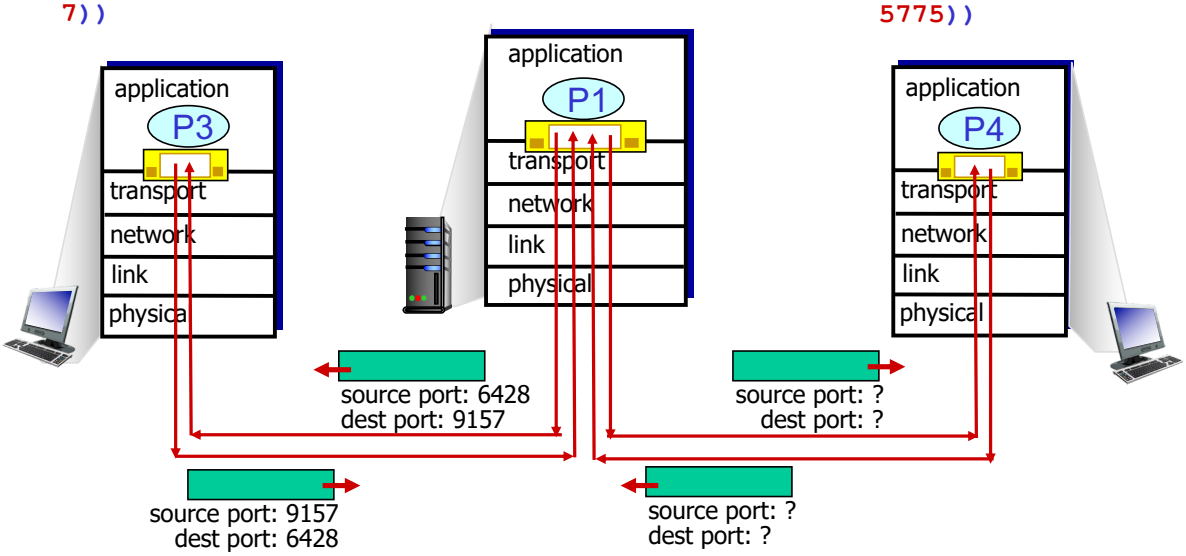
IP datagrams with *same destination port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at destination

# Connectionless demux (cont.)

```
mySocket1 =
 socket(AF_INET,
 SOCK_DGRAM)
mySocket1(('',915
 7))
```

```
serverSocket =
  socket(AF_INET,SOCK_DGRAM)
serverSocket(('',6428))
```

```
mySocket2 =
 socket(AF_INET,
 SOCK_DGRAM)
mySocket2(('',
 5775))
```

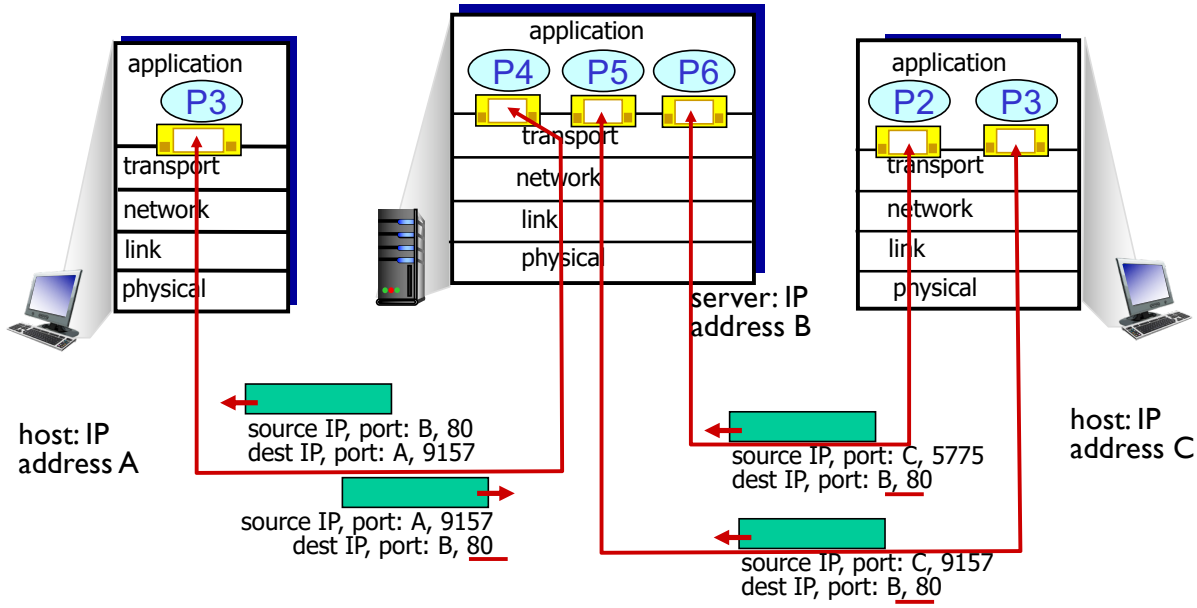





# Connection-oriented demux (TCP)

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

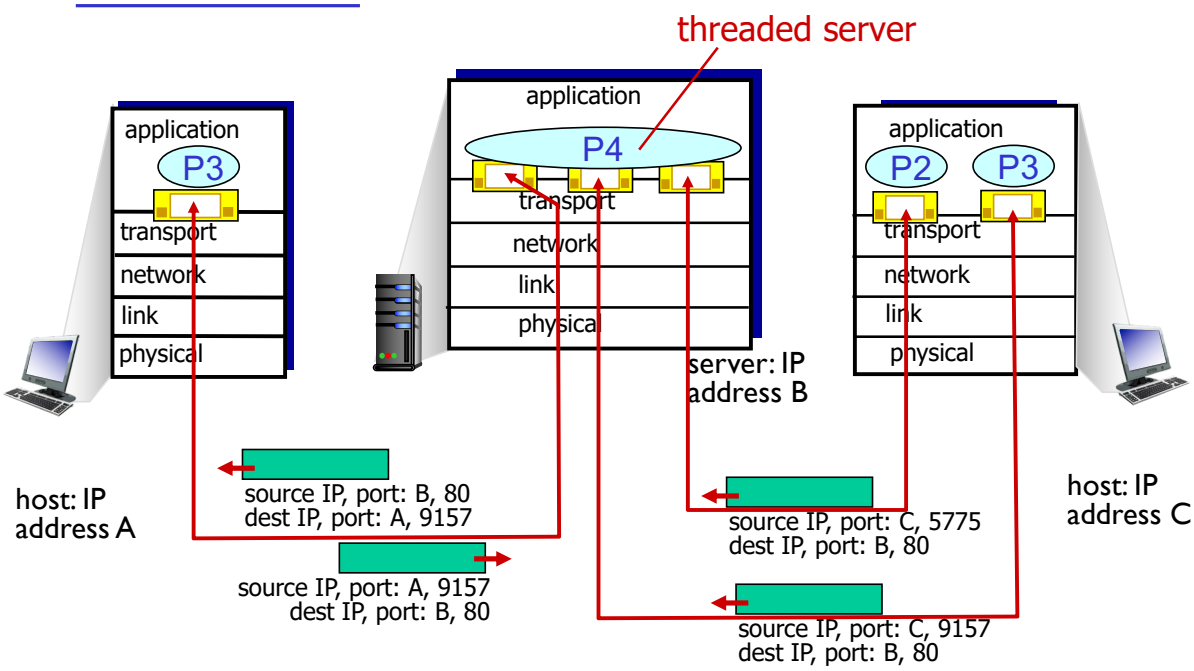
# Connection-oriented demux (cont)




three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets



# Connection-oriented demux: Threaded Web Server



## Chapter 3 outline

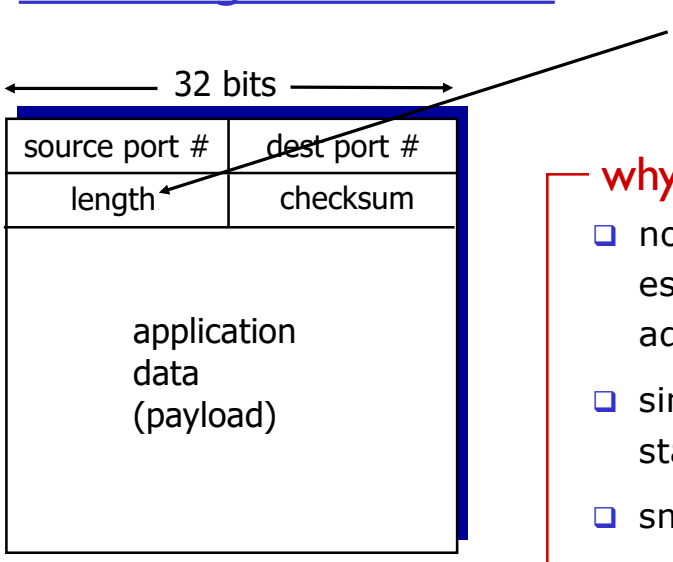
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of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



## UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to application
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- Reliable transfer over UDP
  - add reliability at application layer
  - application-specific error recovery!

# UDP: segment header




UDP segment format

## why is there a UDP?

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



# UDP checksum

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

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1s complement of the sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

# Internet Checksum Example

- When adding numbers, an overflow from the most significant bit needs to be wrapped around.
- Given the three 16-bit words: 0110 0110 01100000, 0101010101010101, 1000111100001100
- The sum of the first two words is

```
  0 1 1 0 0 1 1 0 0 1 1 0 0 0 0 0
  0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
  ---------------------------------
  1 0 1 1 1 0 1 1 1 0 1 1 0 1 0 1
```

- Adding the third word to the sum

```
  1 0 1 1 1 0 1 1 1 0 1 1 0 1 0 1
  1 0 0 0 1 1 1 1 0 0 0 0 1 1 0 0
  ---------------------------------
```

wraparound

Sum

Checksum

If packet has no errors, the sum at rcvr side is:

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

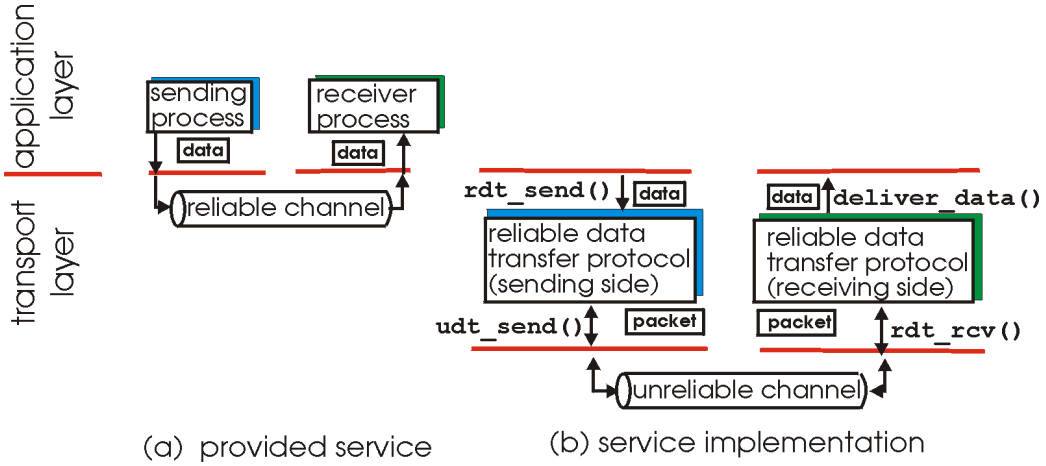


# Chapter 3 outline

- 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 application, transport, link layers

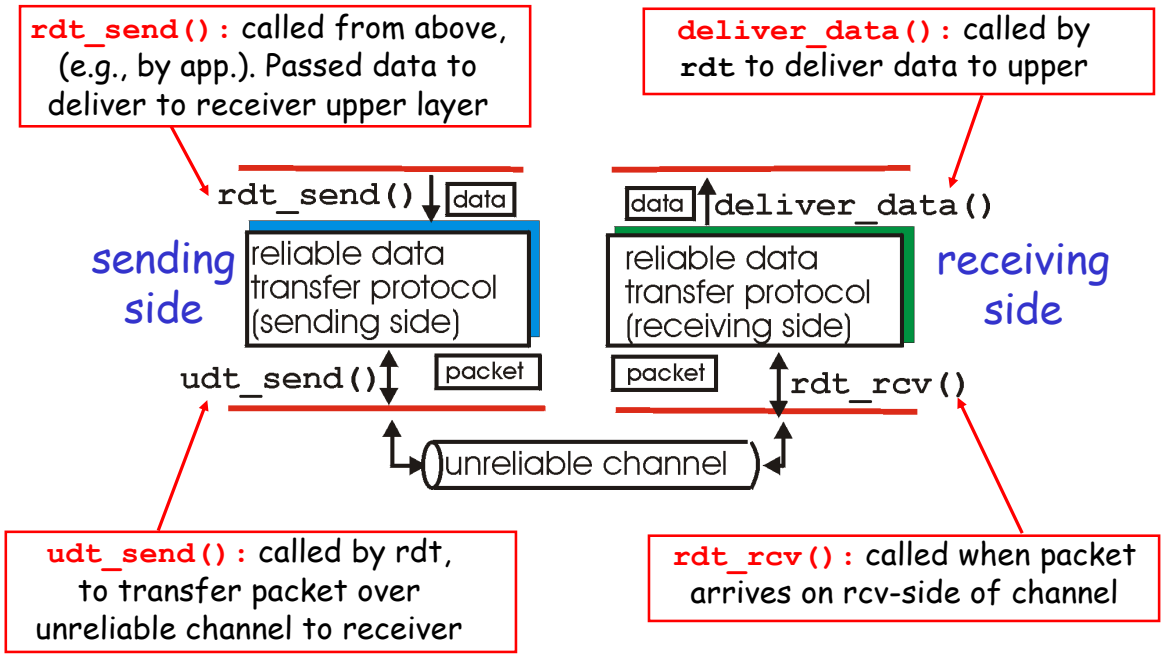



(a) provided service (b) service implementation

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
- Consider only the case of unidirectional data transfer
- Control packets are sent from rcving side to sending side.



# Reliable data transfer: getting started

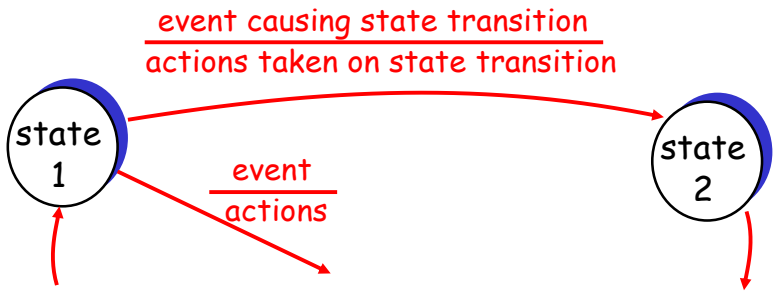


# Reliable data transfer: Getting started

We'll:

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

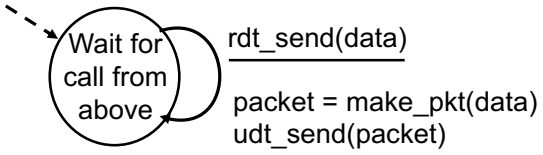
state: when in this "state", the next state is uniquely determined by next event



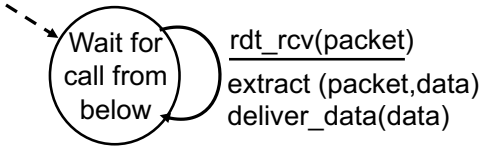



## rdt1.0: Data transfer over a perfectly reliable channel

- underlying channel is 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




rdt1.0: sender side




rdt1.0: receiver side

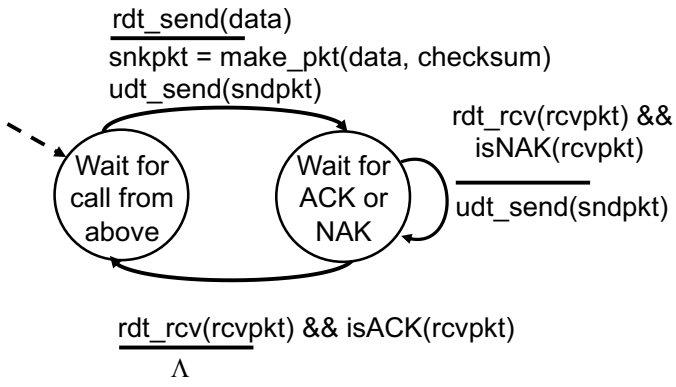
No feedback from receiver to sender

# 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:
  - *Positive acknowledgements (ACKs):* receiver explicitly tells sender that packet received is OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that packet 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
  - retransmission

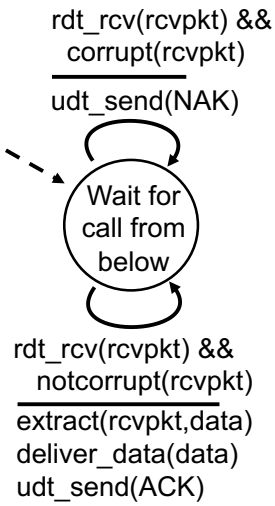


# rdt2.0: FSM specification

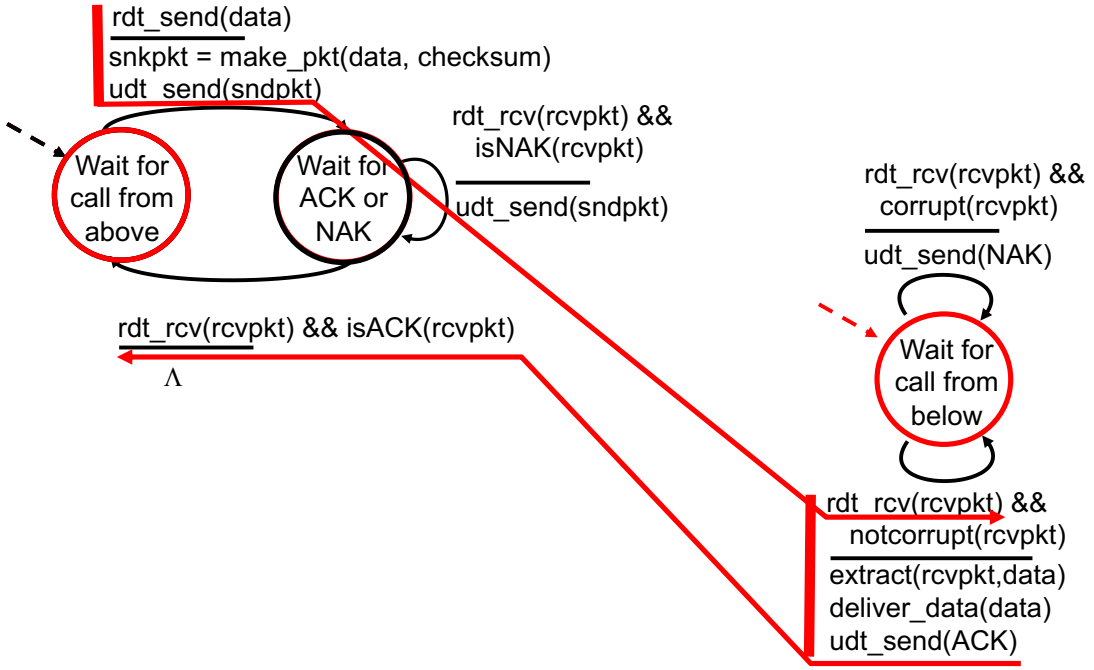



rdt2.0: sender side

rdt2.0: receiver side

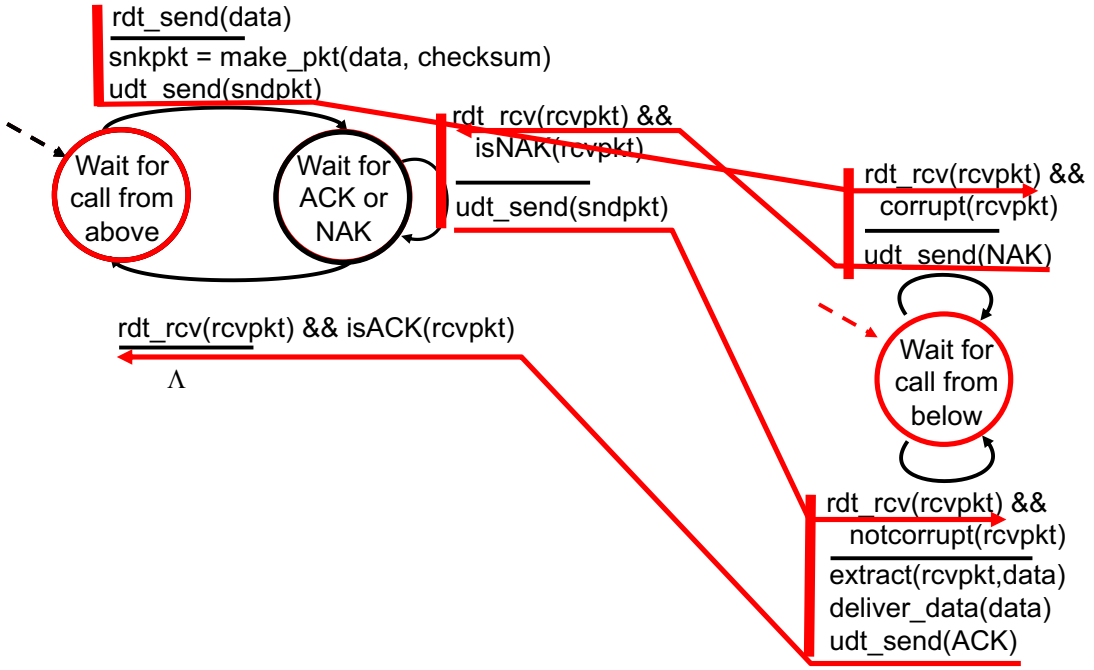


# rdt2.0: operation with no errors






# rdt2.0: error scenario



# rdt2.0 has a fatal flaw!

What happens if ACK/NAK is corrupted?

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

Handling duplicates:

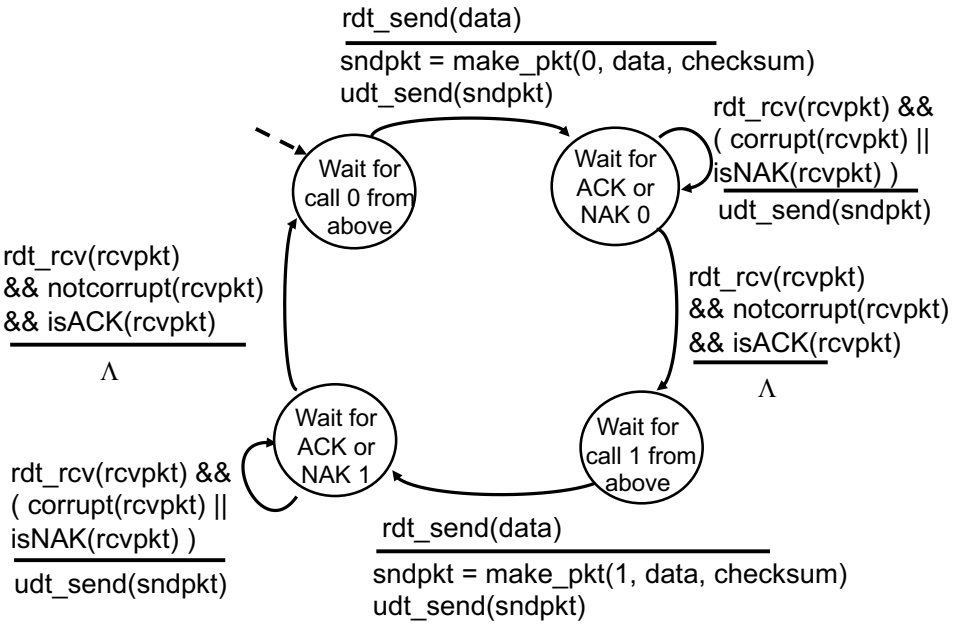
- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

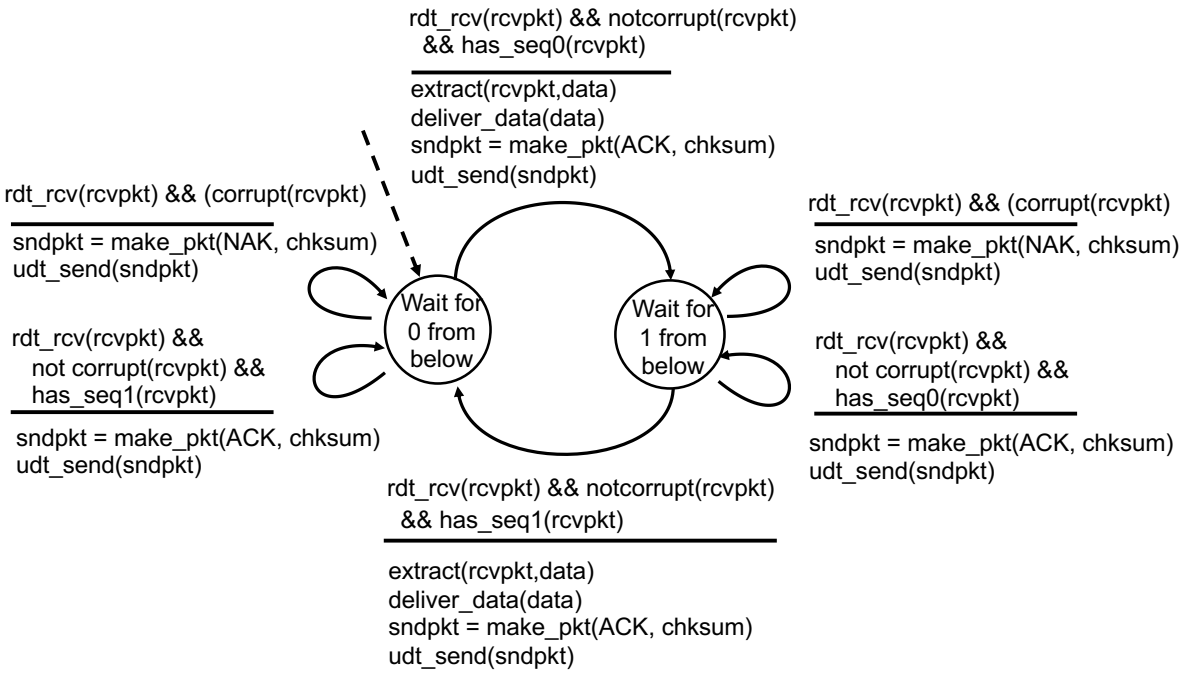
Sender sends one packet, then waits for receiver response



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



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






# rdt2.1: discussion

Sender:

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

Receiver:

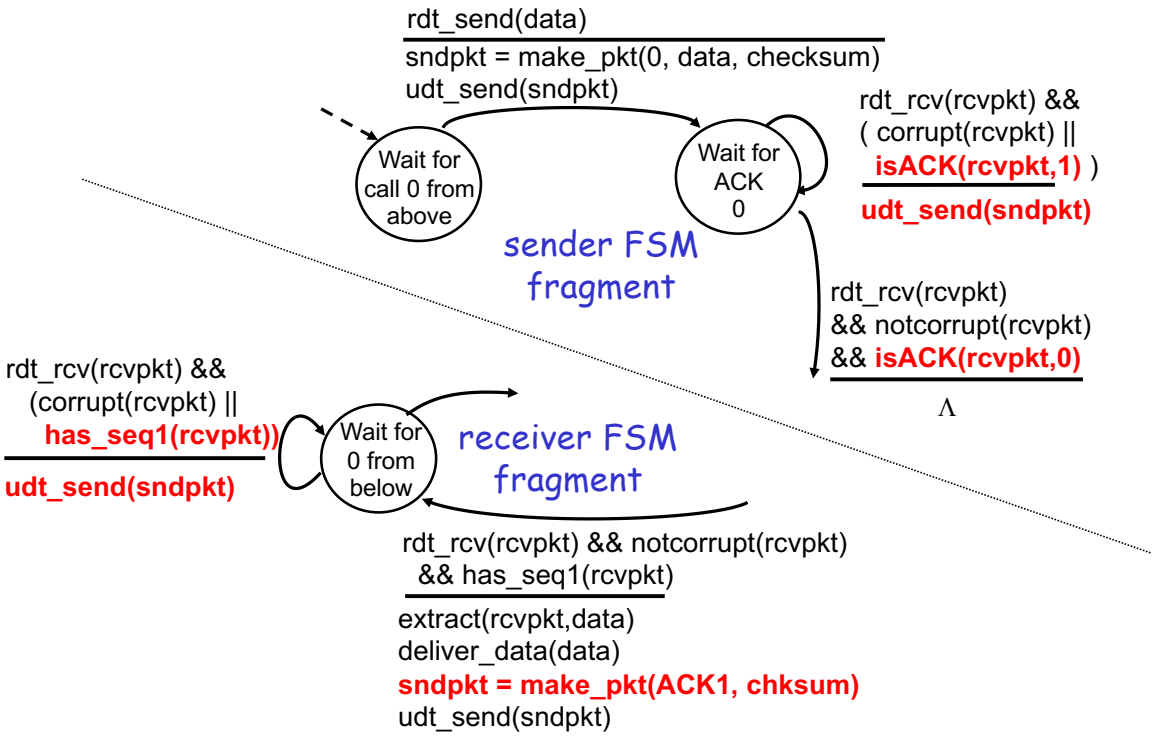
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected packet seq #
- note: receiver *cannot* know if its last ACK/NAK received OK at sender

# rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends an ACK for the last correctly received packet.
  - 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 packets or ACKs)

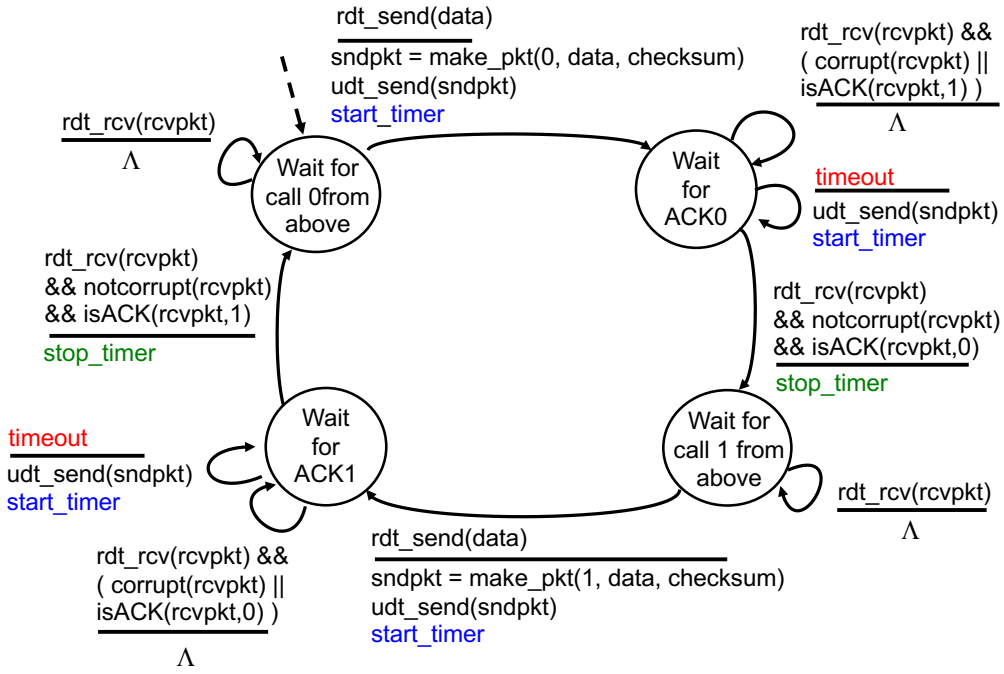
- How to detect packet loss?
- What to do when packet loss occurs?
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

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

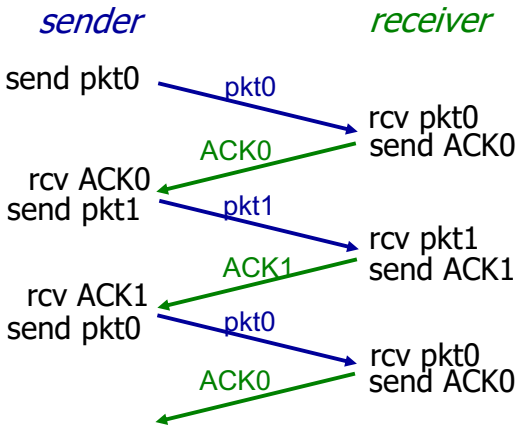
- requires countdown timer
- retransmits if no ACK is received in this time
- if packet (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed



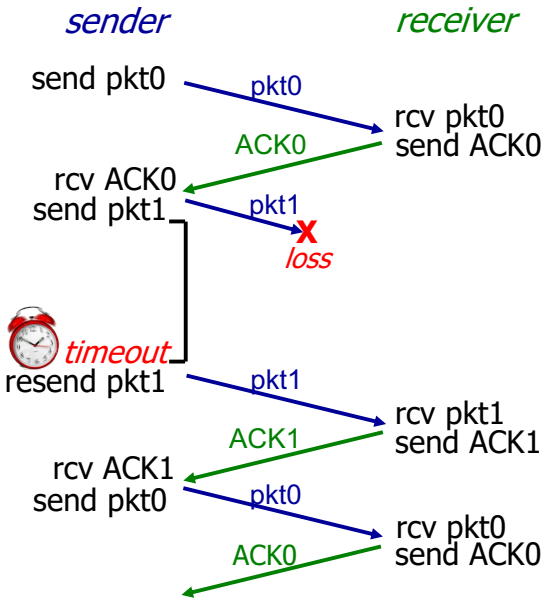
# rdt3.0 sender



# rdt3.0 in action



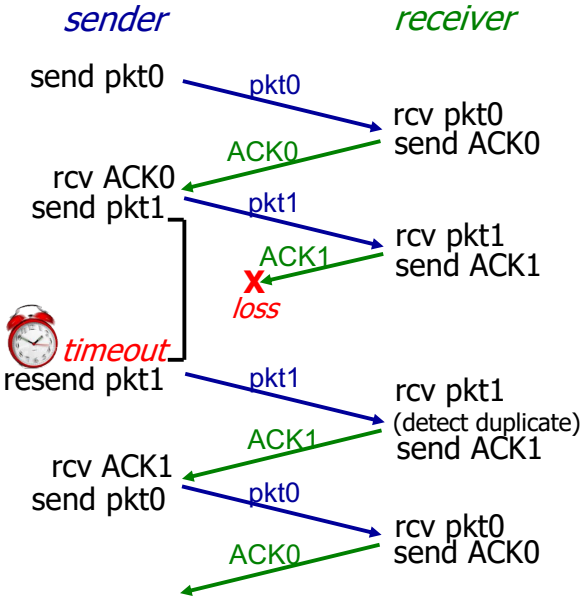

(a) no loss



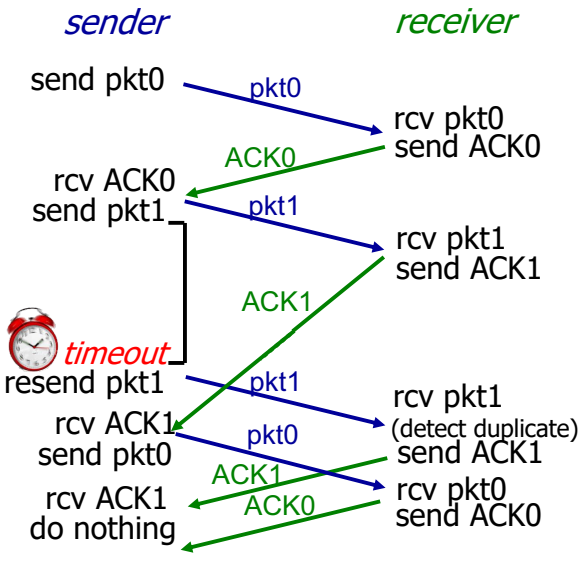

(b) packet loss



# rdt3.0 in action




(c) ACK loss




(d) premature timeout/ delayed ACK

## Performance of rdt3.0

- rdt3.0 works, but performance stinks
- e.g., 1 Gbps link, 30 ms round-trip prop. delay, 1 KB packet

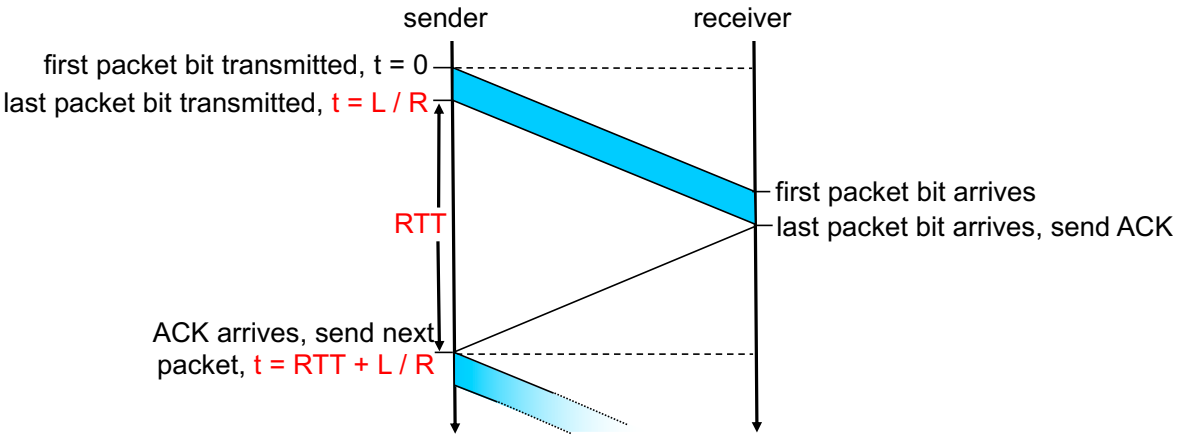
$$d_{trans} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8 \text{ kbits/pkt}}{10^9 \text{ bits/sec}} = 8 \text{ microsec}$$

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

- $U_{sender}$: Utilization (i.e., fraction of time sender busy sending bits)
- 1 KB packet every 30 msec -> 267 kbps effective throughput over 1 Gbps link
- network protocol limits use of physical resources!

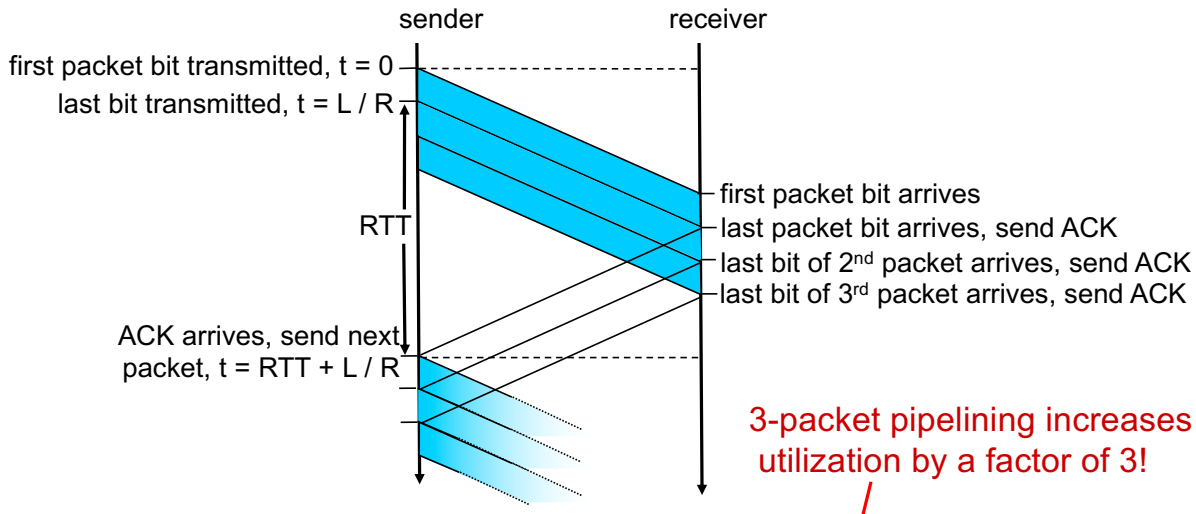


## rdt3.0: stop-and-wait operation




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

# Pipelining: increased utilization




3-packet pipelining increases utilization by a factor of 3!

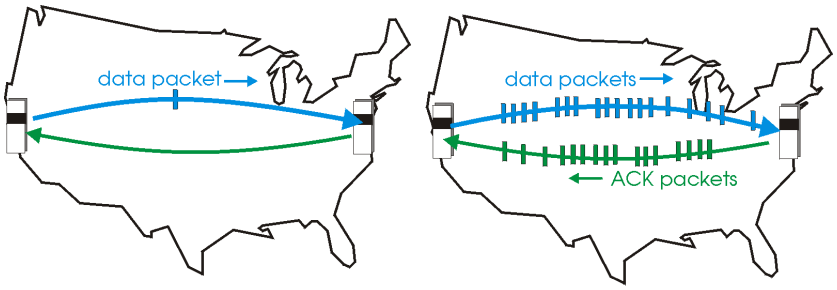
$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$



# 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 Protocols

Go-back-N: big picture

- Sender can have up to *N* unACKed packets in pipeline
- Receiver only sends cumulative ACKs
  - Doesn't ack packet if there's a gap
- Sender has timer for oldest unACKed packet
  - When timer expires, retransmit all unACKed packets

Selective Repeat: big picture

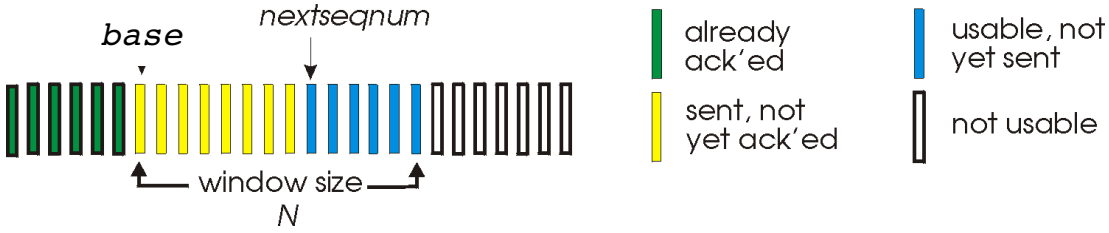
- Sender can have up to *N* unACKed packets in pipeline
- Receiver sends individual ACK for each packet
- sender maintains timer for each unACKed packet
  - When timer expires, retransmit only unACK packet



# Go-Back-N

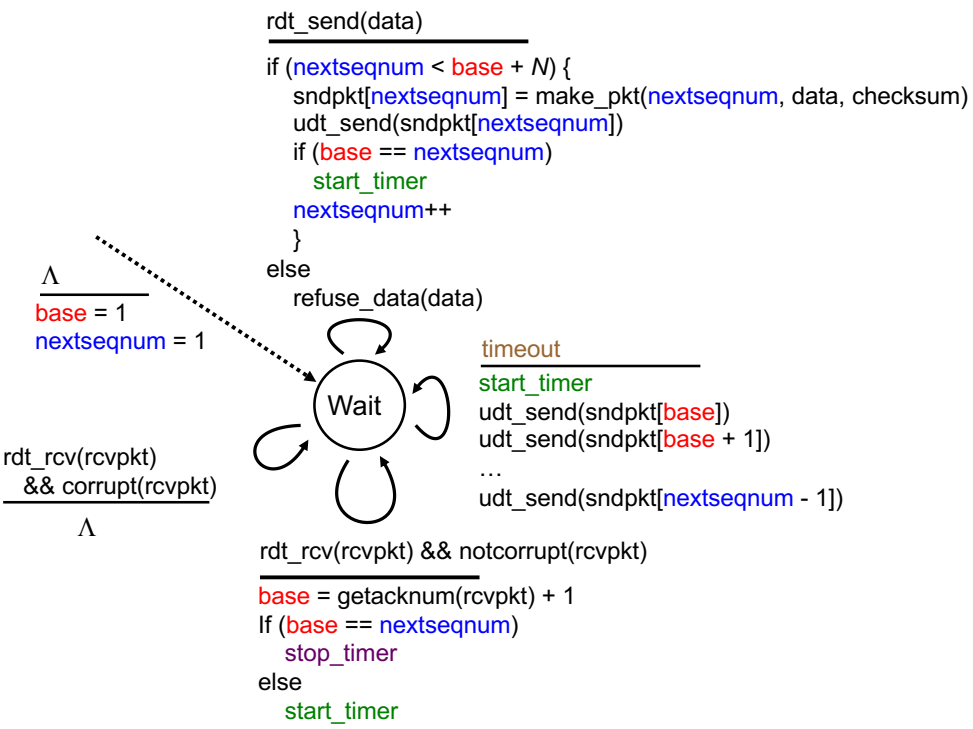
Sender:

- *k*-bit seq # in pkt header. Range [0, $2^k$ – 1]
- "window" of up to *N*, consecutive unack'ed pkts allowed



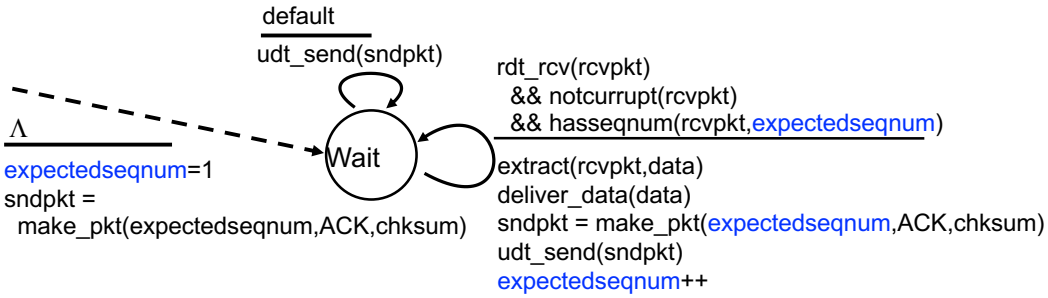

- ACK(*j*): ACKs all pkts up to, including seq # *j* - "cumulative ACK"
  - May receive duplicate ACKs (see receiver)
- Single timer for oldest unACKed packet
- *timeout(j):* retransmit pkt *j* and all higher seq # pkts in window

# GBN: sender extended FSM



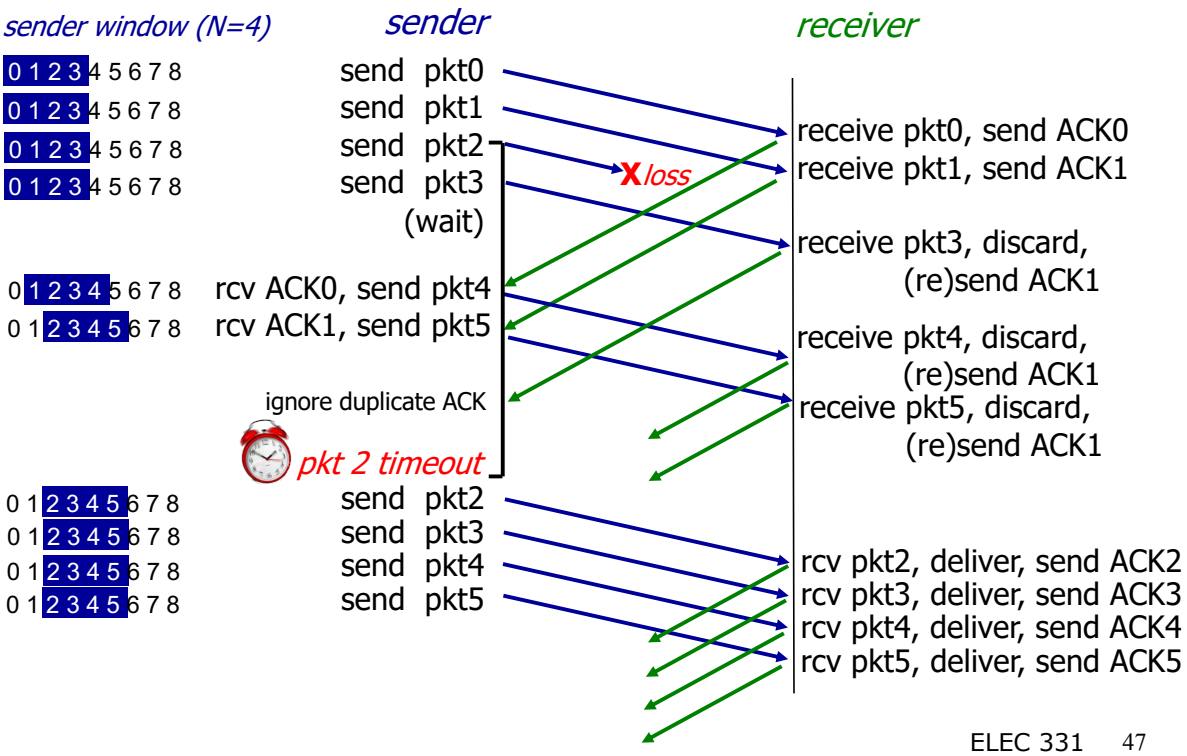



# GBN: receiver extended FSM



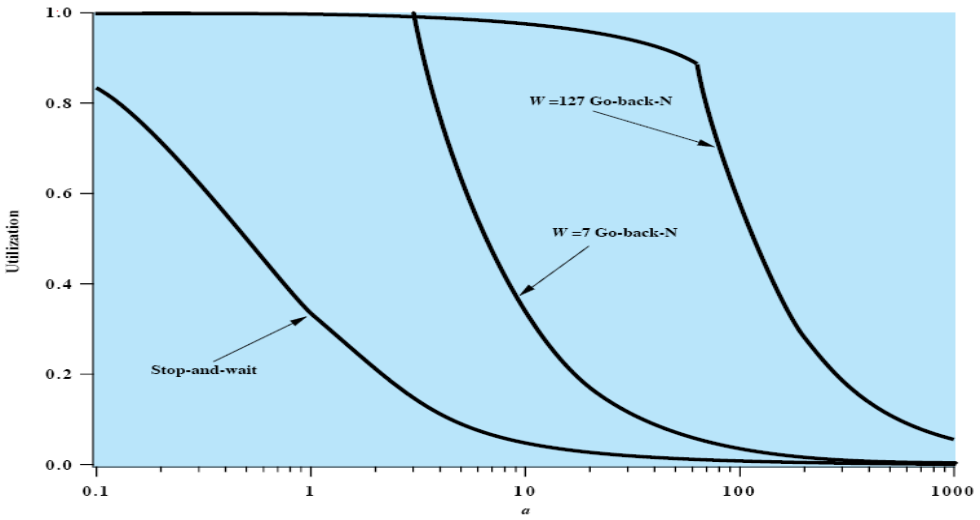

- ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember expectedseqnum
- out-of-order pkt:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

# GBN in action






# Performance of Go Back N




- $W = N$: Window Size
- $a$ = (Propagation Delay)/(Transmission Time)
- Packet error probability = $10^{-3}$. Ignored errors in ACKs.

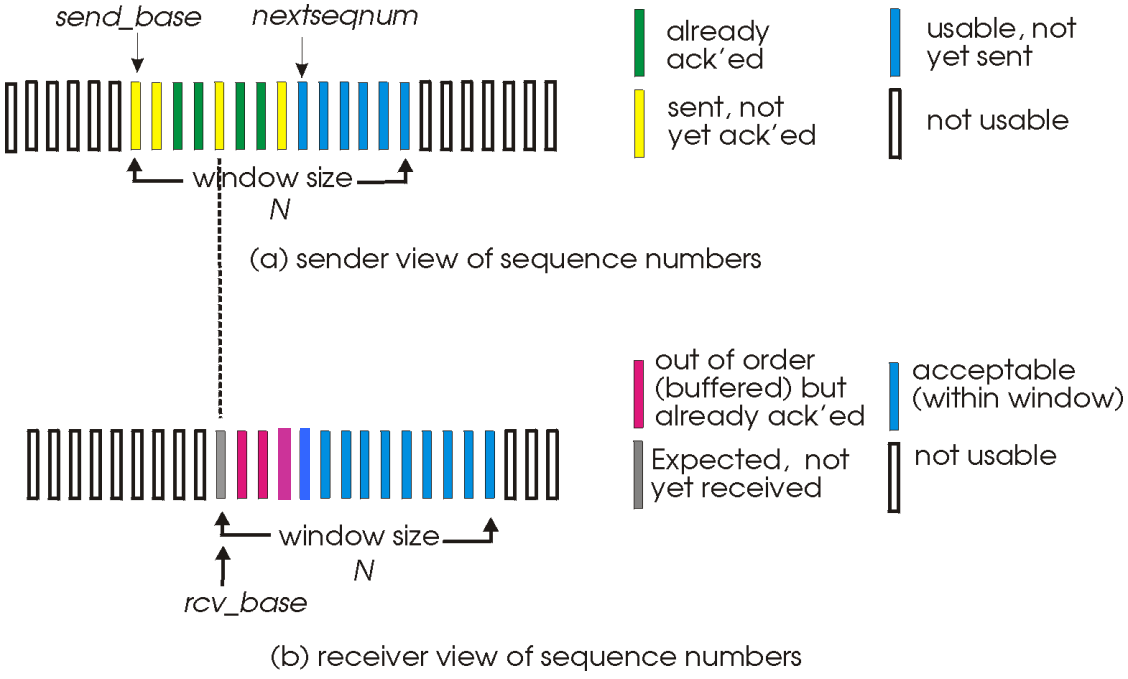
Source: Stallings, 7th edition

# Selective Repeat

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



# Selective repeat: sender, receiver windows



# Selective repeat

**sender**

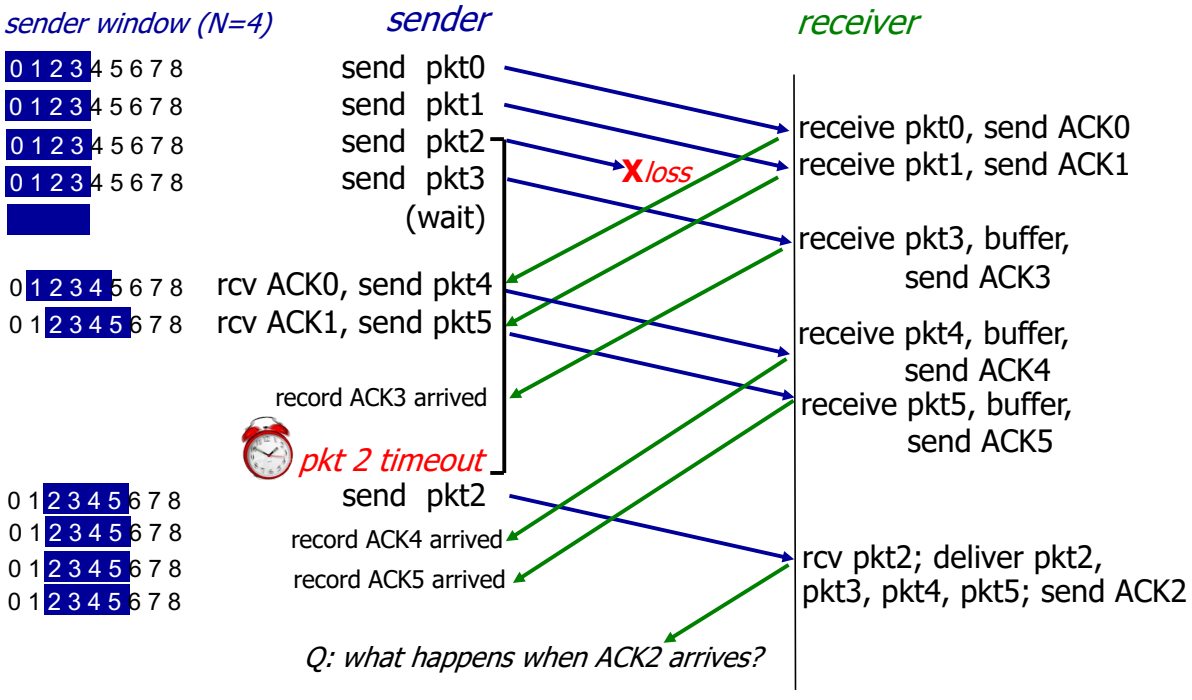
1. Data received from above:
   - if next available seq # in window, send pkt
2. timeout(n):
   - resend pkt n, restart timer
3. ACK(n) in [send_base, send_base+*N*−1]:
   - mark pkt n as received
   - if n smallest unACKed pkt, advance send_base to next unACKed seq #

**receiver**

1. pkt n in [rcv_base, rcv_base+*N*−1] is correctly received
   - send ACK(n)
   - out-of-order: buffer
   - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
2. pkt n in [rcv_base−*N*, rcv_base−1] is correctly received
   - ACK(n)
3. otherwise:
   - ignore



# Selective repeat in action

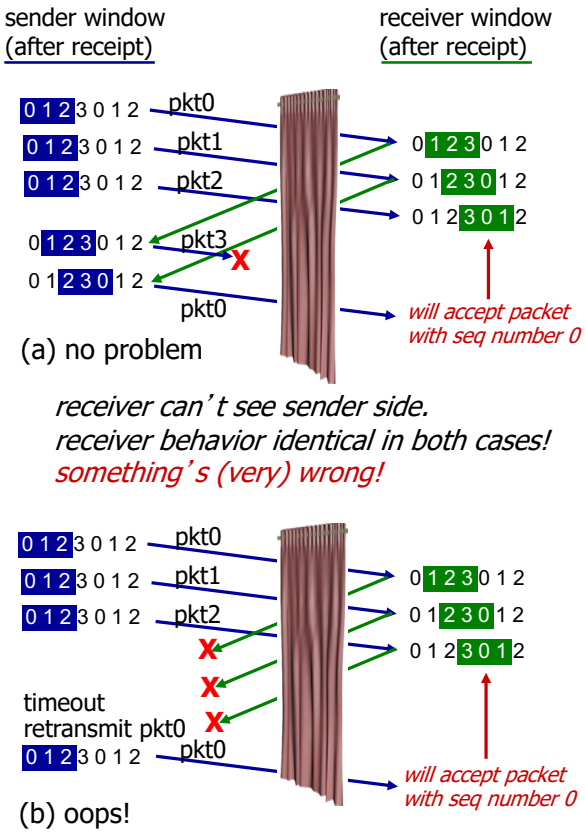


# Selective repeat: dilemma

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?






# 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

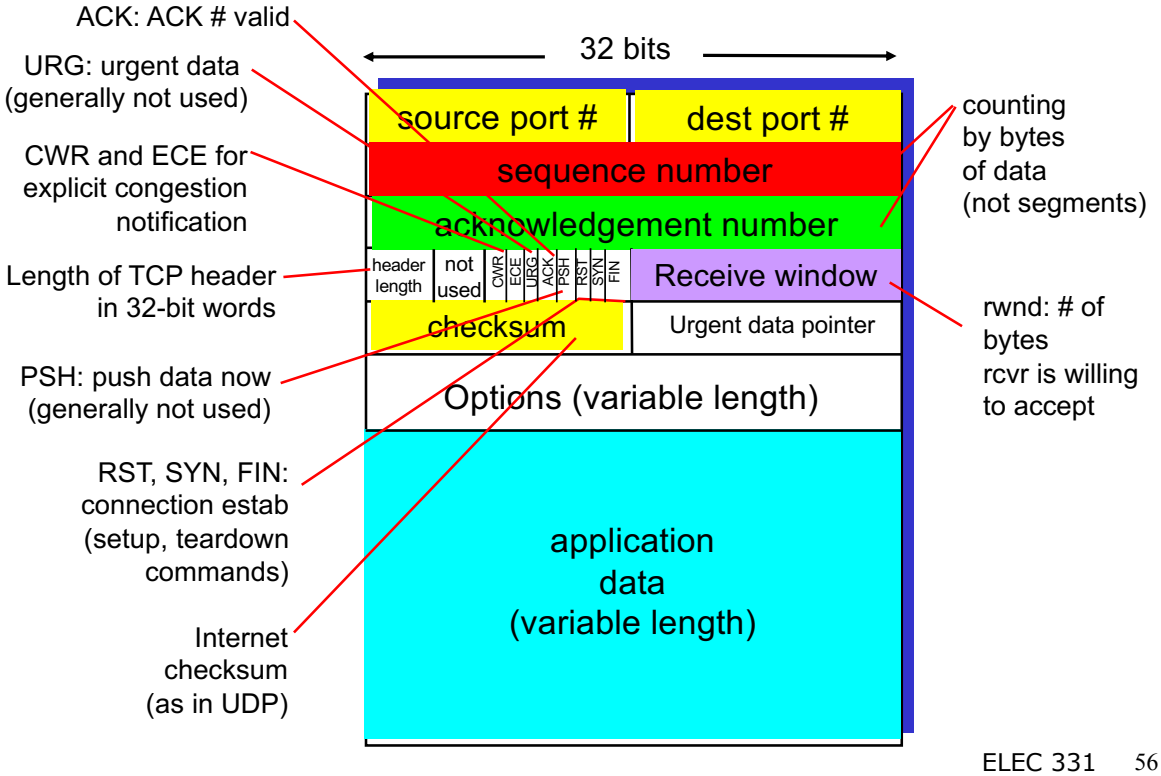
## TCP: Overview (RFCs: 793, 1122, 1323, 2018, 2581)

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


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



## TCP segment structure



# TCP sequence number and ACKs

Seq. # of a segment:

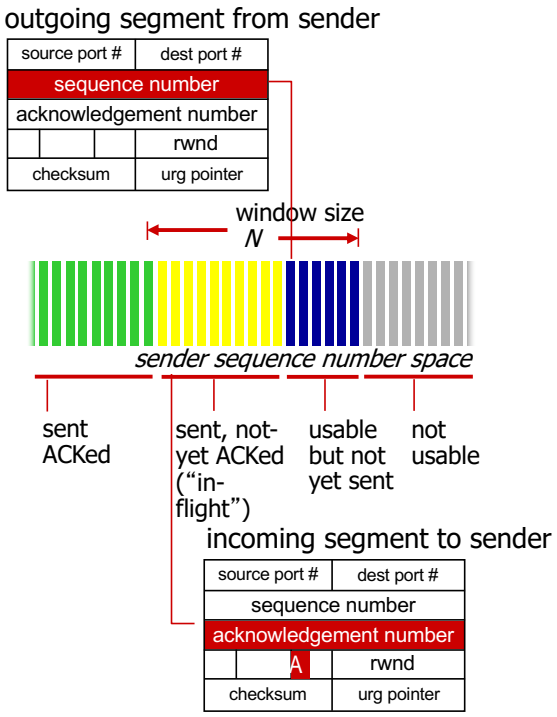
- byte stream "number" of first byte in segment's data

ACKs:

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

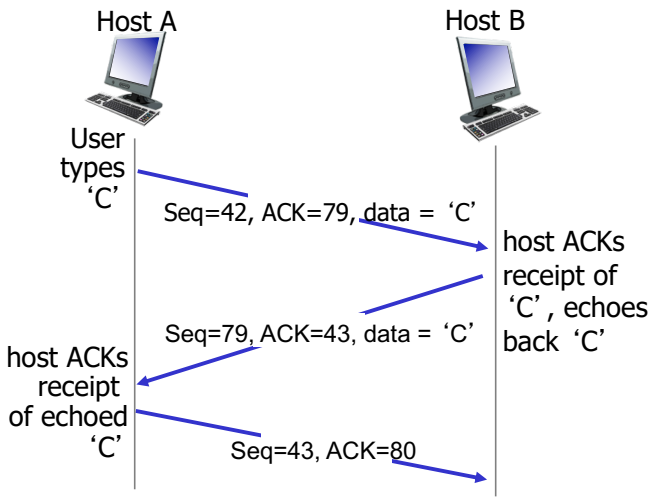
Q: how receiver handles out-of-order segments

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






# TCP sequence number and ACKs




simple telnet scenario

# TCP round trip time estimation and timeout

Q: how to set TCP timeout value?

- longer than round trip time (RTT)
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

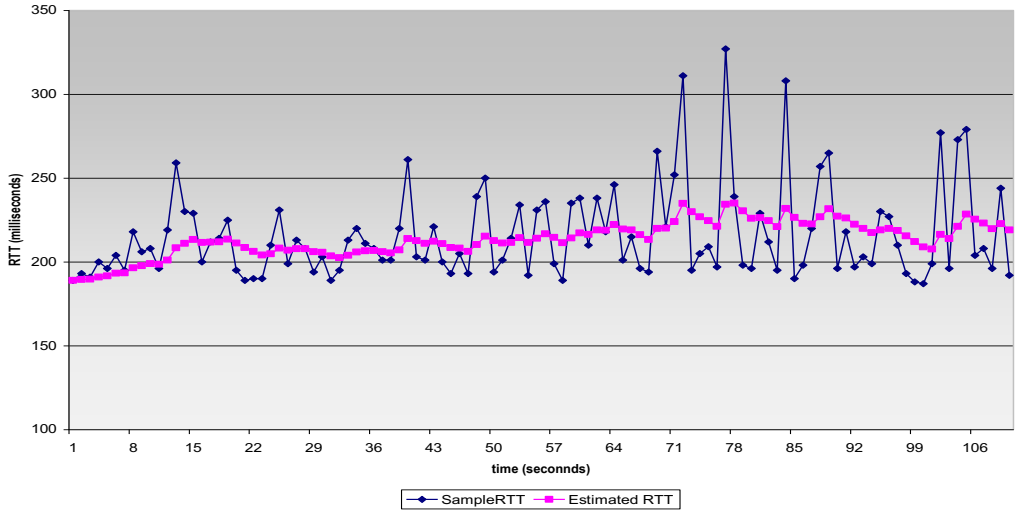
- **SampleRTT:** measured time when segment is sent (passed to IP) until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **SampleRTT**



# TCP round trip time estimation and 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 estimation and timeout

## Setting the timeout interval

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

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

(typically, β = 0.25)

Then, set the timeout interval:

**TimeoutInterval = EstimatedRTT + 4*DevRTT**

estimated RTT

"safety margin"



# 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 (rdt) service

- 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 above:

- 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`

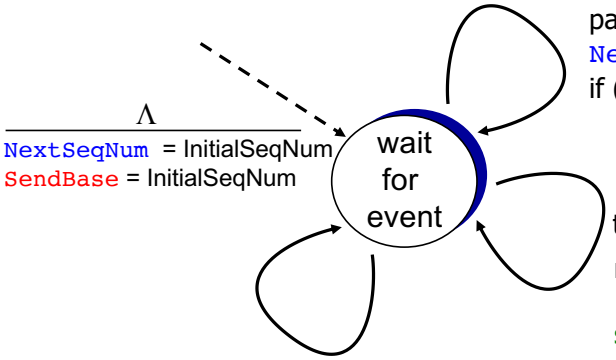
Timer timeout:

- 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 still unacked segments

# TCP sender (simplified)




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

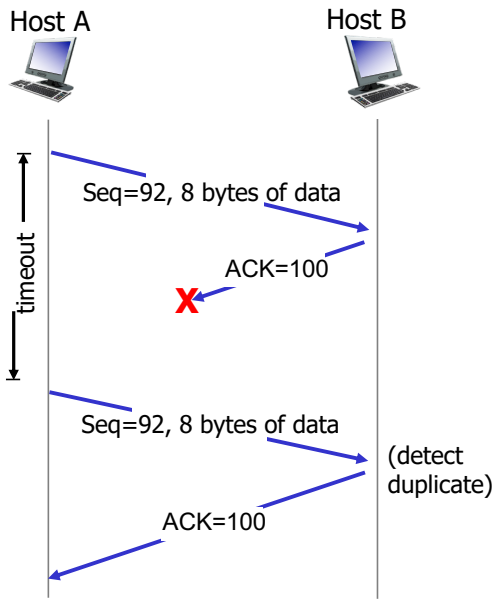
timer timeout
retransmit not-yet-acked segment with smallest seq. #
start timer

ACK received, with ACK field value y

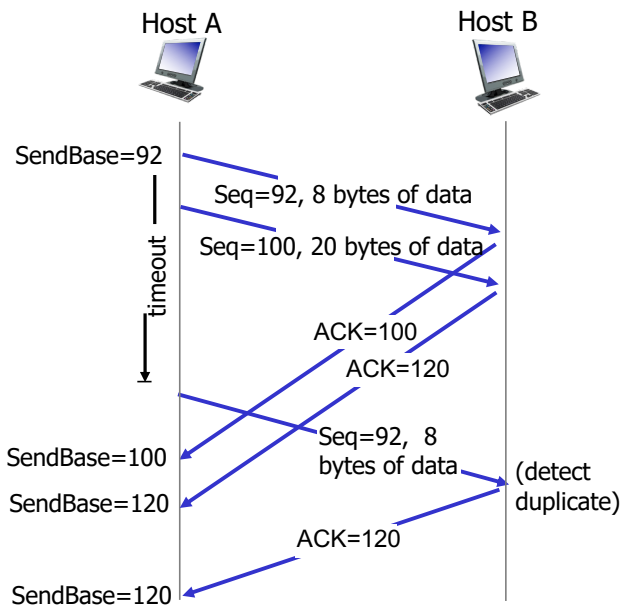
```
if (y > SendBase) {
   SendBase = y
   /* SendBase – 1: last cumulatively ACKed byte */
   if (there are currently any not-yet-acked segments)
       start timer
   else stop timer
    }
```



# TCP: retransmission scenarios

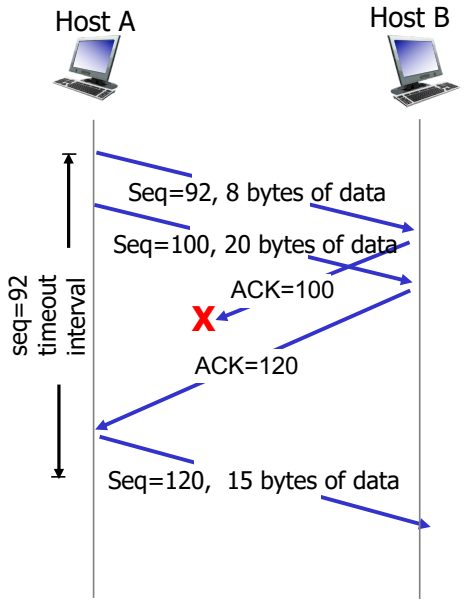



lost ACK scenario




premature timeout

# TCP retransmission scenarios (cont.)




cumulative ACK



# Doubling the Timeout Interval

- Each time TCP re-transmits, it sets the next timeout interval to twice of previous value,
  - rather than deriving it from the last **`EstimatedRTT`** and **`DevRTT`**
- E.g., **`TimeoutInterval`** = 0.75 sec before retransmission
  - After 1st retransmission, **`TimeoutInterval`** = 1.5 sec
  - After 2nd retransmission, **`TimeoutInterval`** = 3 sec
- After an ACK is received, **`TimeoutInterval`** is derived from the most recent values of **`EstimatedRTT`** and **`DevRTT`**.
- Limited form of congestion control.

# TCP ACK Generation Recommendation [RFC 5681]

| 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 500 ms 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 (which is lower end of gap) |
| Arrival of segment that partially or completely fills gap | Immediately 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.

*TCP fast retransmit*

if sender receives 3 duplicate ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

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

# Fast Retransmit Algorithm

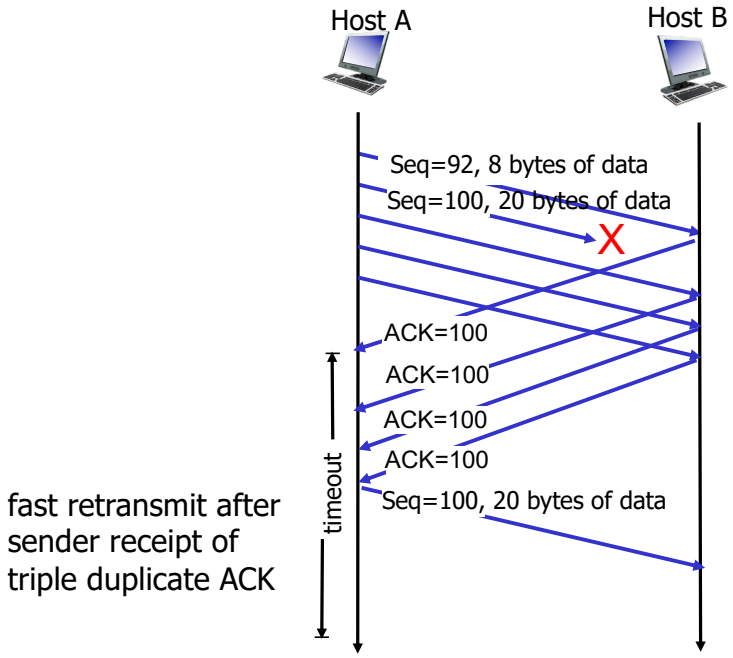
```
event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently any not-yet-acknowledged segments)
                start timer
            else stop timer
          }
        else {
            increment number of duplicate ACKs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

a duplicate ACK for already ACKed segment

TCP Fast Retransmit



# Fast Retransmit based on Duplicate ACKs




fast retransmit after sender receipt of triple duplicate ACK

# Chapter 3 outline

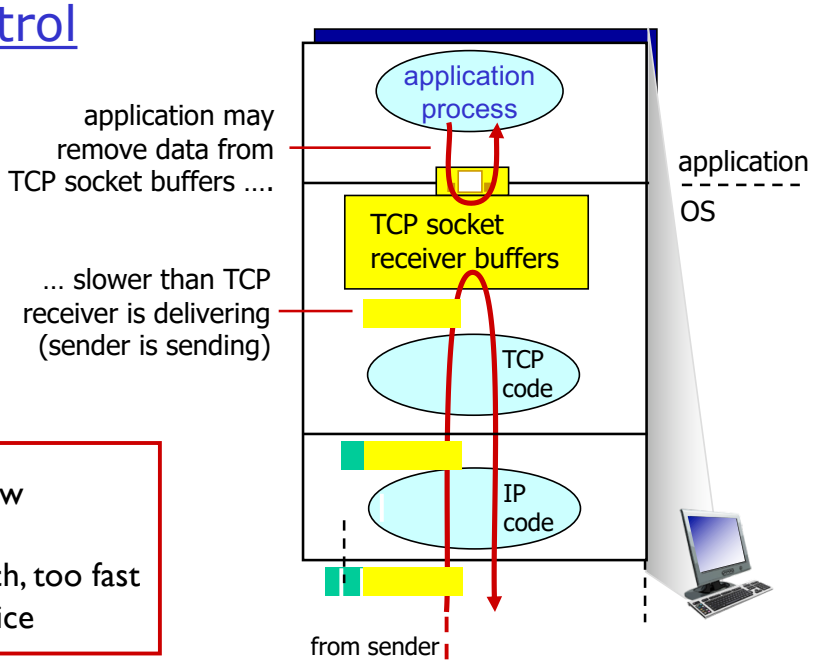
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of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



# TCP Flow Control

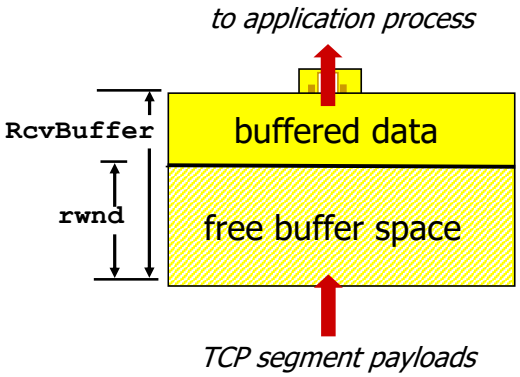
*flow control*

- sender won't overflow receiver's buffer by transmitting too much, too fast
- speed-matching service




receiver protocol stack

# TCP flow control




*receiver-side buffering*

- spare room in buffer

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

- Receiver advertises spare room by including value of **`rwnd`** in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- Sender limits unACKed data to **`rwnd`**
  - guarantees receive buffer doesn't overflow

```
LastByteSent - LastByteAcked
  ≤ rwnd
```



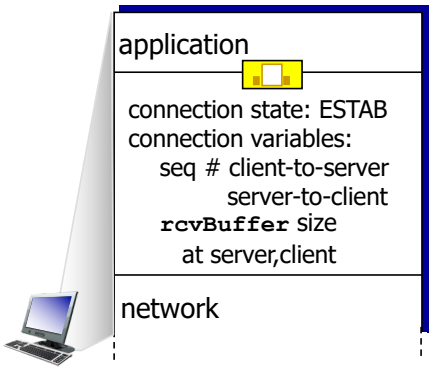
# Chapter 3 outline

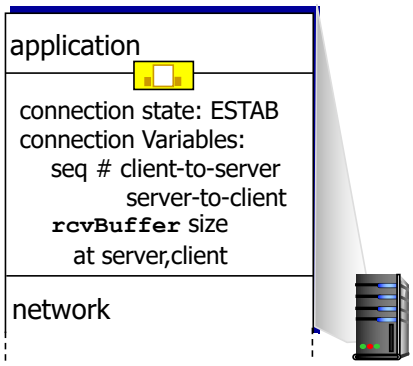
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of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

## TCP Connection Management

before exchanging data, sender/receiver "handshake":

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



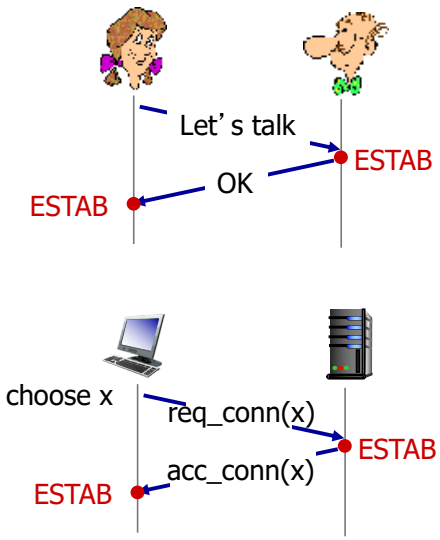

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,
   serverPort))
```

```
serverSocket = socket(AF_INET,
   SOCK_STREAM)
serverSocket.bind(('',serverPort)
connectionSocket, addr =
   serverSocket.accept()
```



## Agreeing to establish a connection

2-way handshake:

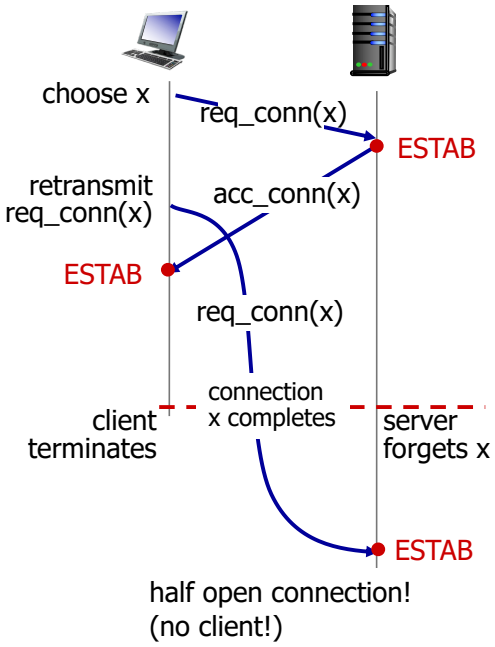


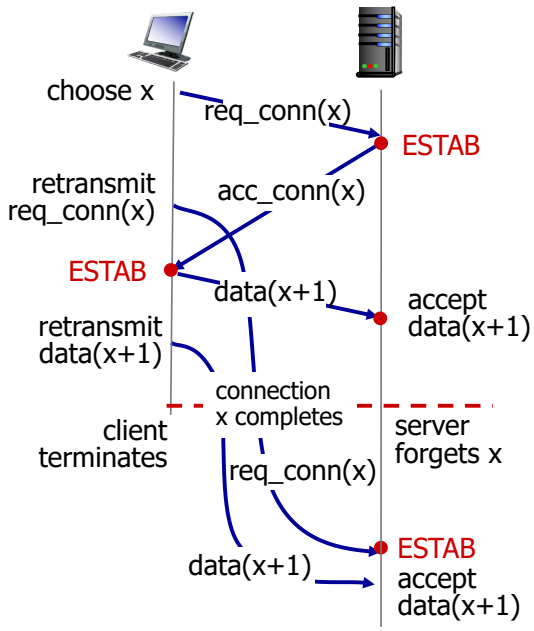

Q: Will 2-way handshake always work in network?

- 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

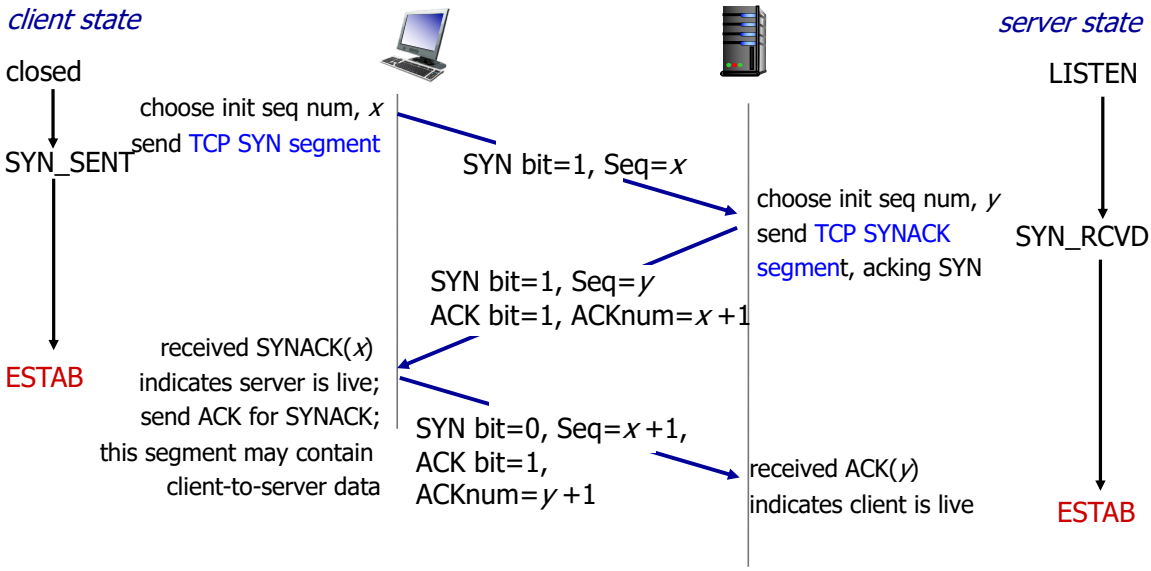
2-way handshake failure scenarios:



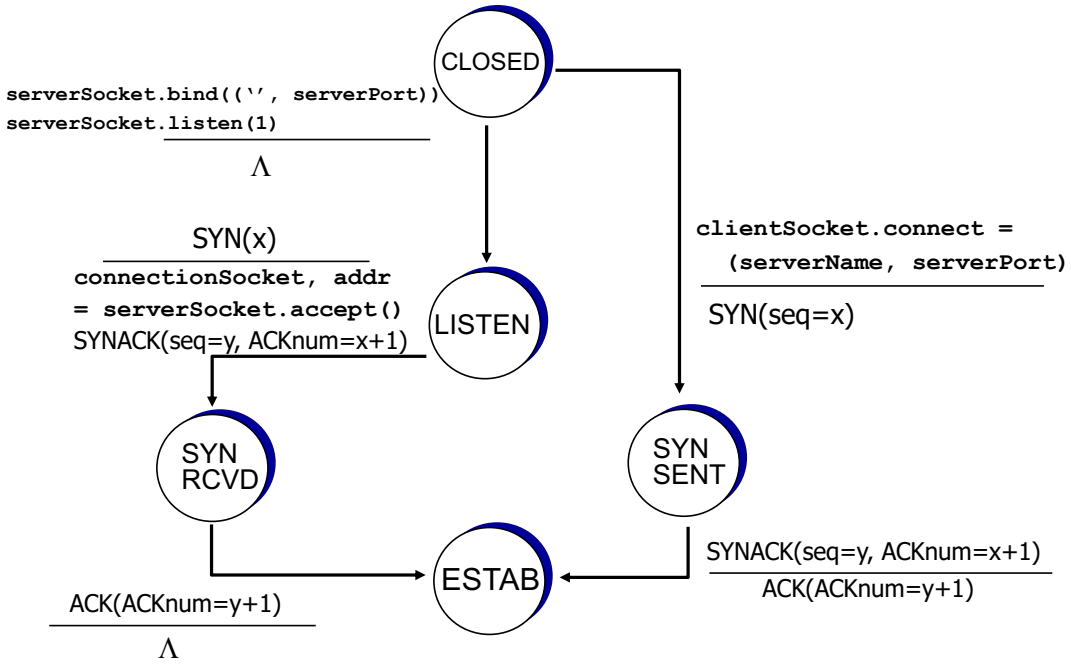






# TCP 3-way handshake



# TCP 3-way handshake: FSM

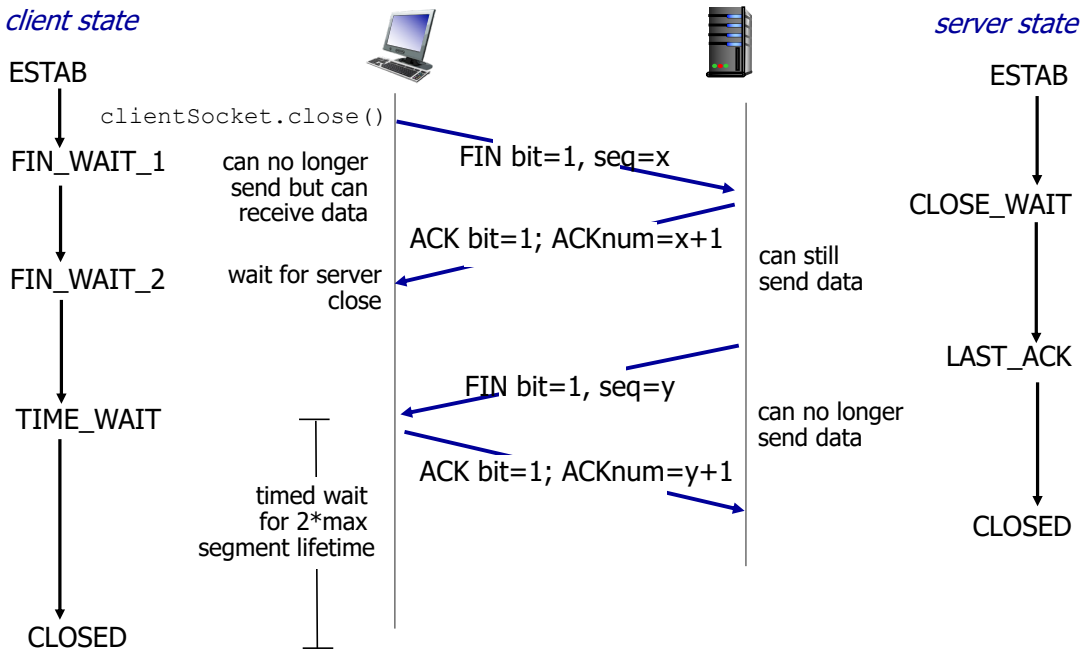





# TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

# TCP: closing a connection






# 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!



# 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

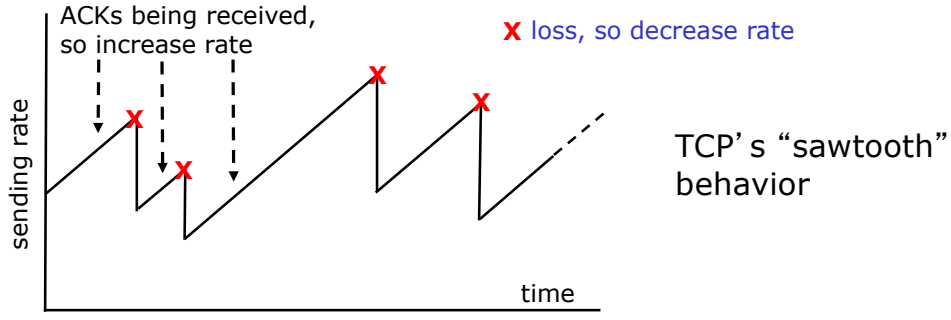
# TCP Congestion Control

- Goal: TCP sender should transmit as fast as possible, but without congesting network
  - Q: How to find the rate just below congestion level?
- Decentralized: each TCP sender sets its own rate, based on implicit feedback:
  - ACK: segment received (a good thing!), network not congested, so increase sending rate
  - lost segment: assume loss due to congestion, so decrease sending rate



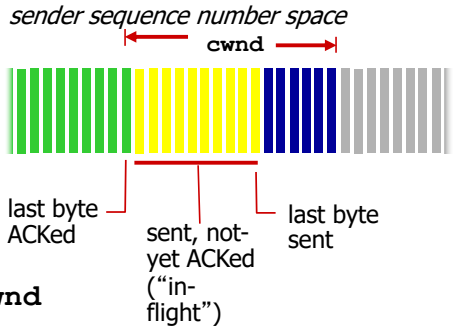
# TCP congestion control: Bandwidth probing

- "probing for bandwidth": increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network)




- Q: how fast to increase/decrease?
  - details to follow

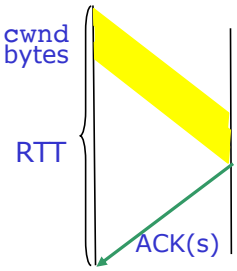
# TCP Congestion Control: Details




- Congestion window: **cwnd**
- sender limits rate by limiting number of unACKed bytes "in pipeline":

**LastByteSent - LastByteAcked**

**≤ min{rwnd, cwnd}**




- Roughly,

$$\text{rate} \sim \frac{\text{cwnd}}{\text{RTT}} \text{ Bytes/sec}$$

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



# TCP Congestion Control: More Details

Loss event: reduce **cwnd**

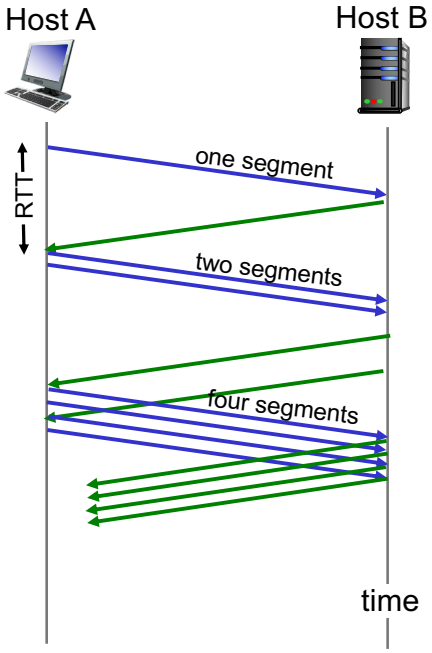
- timeout: no response from receiver
  - cut **cwnd** to 1 MSS
- 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
  - cut **cwnd** in half, less aggressively than on timeout

ACK received: increase **cwnd**

- Slow Start phase:
  - increase exponentially fast (despite name) at connection start, or following timeout
- Congestion Avoidance:
  - increase linearly

## TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially `cwnd` = 1 MSS
  - e.g., if MSS = 500 bytes, RTT = 200 ms, initial rate = 20 kbps
  - increment `cwnd` by one MSS for each new ACK received
  - result in double `cwnd` every RTT
- summary: initial rate is slow but ramps up exponentially fast





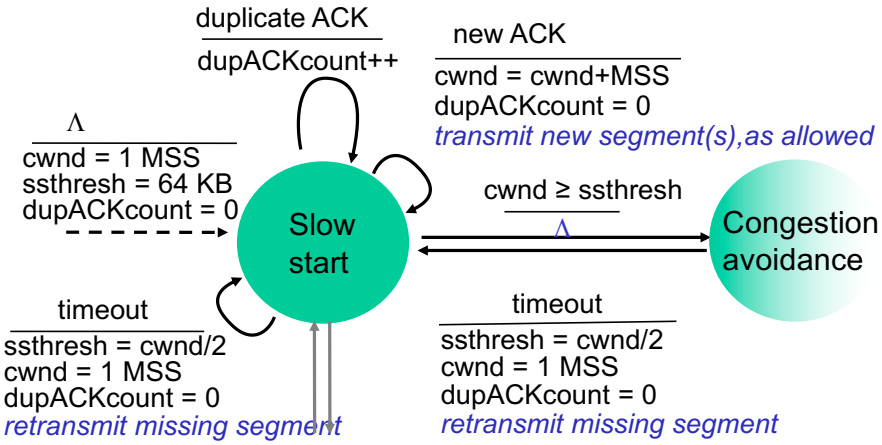

## TCP: detecting, reacting to loss

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

## Transferring into/out of Slow Start

**ssthresh: cwnd** threshold maintained by TCP

- on loss event: set **ssthresh to cwnd/2**
  - remember (half of) TCP rate when congestion last occurred
- when **cwnd ≥ ssthresh**: transition from slow start to congestion avoidance phase






## TCP: Congestion avoidance

- when **cwnd > ssthresh**, increase **cwnd** linearly
  - increase **cwnd** by 1 MSS per RTT
- Implementation:

**cwnd = cwnd + MSS(MSS/cwnd)**

for each new ACK received

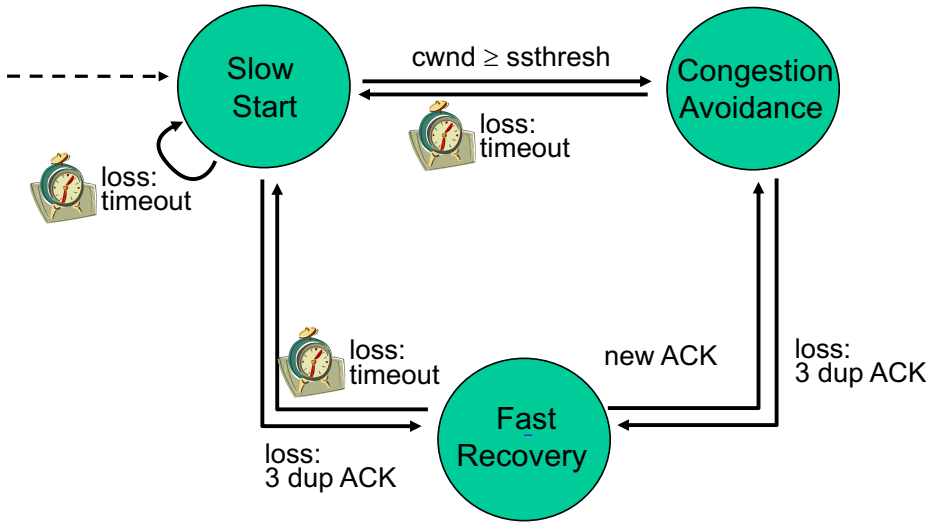
- e.g., **MSS** = 1460 bytes, **cwnd** = 14600 bytes

| | cwnd (bytes) |
|---|---|
| | 14600 |
| 1st new ACK | 14746 |
| 2nd new ACK | 14891 |
| … | … |
| 10th new ACK | 16000 |

AIMD

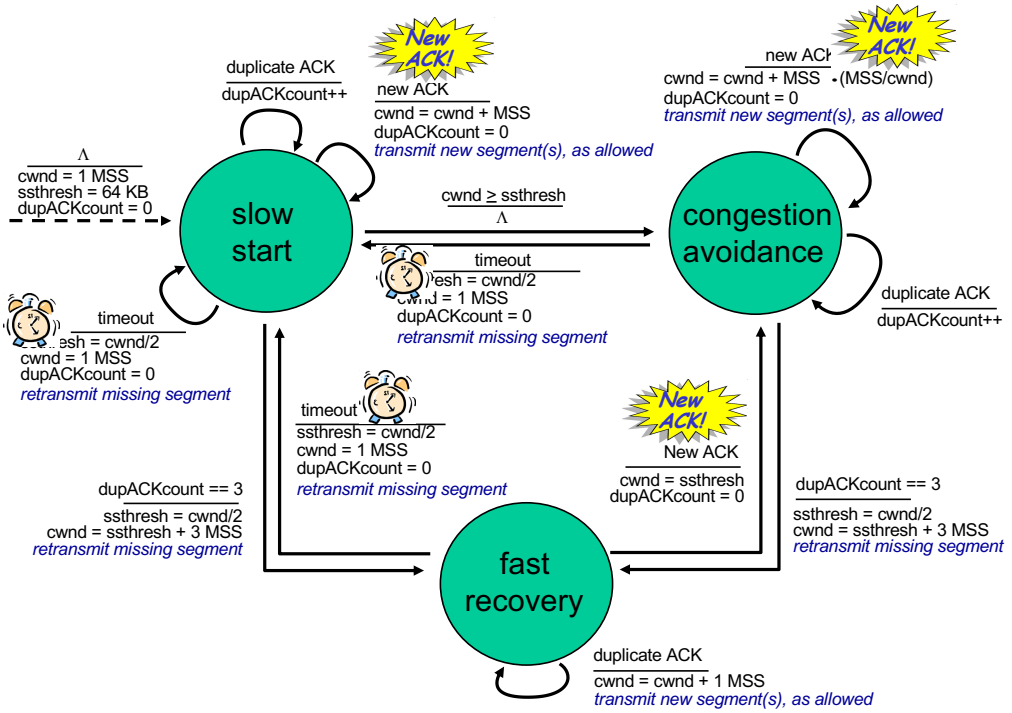
- ACKs: increase **cwnd** by 1 MSS per RTT: additive increase
- loss: cut **cwnd** in half (non-timeout-detected loss): multiplicative decrease
- AIMD: Additive Increase Multiplicative Decrease

# TCP congestion control FSM: Overview

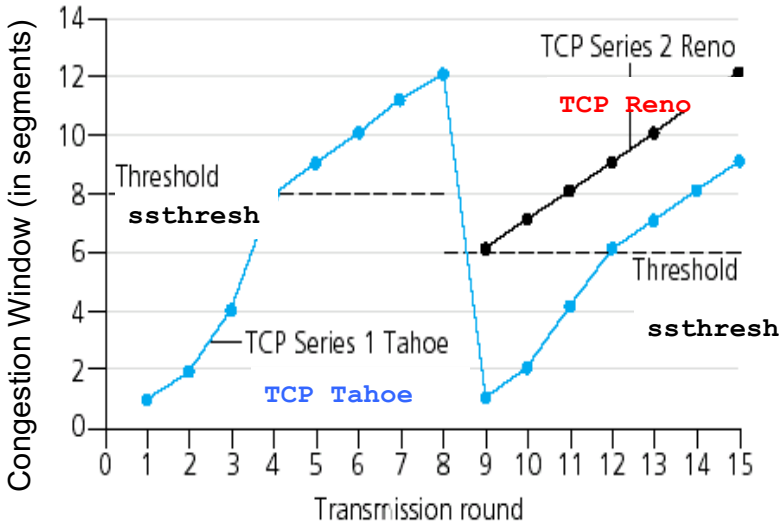





# TCP congestion control FSM: details



# Popular Flavors of TCP





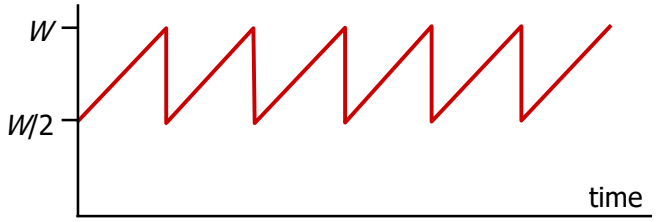

# Summary: TCP Reno Congestion Control

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

## TCP throughput

- What's the average throughout of TCP as a function of congestion window size and RTT?
  - Ignore slow start
- Let *W* be the congestion window size when loss occurs.
- When window is *W*, throughput is *W*/*RTT*
- Just after loss, window drops to *W*/2, throughput to *W*/2*RTT*.
- Assumption: *RTT* and *W* are approximately constant over the duration of the connection.
- Average throughout: 0.75 *W*/*RTT*






## TCP Future

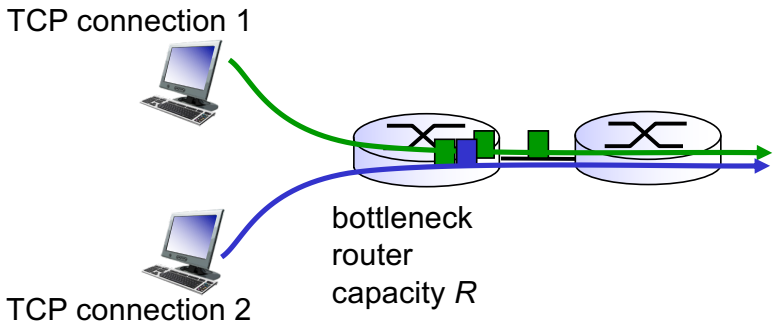
- Example: 1500-byte segments, 100 ms RTT, want 10 Gbps throughput
- Requires average window size = 83,333 in-flight segments
- Throughput in terms of loss rate *L*:

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

- ➜ $L = 2 \times 10^{-10}$ (i.e., one loss event every $5 \times 10^{9}$ segments)
- New versions of TCP for high-speed are needed!

# TCP Fairness

Fairness goal: if *K* TCP sessions share same bottleneck link with transmission rate of *R* bits/sec, each should have average rate of *R*/*K*

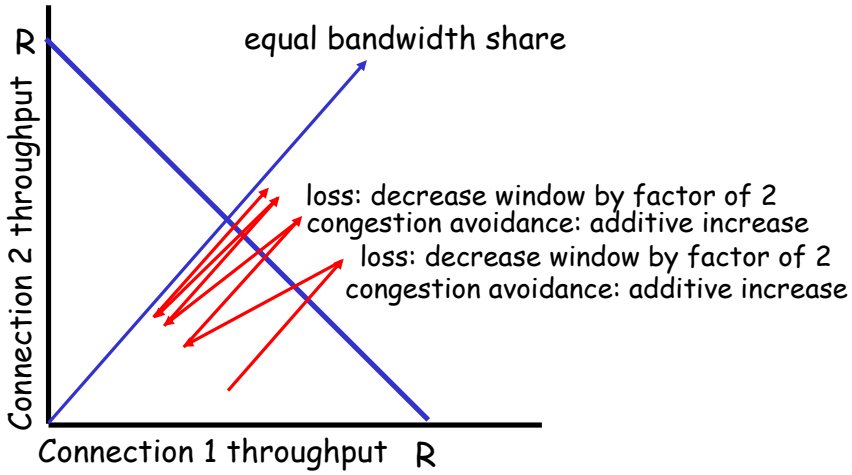





# Why is TCP fair?

Two competing sessions sharing a single link with rate R:

- Two connections have the same MSS and RTT.
- Ignore slow start. Connections operate in congestion avoidance.
- "Additive increase" gives slope of 1, as throughout increases.
- "Multiplicative decrease" decreases throughput proportionally.



# Fairness (more)

## Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

## Fairness and parallel TCP connections

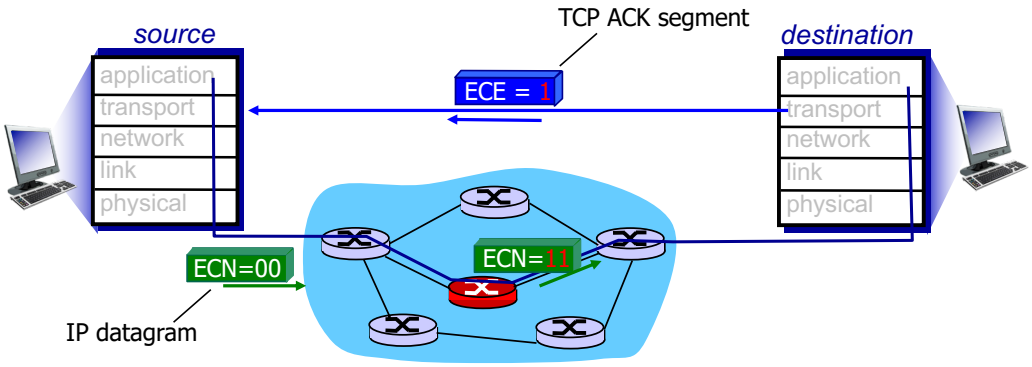
- nothing prevents app from opening parallel cnctions between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 cnctions;
  - new app asks for 1 TCP, gets rate *R*/10
  - new app asks for 11 TCPs, gets *R*/2



# Explicit Congestion Notification (ECN)

network-assisted congestion control:

- two bits in IP header (ToS field) marked by network router to indicate congestion
- congestion indication carried to receiving host
- receiver (seeing congestion indication in IP datagram) sets ECE (Explicit Congestion Notification Echo) bit on receiver-to-sender ACK segment to notify sender of congestion



# 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"
- two network layer chapters:
  - data plane
  - control plane