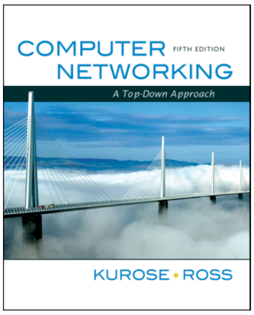
Introduction to Computer Networking

Guy Leduc

Chapter 3 Transport Layer



Computer Networking: A Top Down Approach, 5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

Chapter 3: Transport Layer

Our goals:

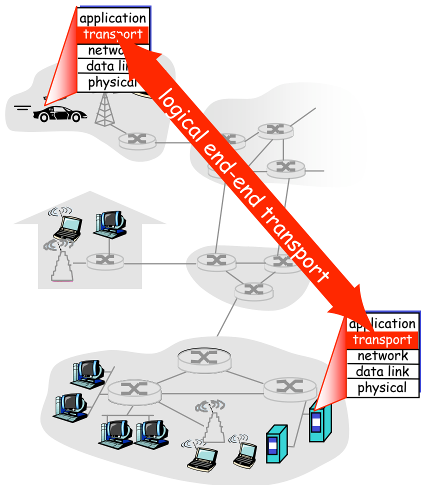
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion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

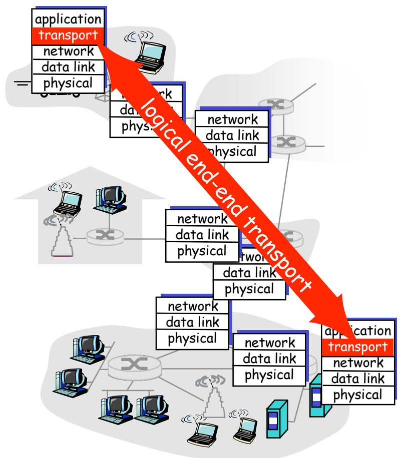
(see example in reference book)

12 kids sending letters to 12 kids

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

Internet transport-layer protocols

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



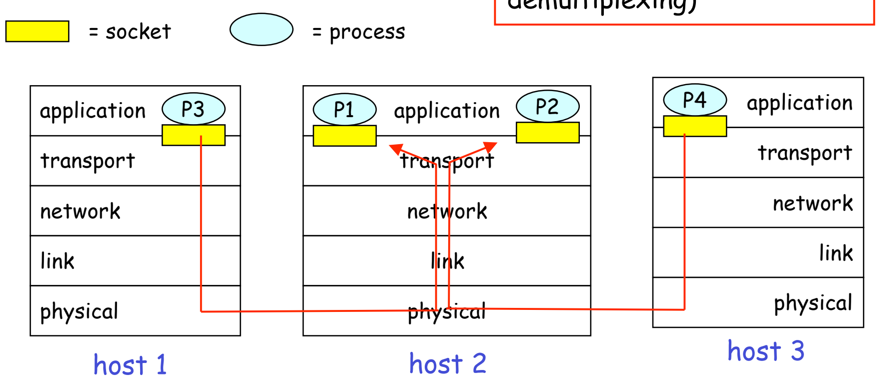
Chapter 3 outline

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

Multiplexing/demultiplexing

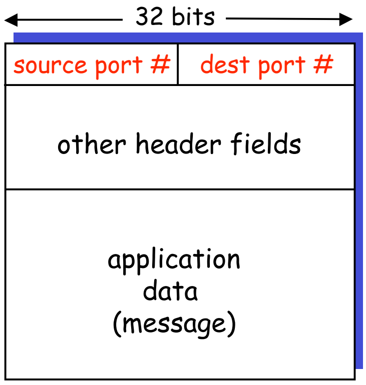
Demultiplexing at rcv host:
delivering received segments to correct socket

Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



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:

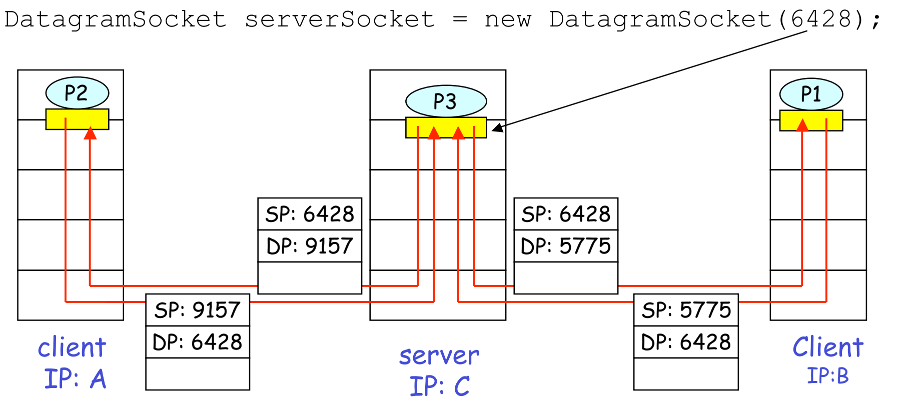
```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

- 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 (+ source IP) to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

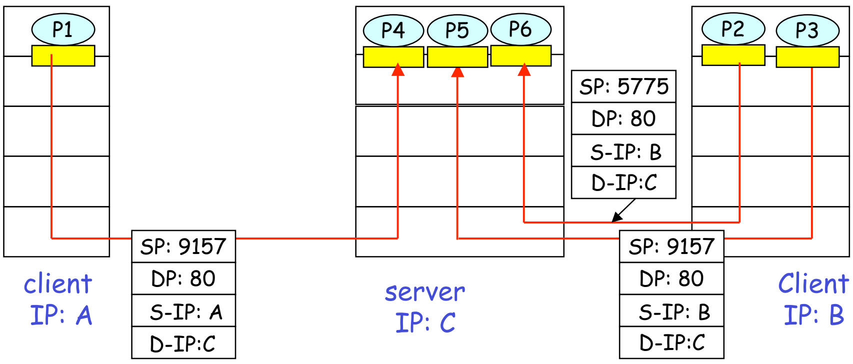


SP (together with Source IP) provides "return address" to app process

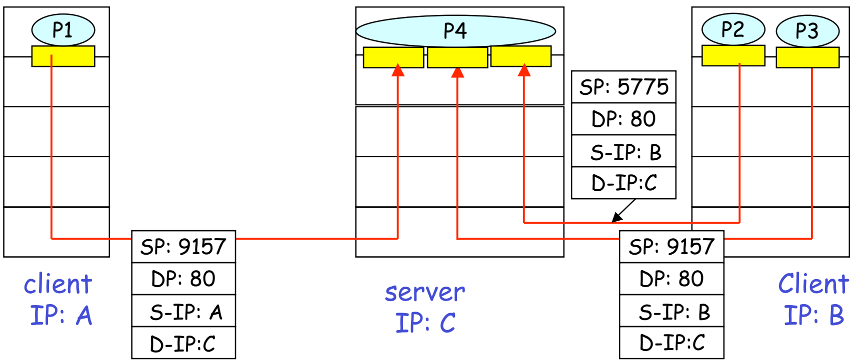
Connection-oriented demux

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

Connection-oriented demux (cont)



Connection-oriented demux: Threaded Web Server



Chapter 3 outline

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

UDP: User Datagram Protocol [RFC 768]

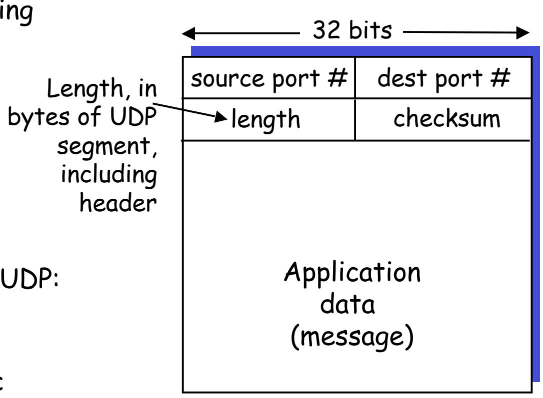
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

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

UDP: more

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



UDP segment format

UDP checksum

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

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

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

Internet Checksum Example

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

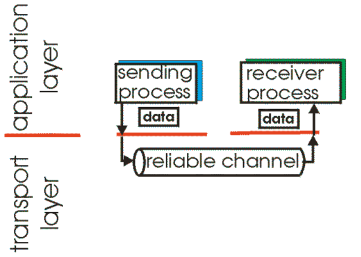
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	1	0	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

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!

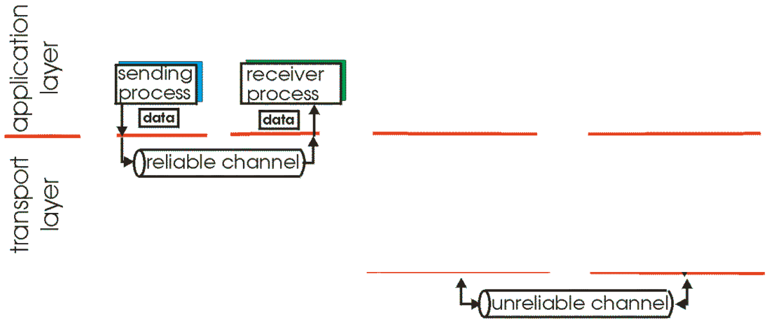


(a) provided service

- 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!

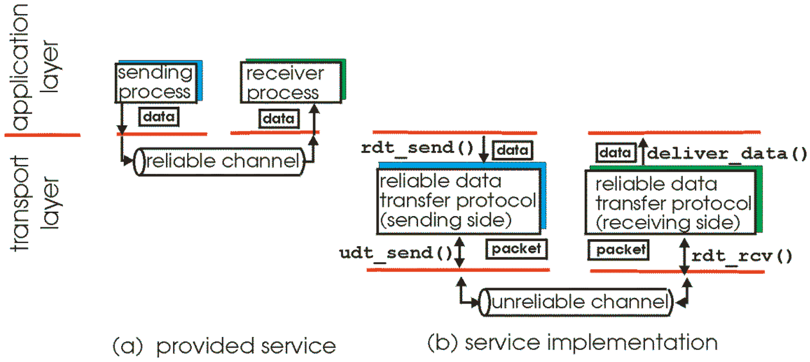


(a) provided service (b) service implementation

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

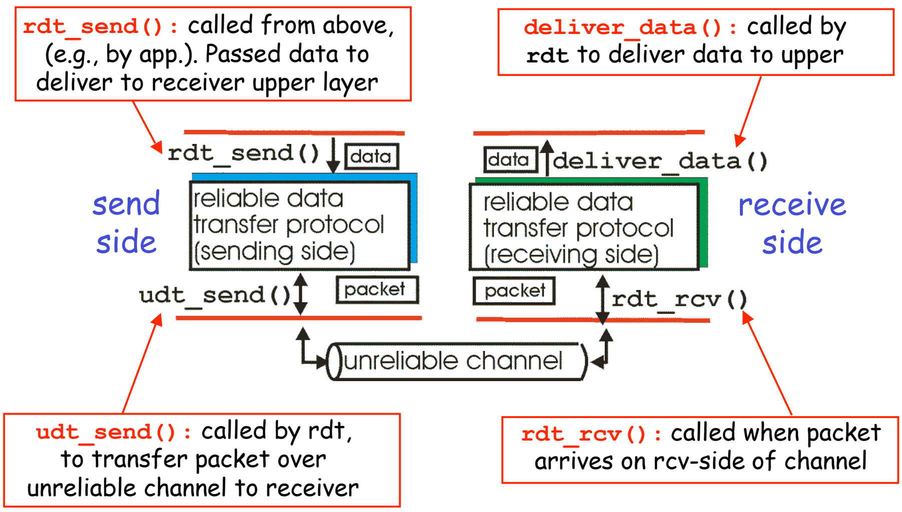
Principles of Reliable data transfer

- 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)

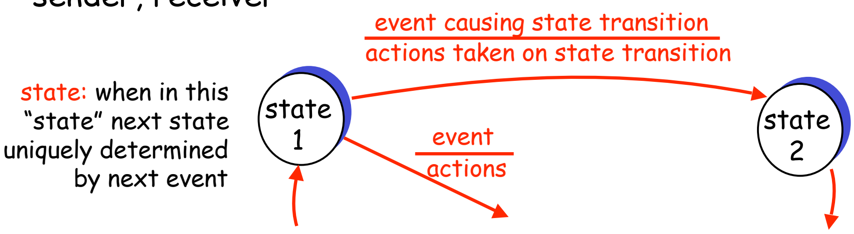
Reliable data transfer: getting started



Reliable data transfer: getting started

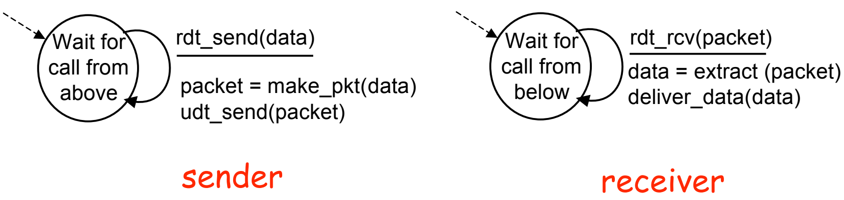
We'll:

- 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



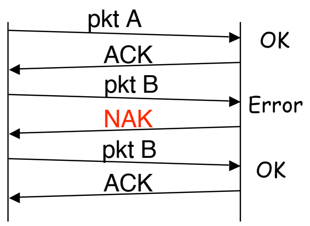
Rdt1.0: reliable transfer over a reliable channel

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

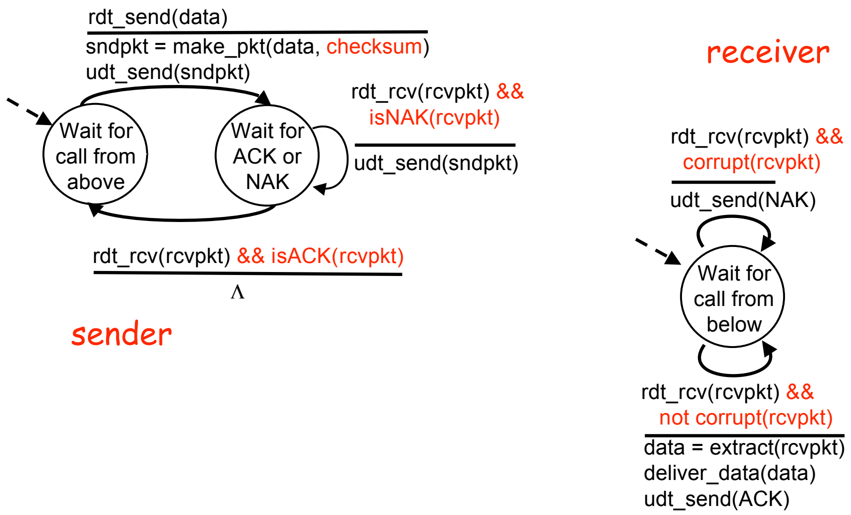


Rdt2.0: channel with bit errors

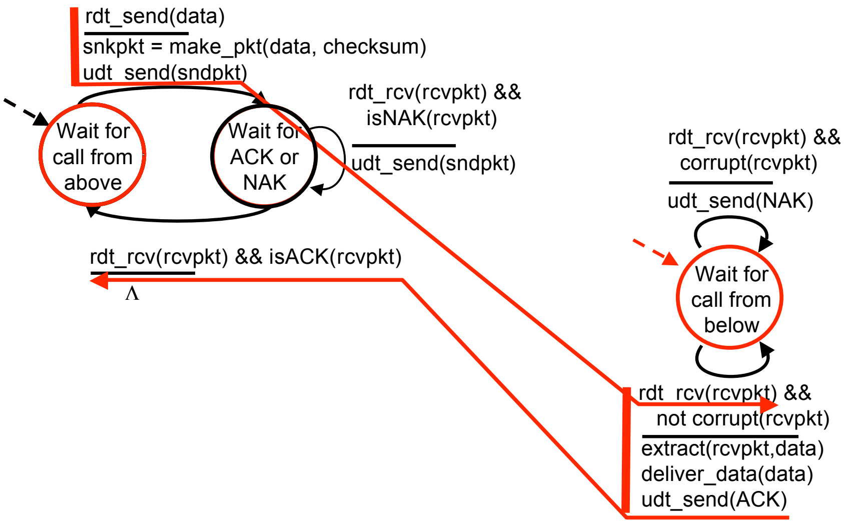
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - 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



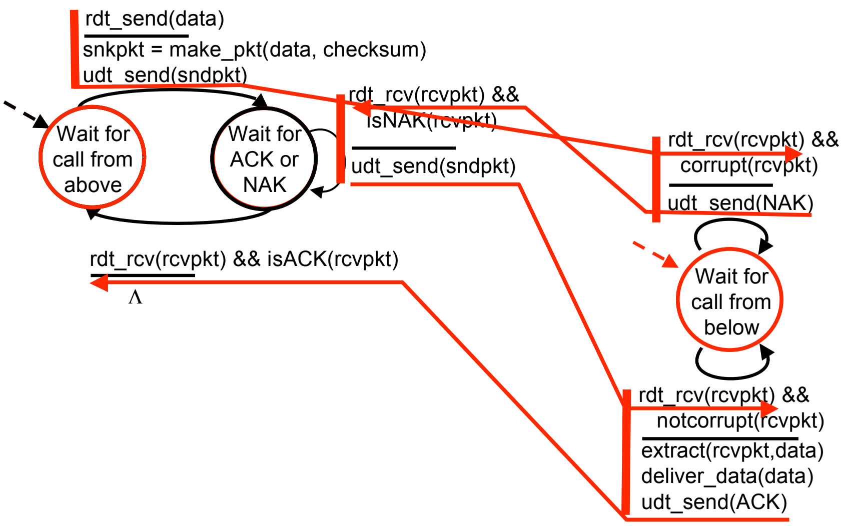
rdt2.0: FSM specification



rdt2.0: operation with no errors



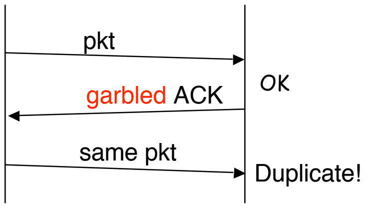
rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- garbled ACK/NAK detected by checksum too
- garbled ACK/NAK discarded
- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

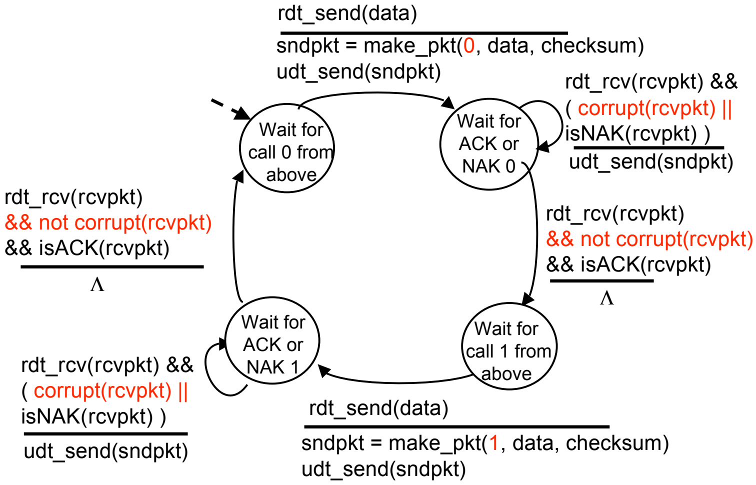


Handling duplicates:

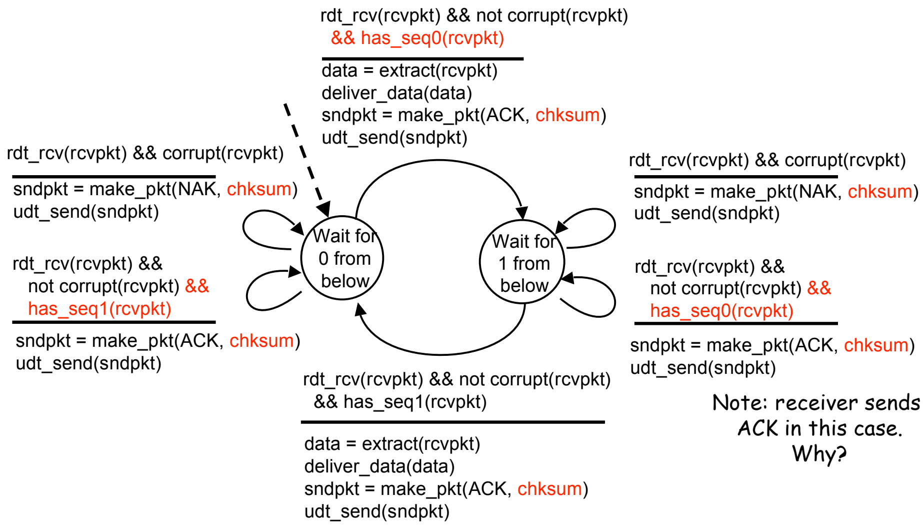
- sender retransmits current pkt if ACK/NAK garbled
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

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. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

Receiver:
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #

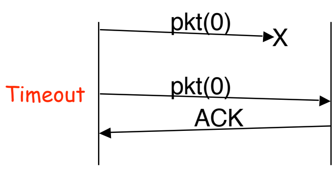
rdt3.0: channels with errors *and* loss

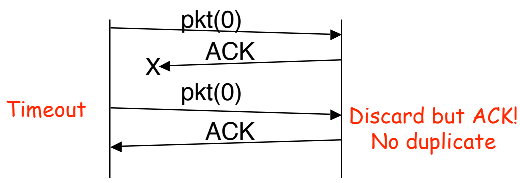
New assumption: underlying channel can also lose packets (data or ACKs)

- 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
- requires countdown timer

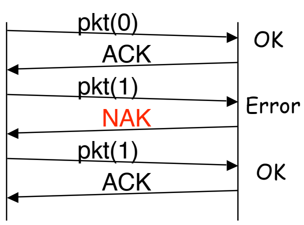




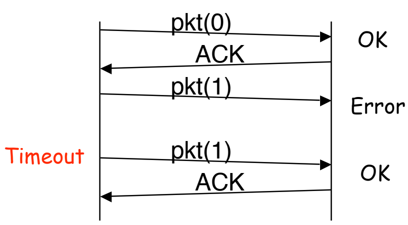
Rdt3.0: actually no need for NAKs!

- Up to now:
  - Timer for packet loss
  - NAK for packet errors
- Simpler:
  - Timer for both packet loss and errors!
- NAK would improve recovery time, but it's not our concern here!

With ACKs & NAKs

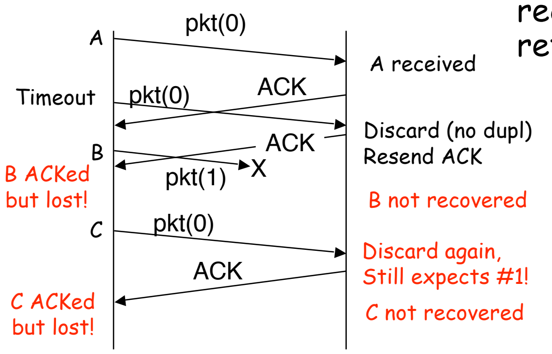


With ACKs only



Flaw: Delayed packets or ACKs

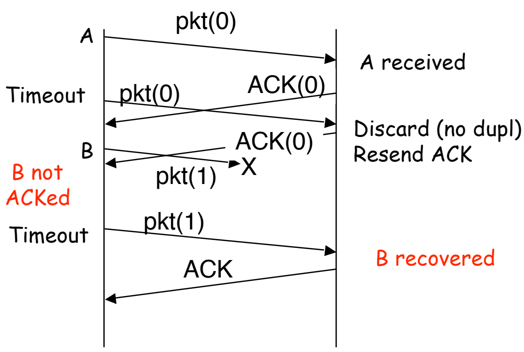
- If pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
- However, race conditions are possible between the received ACK and the retransmitted packet!



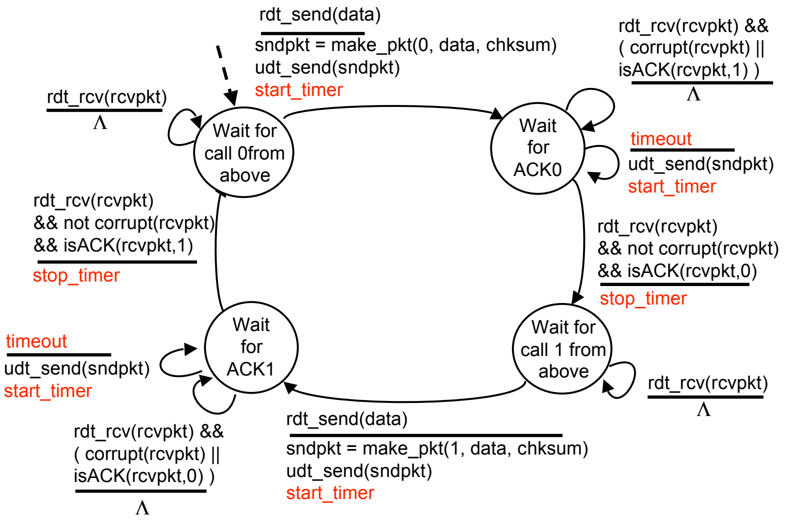
Note: such race conditions are only possible over full-duplex channels

Fixing rdt3.0

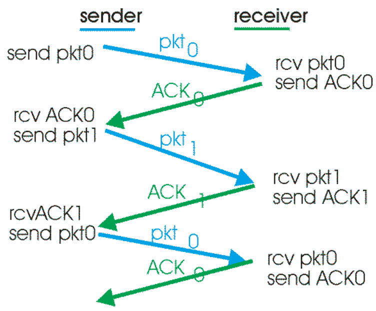
- receiver must specify seq # of pkt being ACKed



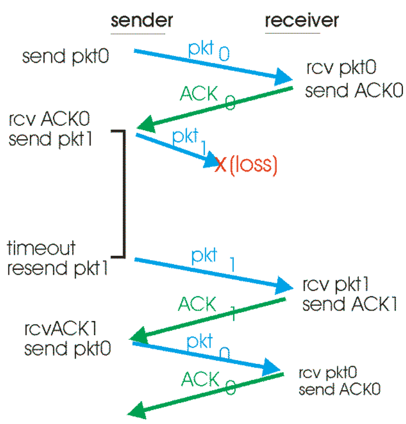
rdt3.0 sender



rdt3.0 = Alternating-bit protocol (1969)

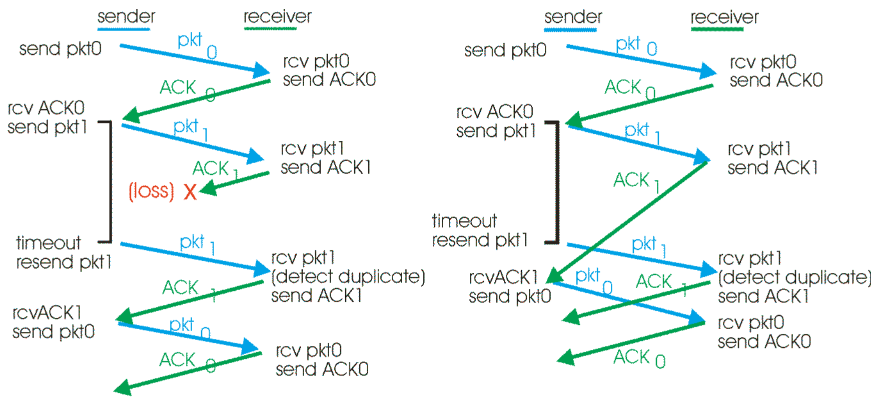


(a) operation with no loss



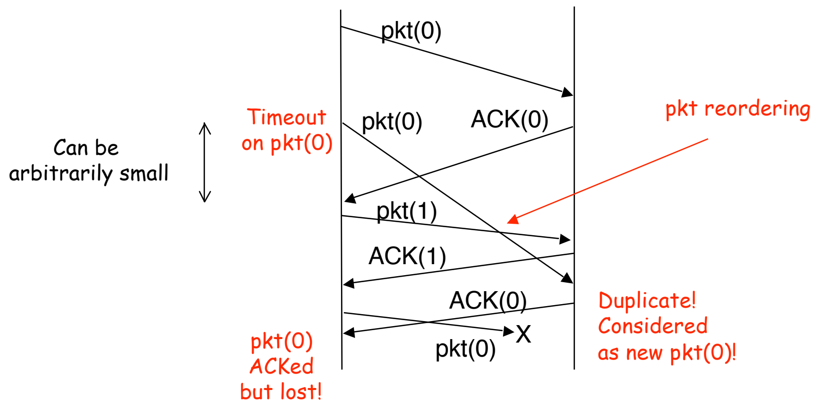
(b) lost packet

rdt3.0 in action (2)



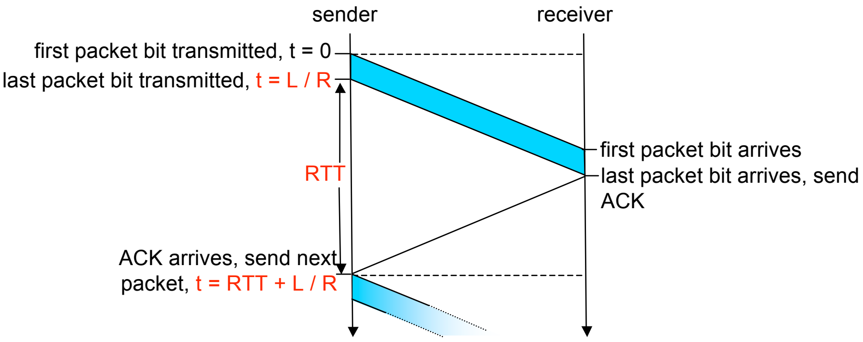
(c) lost ACK   (d) premature timeout

rdt3.0 still incorrect if pkt or ACK reordering is possible



Solution: Choose timeout so large that when a pkt is retransmitted the sender is sure that the previous copy of this pkt and its ACK have disappeared from the network.
Better solution: Use a much larger seq# space (see later).

Performance of rdt3.0: stop-and-wait operation



U_{sender}: utilization = fraction of time sender busy sending

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{1}{1 + \frac{R \cdot RTT}{L}}$$

R · RTT/2 = Bandwidth-delay product
= "in-flight" bits

Performance of rdt3.0

- Was OK over local low-speed networks, but…
- Example: 1 Gbps link, 15 ms end-to-end prop. delay, 1KB packet:

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

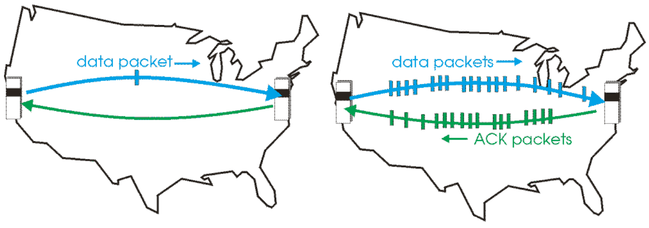
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

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

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

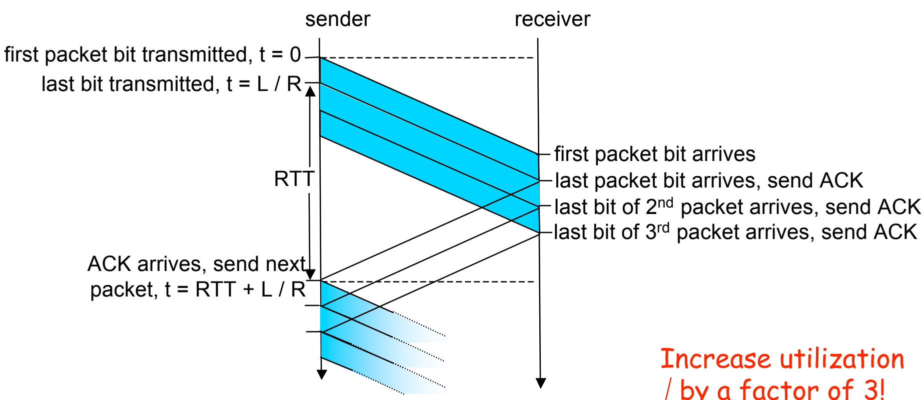


(a) a stop-and-wait protocol in operation

(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!

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

Pipelining Protocols

Go-back-N: overview

- *sender:* up to N unACKed pkts in pipeline
- *receiver:* only sends cumulative ACKs
  - doesn't ACK pkt if there's a gap
- *sender:* has timer for oldest unACKed pkt
  - if timer expires: retransmit all unACKed packets

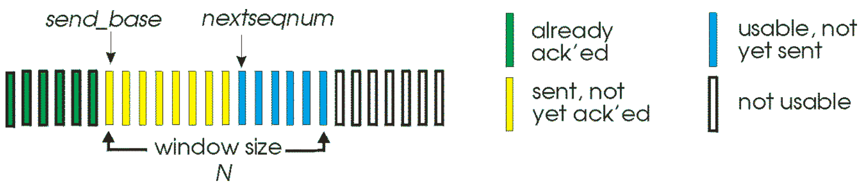
Selective Repeat: overview

- *sender:* up to N unACKed packets in pipeline
- *receiver:* ACKs individual pkts
- *sender:* maintains timer for each unACKed pkt
  - if timer expires: retransmit only unACKed packet

Go-Back-N (GBN)

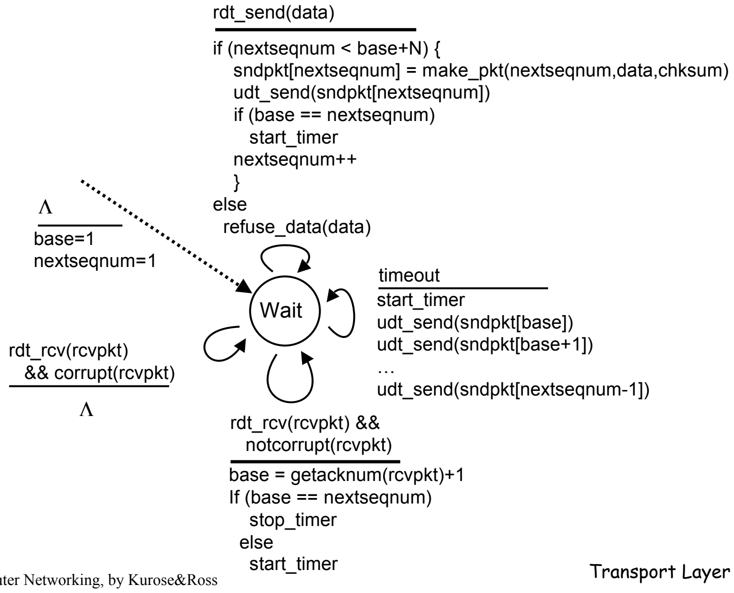
Sender:

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

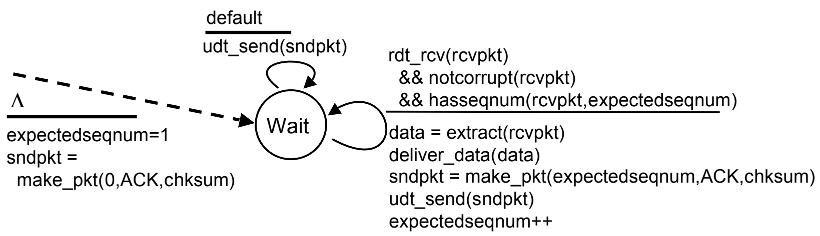


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- single timer: conceptually on the oldest transmitted and unack'ed pkt
- *timeout:* retransmit all pkts in window

GBN: sender extended FSM



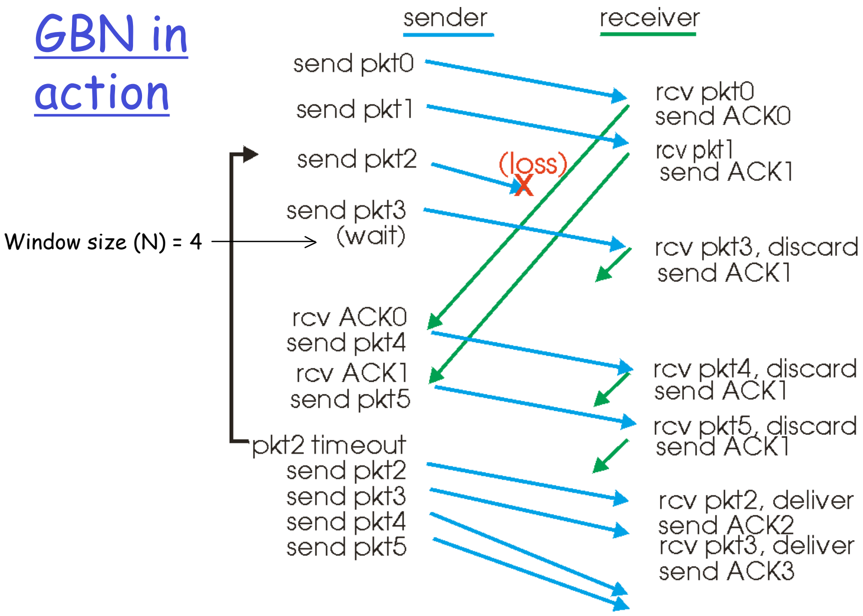
GBN: receiver extended FSM



ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember **expectedseqnum**

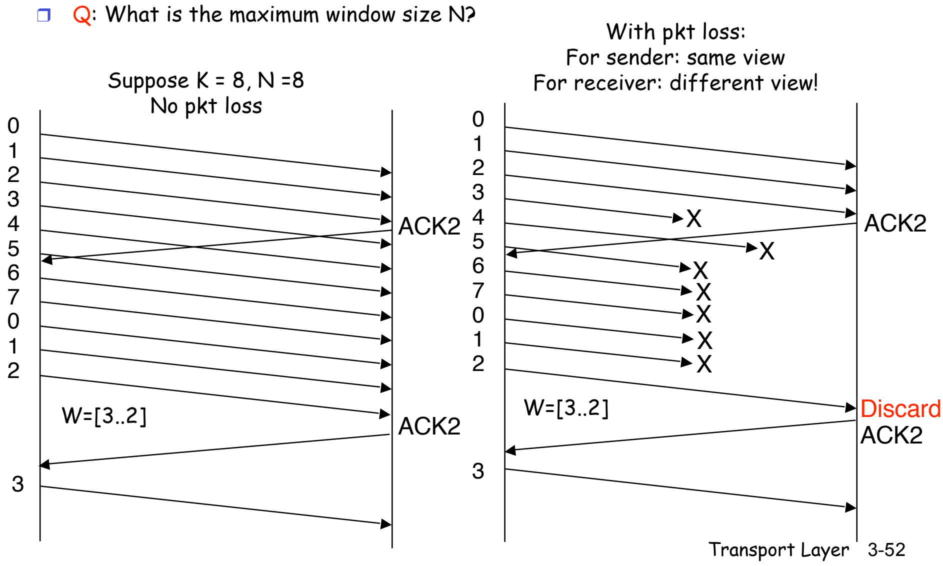
- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #

GBN in action



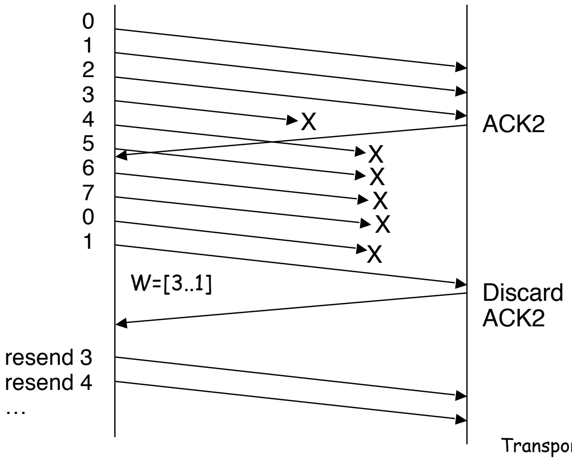
Maximum window size with GBN

- If seq# size = K ("The modulo")
- Q: What is the maximum window size N?



GBN: maximum window size

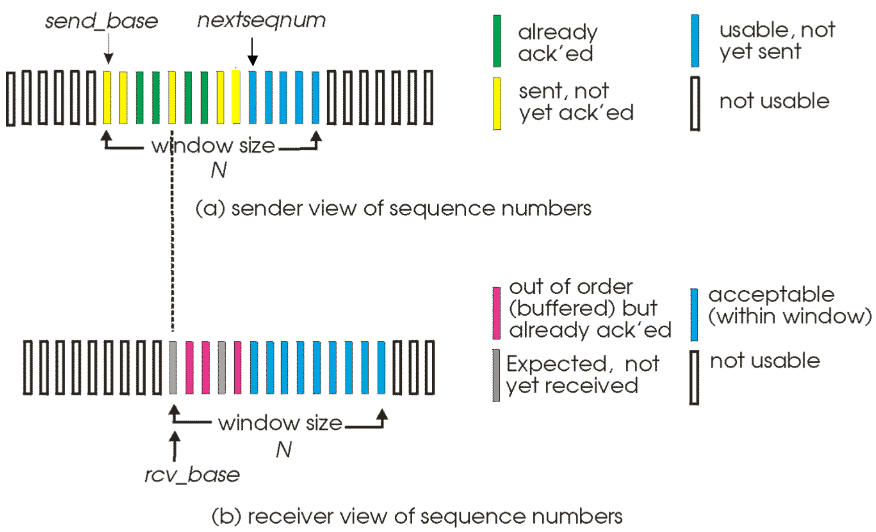
- Constraint:
  - window size (N) ≤ seq# size (K) - 1



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

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

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

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

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

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

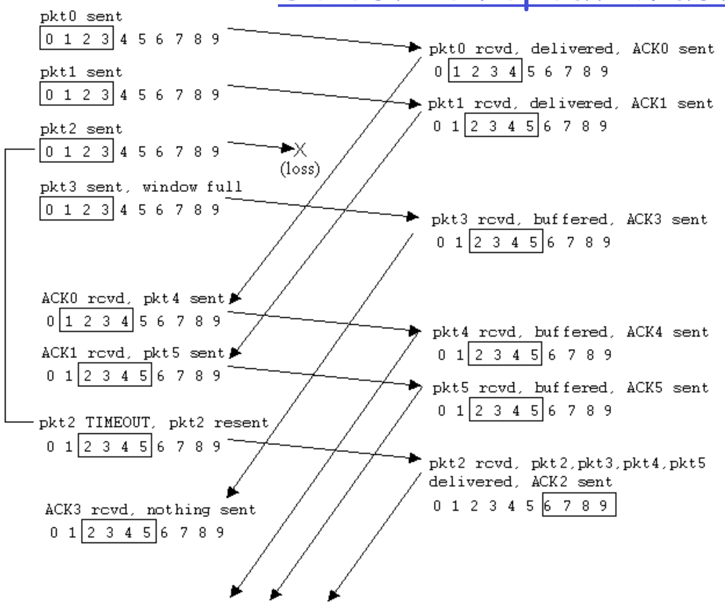
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action

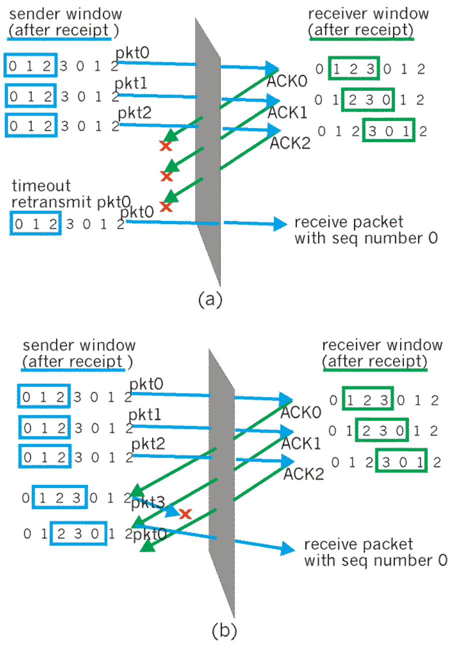


Selective repeat: dilemma

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)

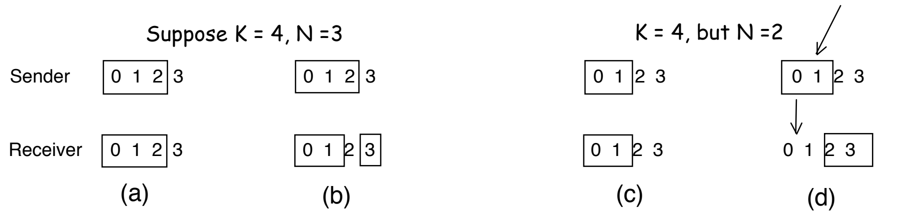
Q: what relationship between seq # size and window size?



Maximum window size with selective repeat

- If seq# size = K ("The modulo")
- Q: What is the maximum window size N?
- A: N ≤ K/2

Retransmission of received pkts fall outside rec window



(a) Initial situation with a window of size N=3.
(b) After 3 frames have been sent and received but not ACK'ed.
The upper side of the rec window falls in the sending window
(c) Initial situation with a window of size N=2.
(d) After 2 frames sent and received but not ACK'ed.
The upper side of the rec window does not fall in the sending window

Maximum window size: general principle

- Notations:
  - seq# space = [0..K-1], i.e. modulo K
  - Maximum sender window size = Ns
  - Maximum receiver window size = Nr
- Requirement:
  - Ns + Nr ≤ K
- Particular cases:
  - GBN: Nr = 1, Ns ≤ K - 1
  - Selective repeat: Ns = Nr ≤ K/2
  - Alternating-bit: K = 2, Ns = Nr = 1
    - Special case of both GBN and Selective Repeat

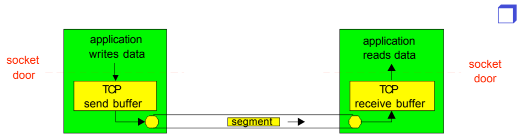
Chapter 3 outline

- 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

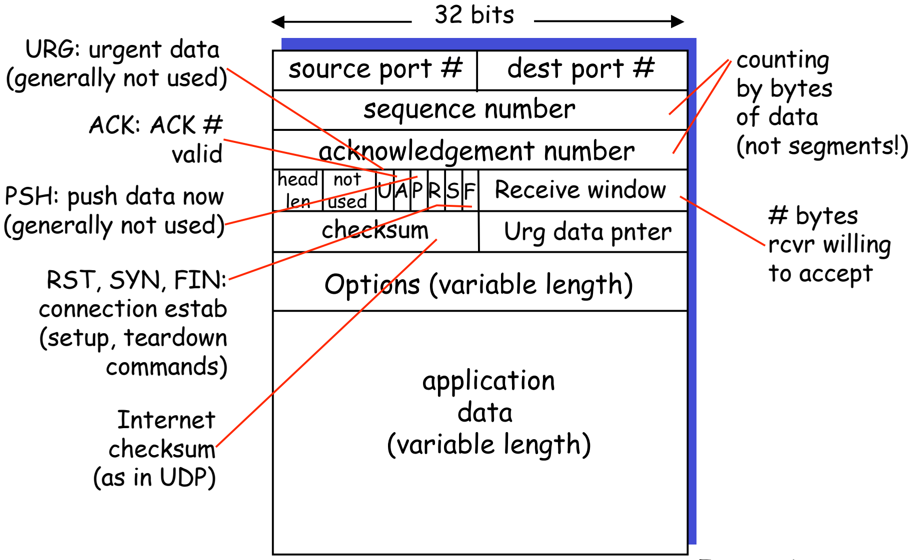
RFCs: 793, 1122, 1323, 2018, 2581

- 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:

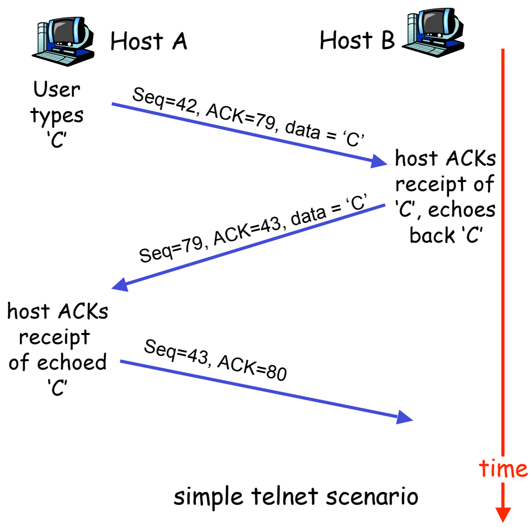
- byte stream "number" of first byte in segment's data

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 implementer

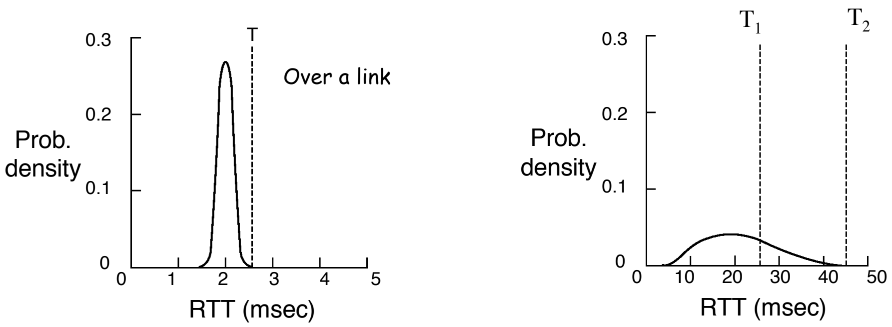


simple telnet scenario

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

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



TCP Round Trip Time and Timeout

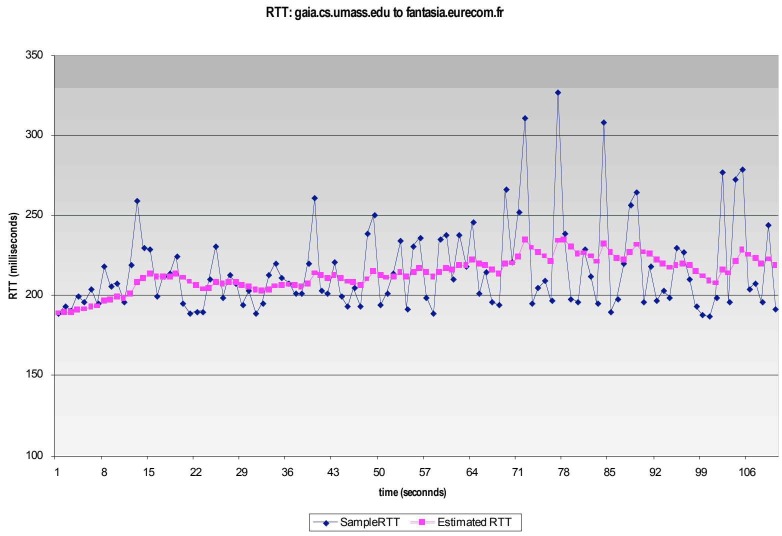
Q: how to estimate RTT?

- `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`

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

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

Example RTT estimation:



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:

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

(typically, β = 0.25)

Then set timeout interval:

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

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

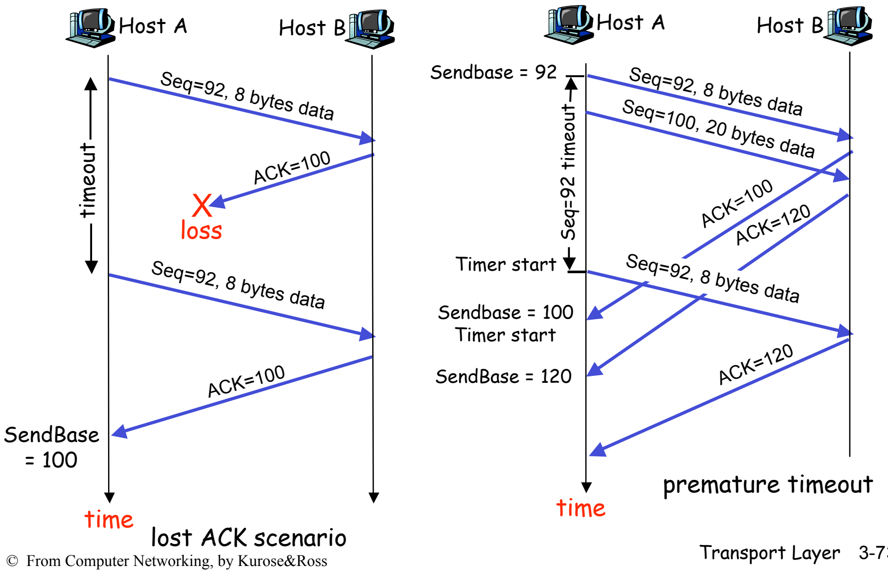

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

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

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

TCP sender (simplified)

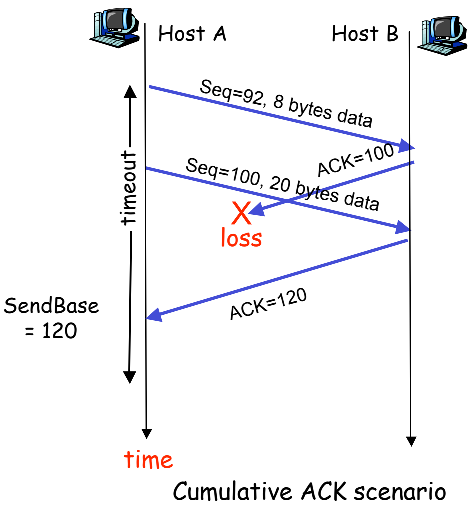
Comment:
• SendBase-1: last cumulatively ack'ed byte

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

TCP: retransmission scenarios



TCP retransmission scenarios (more)

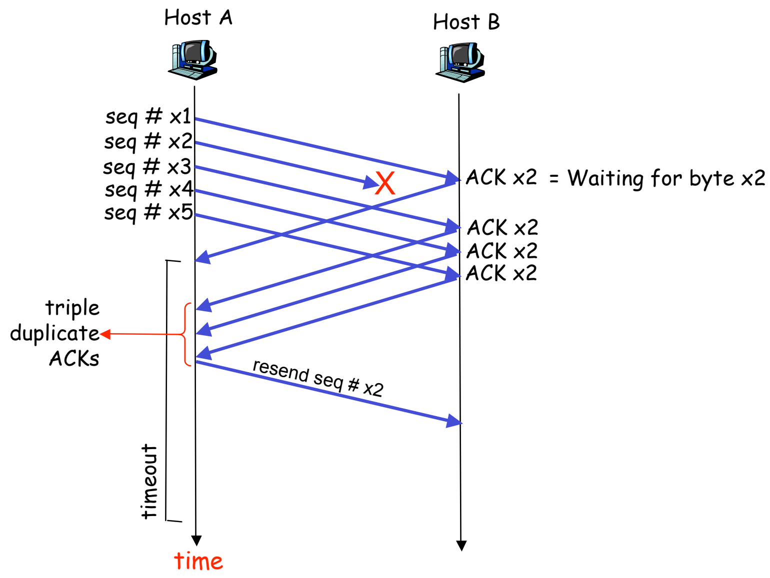


TCP ACK generation [RFC 1122, RFC 2581]

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-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

Fast Retransmit

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs:
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs for that segment.
- 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) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
            }
          else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                 }
```

a duplicate ACK for already ACKed segment

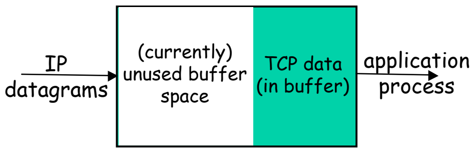
fast retransmit

Chapter 3 outline

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

TCP Flow Control

- receive side of TCP connection has a receive buffer:



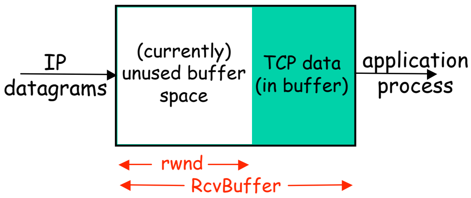
- app process may be slow at reading from buffer

flow control

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

- *speed-matching service:* matching send rate to receiving application's drain rate

TCP Flow control: how it works



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

- unused buffer space:

= rwnd

= RcvBuffer-[LastByteRcvd - LastByteRead]

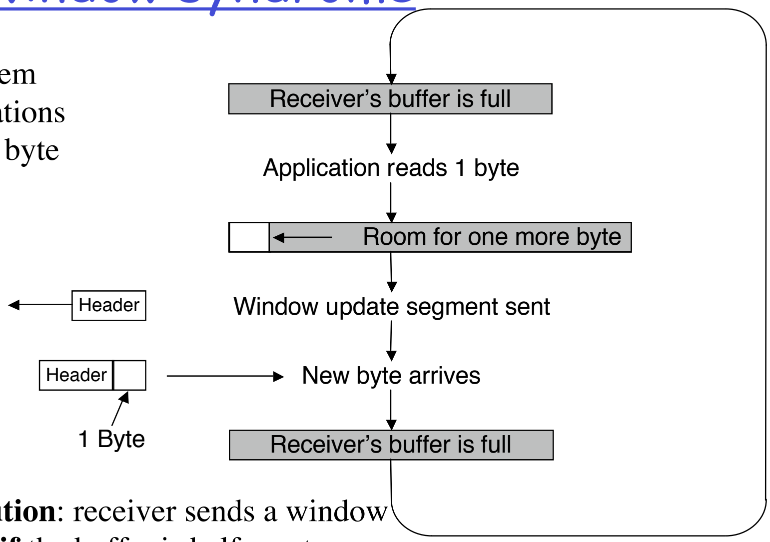
- receiver: advertises unused buffer space by including `rwnd` value in segment header
- sender: limits # of unACKed bytes to `rwnd`
  - guarantees receiver's buffer doesn't overflow

Nagle algorithm

- When data come in from socket one byte at the time
  - send first byte and buffer all the rest until the outstanding byte is acknowledged
  - sends other segments as one per RTT
- Useful for Telnet
  - Otherwise:
  - 41 bytes segments containing 1 byte of data
  - 40 bytes of TCP/IP header overhead

Silly window syndrome

Still a problem with applications reading one byte at a time



Clark's solution: receiver sends a window update **only if** the buffer is half empty, **or if** a full segment can be received

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

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

- *server:* contacted by client

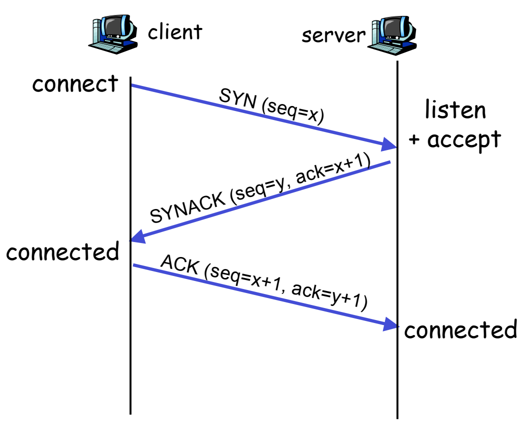
```
Socket connectionSocket = welcomeSocket.accept();
```


Three-way handshake

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data
- starts timer for SYN retransmission

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



If server port not open, TCP server sends back a RST segment

TCP Connection Management (cont.)

Closing a connection:

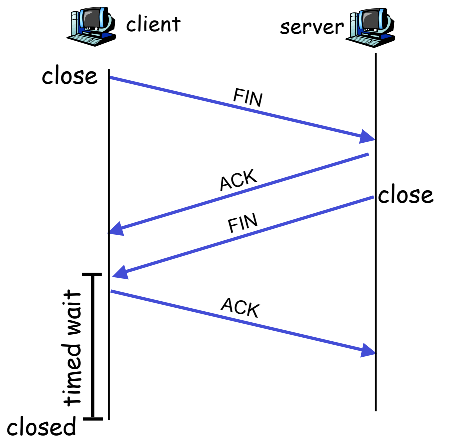
client closes socket: `clientSocket.close();`

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

Step 2: server receives FIN, replies with ACK. When all data sent, closes connection, sends FIN

Note: two directions closed separately

Symmetric/graceful release



TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

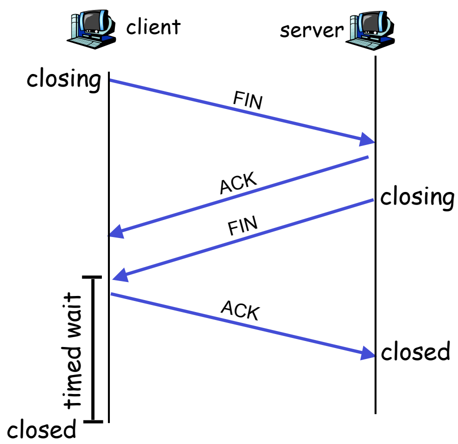
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

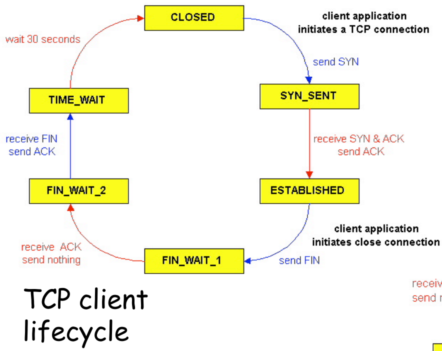
FINs have seq#, like data segments

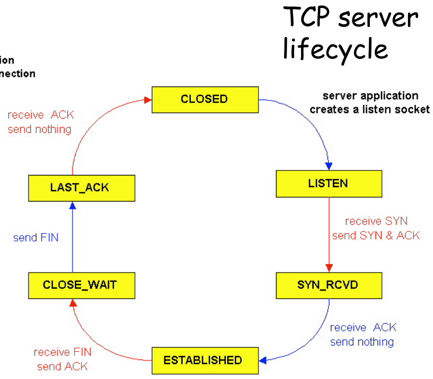
Can recover loss of last data bytes

Note: with small modification, can handle simultaneous FINs.

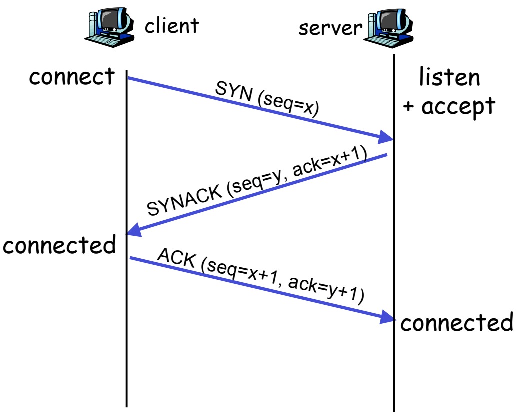


TCP Connection Management (cont)





Choosing initial seq#



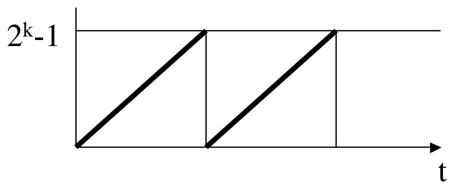
- Consider choosing x
- Constraint: x not used recently in a former connection by this client (i.e. same port#) to send TCP segments to this server (same port#)
- Recently = less than packet lifetime
- Otherwise a still alive duplicate segment from the former connection can be confused with this SYN

Choosing initial seq# (2)

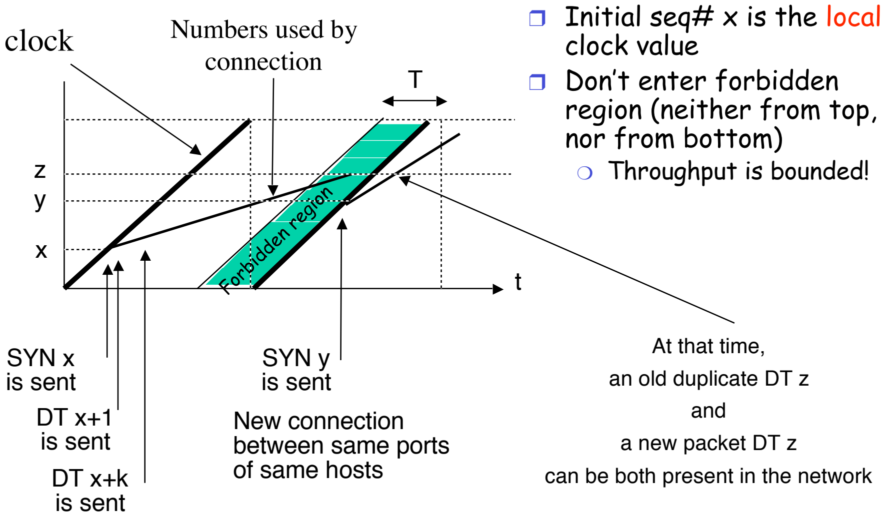
1. In practice
   1. If client can only reuse the same port# when a maximum packet lifetime has elapsed after the last packet has been sent, and
   2. If, when a host fails (= unknown duration), it waits at least a maximum packet lifetime before opening the first connection,
   3. Then, any initial seq# x and y are OK, they could be randomly chosen
2. In principle, could avoid these delays by using clocks (see next slide)

Using clocks

- Let T be the maximum packet lifetime in the network
- Each host has a "clock"
  - that never stops running, even when the station is down
  - that need not be synchronized with other clocks
- The clock is merely a regular and reliable counter of say k bits such that 2^k ticks > T



Using clocks (2)

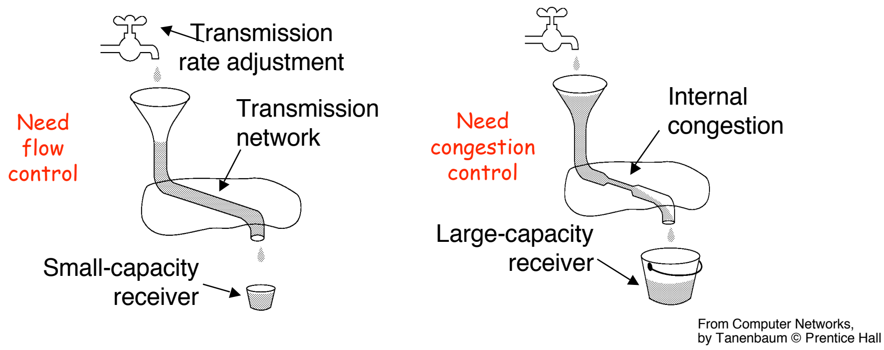


- Initial seq# x is the local clock value
- Don't enter forbidden region (neither from top, nor from bottom)
  - Throughput is bounded!

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

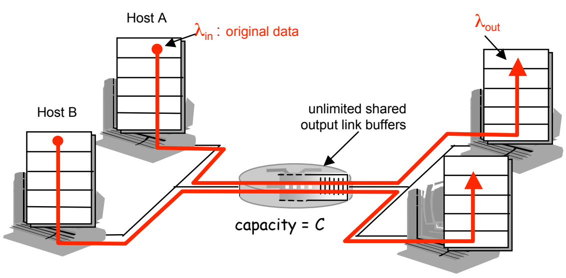
Principles of Congestion Control

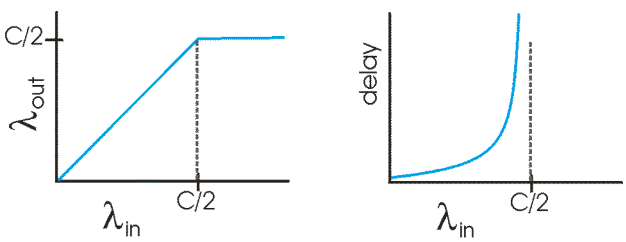


- informally: "too many sources sending too much data too fast for *network* to handle"
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queuing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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

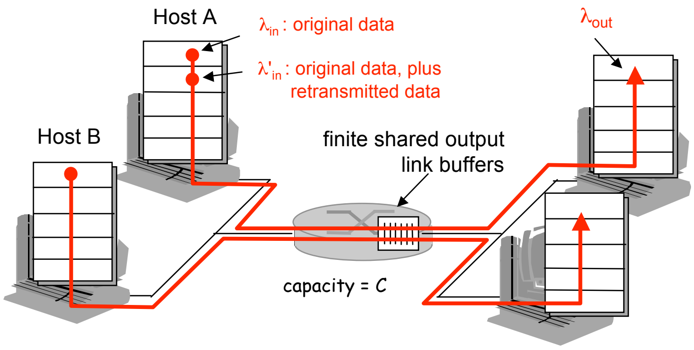




- large delays when congested
- maximum achievable throughput

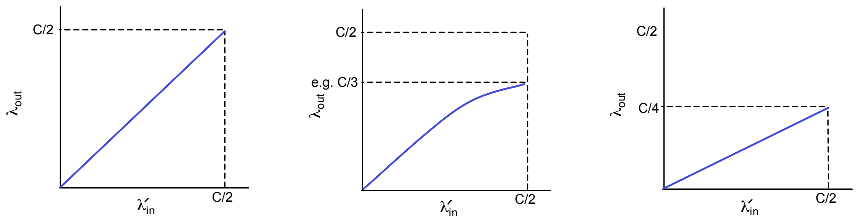
Causes/costs of congestion: scenario 2

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



Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a) sender avoids loss

b) only lost packets retransmitted

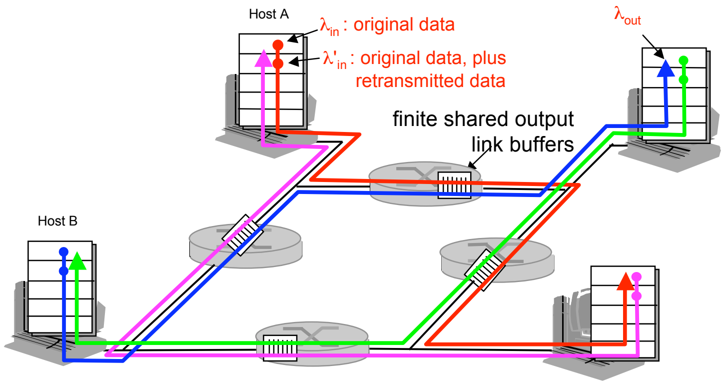
c) all packets retransmitted once

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

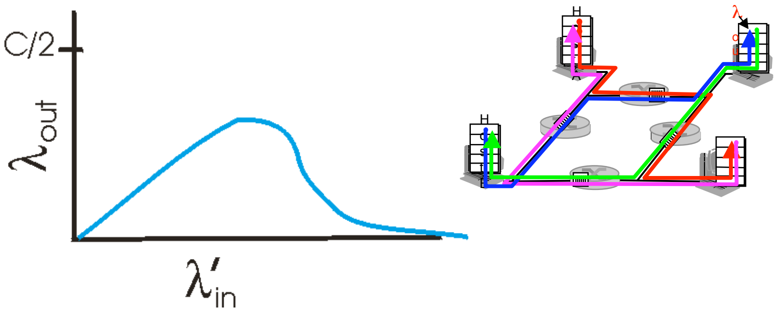
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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



Causes/costs of congestion: scenario 3



Another "cost" of congestion:

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

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:
- routers provide feedback to end-systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at (ATM ABR)
- Not covered in this course

Chapter 3 outline

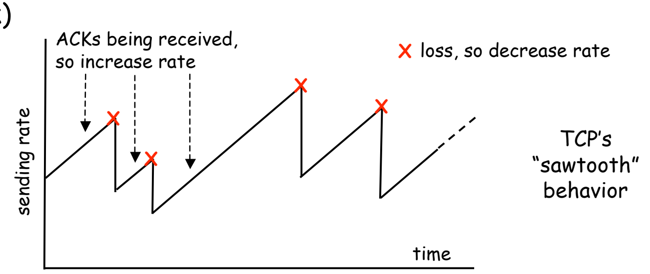
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of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

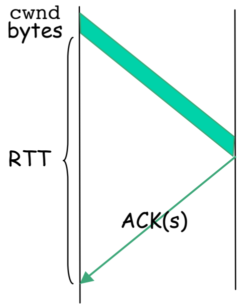
- sender limits rate by limiting number of unACKed bytes "in pipeline":

  LastByteSent-LastByteAcked ≤ cwnd

  - **cwnd**: differs from **rwnd** (how, why?)
  - sender limited by **min(cwnd,rwnd)**
- roughly,

$$\text{rate} = \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

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



TCP Congestion Control: more details

segment loss event: reducing cwnd

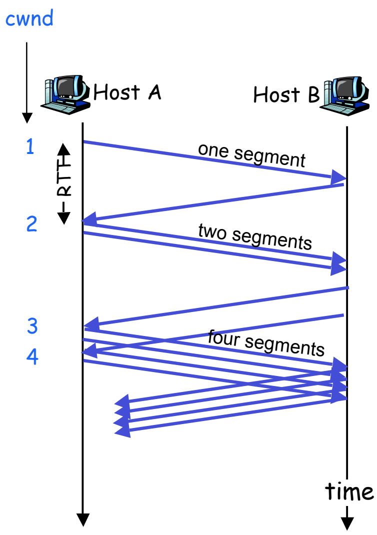
- timeout: no response from receiver
  - cut **cwnd** to 1
- 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
  - cut **cwnd** in half, less aggressively than on timeout

ACK received: increase cwnd

- slowstart phase:
  - increase exponentially fast (despite name) at connection start, or following timeout
- congestion avoidance:
  - increase linearly

TCP Slow Start

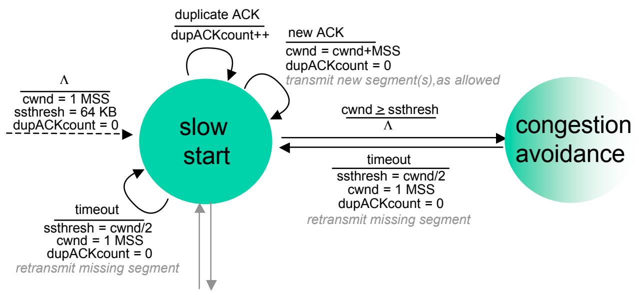
- when connection begins, **cwnd** = 1 MSS (Maximum Segment Size)
  - example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when threshold reached
  - double **cwnd** every RTT
  - done by incrementing **cwnd** by 1 for every ACK received



Transitioning into/out of slowstart

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 slowstart to congestion avoidance phase



TCP: congestion avoidance

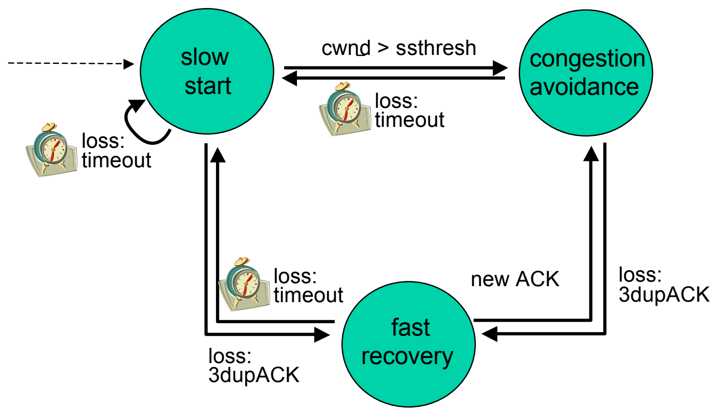
- when **cwnd > ssthresh** grow **cwnd** linearly
  - increase **cwnd** by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation: **cwnd = cwnd + MSS/cwnd** for each ACK received

AIMD

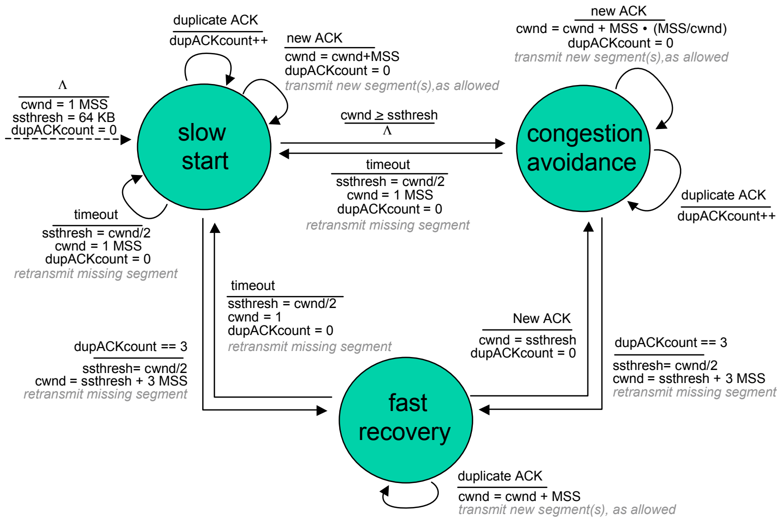
- ACKs: increase **cwnd** by 1 MSS per RTT: additive increase
- loss: cut **cwnd** in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

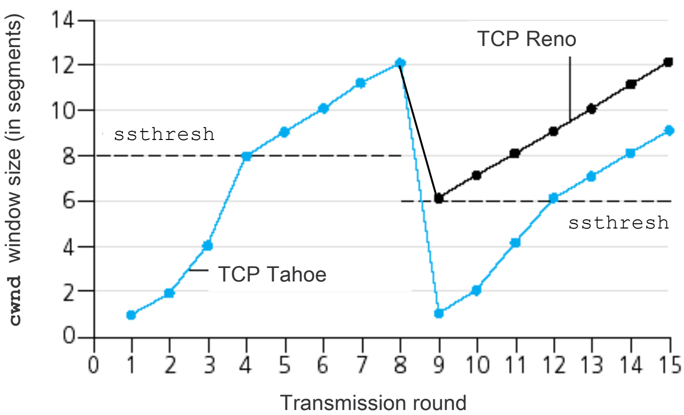
TCP congestion control FSM: overview



TCP congestion control FSM: details



Popular "flavors" of TCP

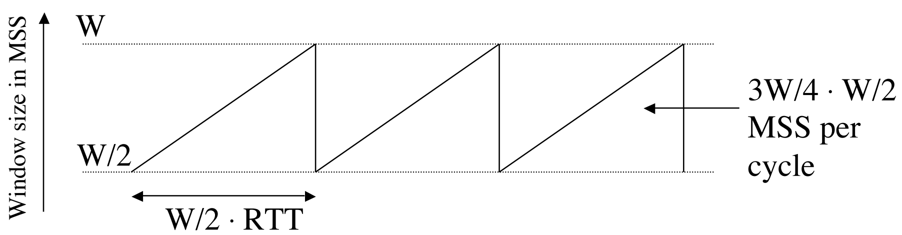


Summary: TCP Congestion Control

- when **cwnd < ssthresh**, sender in slow-start phase, window grows exponentially.
- when **cwnd >= ssthresh**, sender is in congestion-avoidance phase, window grows linearly.
- when triple duplicate ACK occurs, **ssthresh** set to **cwnd/2**, **cwnd** set to ~ **ssthresh**
- when timeout occurs, **ssthresh** set to **cwnd/2**, **cwnd** set to 1 MSS.

TCP "goodput" in steady state

- Goodput = not counting overhead and retransmitted bytes
- What's the average goodput of TCP as a function of window size and RTT?
  - Ignoring slow start
- Let W be the window size (in MSS) when losses occur
- When window is W, goodput is W/RTT
- Just after loss, window drops to W/2, goodput to 0.5 W/RTT
- Average goodput: 0.75 W/RTT



TCP goodput in steady state (2)

- Average window size (in MSS) = 3W/4
- Number of MSS per cycle = $3W/4 \cdot W/2 = 3W^2/8 = 1/p$
  - Where p is the packet loss ratio
    - One packet loss per cycle! (if p small enough)
  - So $W = \sqrt{8/3p}$
- Average goodput (in MSS/sec) = aver. window / RTT = 3W / 4RTT

$$= \sqrt{3/2}\,\frac{1}{RTT\sqrt{p}}$$

- Average goodput (in bps) $= \sqrt{3/2}\,\frac{MSS}{RTT\sqrt{p}}$

TCP Futures: TCP over "long, fat pipes"

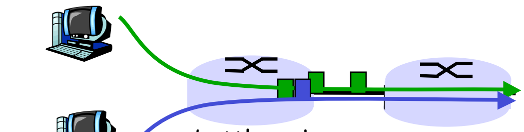
- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83 333 in-flight segments
- Throughput in terms of loss rate:

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

- Needs packet loss rate $p = 2 \cdot 10^{-10}$ *Wow!*
- New versions of TCP for high-speed needed!

TCP Fairness

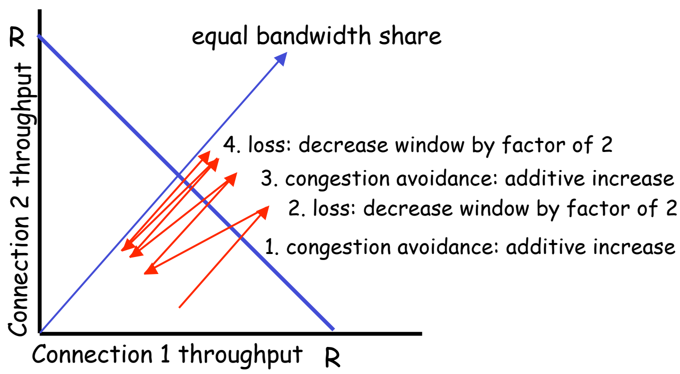
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

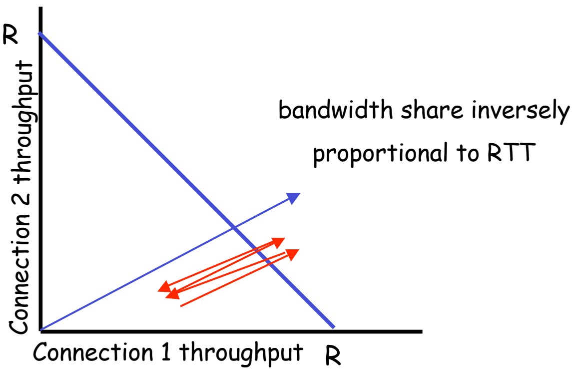
Two competing sessions having the same RTT:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



And when RTTs are different?

- If RTT of connection 2 = 2 x RTT of connection 1
- Connection 1 ramps up twice more quickly



Fairness (more)

Fairness and UDP

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

Fairness and parallel TCP connections

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