COMPUTER NETWORKS

UNIT-3

transport layer

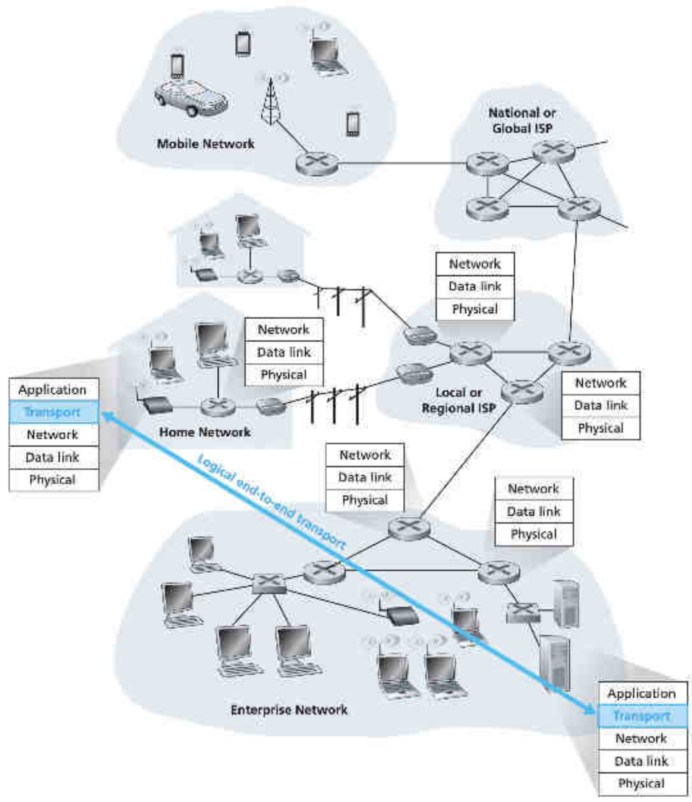
feedback/corrections: vibha@pesu.pes.edu VIBHA MASTI

TRANSPORT LAYER SERVICES

- Logical communication between app processes on different hosts

actions in end systems

- Sender: breaks app messages into segments (adding headers), passes to network layer; determines segment header field values, passes to IP
- Receiver: reassembles segments into messages, passes to application layer; receives from IP, checks header values, extracts app message, demuxes message to app layer via socket



TCP vs UDP

TCP: Transmission Control Protocol

- 3-way handshake (connection setup)
- Reliable
- In order
- Congestion control (buffer, packet loss)
- Flow control (rate, acknowledgement)

UDP: User Datagram Protocol

- Unreliable, connectionless
- Low effort
- no order

- no delay guarantee
- no bandwidth guarantee

MULTIPLEXING & DEMULTIPLEXING

- Extend host-to-host delivery to process-to-process delivery

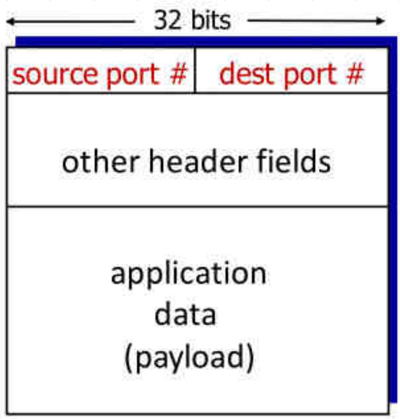
- Multiplexing: handle data from multiple sockets, add transport header and send transport segment to network layer (source port number)

- Demultiplexing: use header info to deliver received segments to correct socket (destination port number)

- Port numbers: 0 to 65535 ($2^{16}-1$) 16 bit number

- Ports 0 to 1023 are well-known port numbers, restricted / reserved and cannot be used by user / OS

• eg: HTTP: port 80, DNS: 53, SSH: 22, FTP: 21



TCP/UDP segment format

CONNECTIONLESS DEMULTIPLEXING

- UDP multiplexing / demultiplexing
- Creating socket: local port no. specified
- Creating datagram to send into UDP socket, must specify dest IP, dest port
- Receiving host: checks dest port in segment, directs UDP segment to socket with same port
- Multiple different source IPs/ ports but same dest socket → delivered to same socket (eg: http - 80) at receiving host

- UDP socket identified by a two-tuple consisting of (destination IP address, destination port number)

Example

UDPClient.py

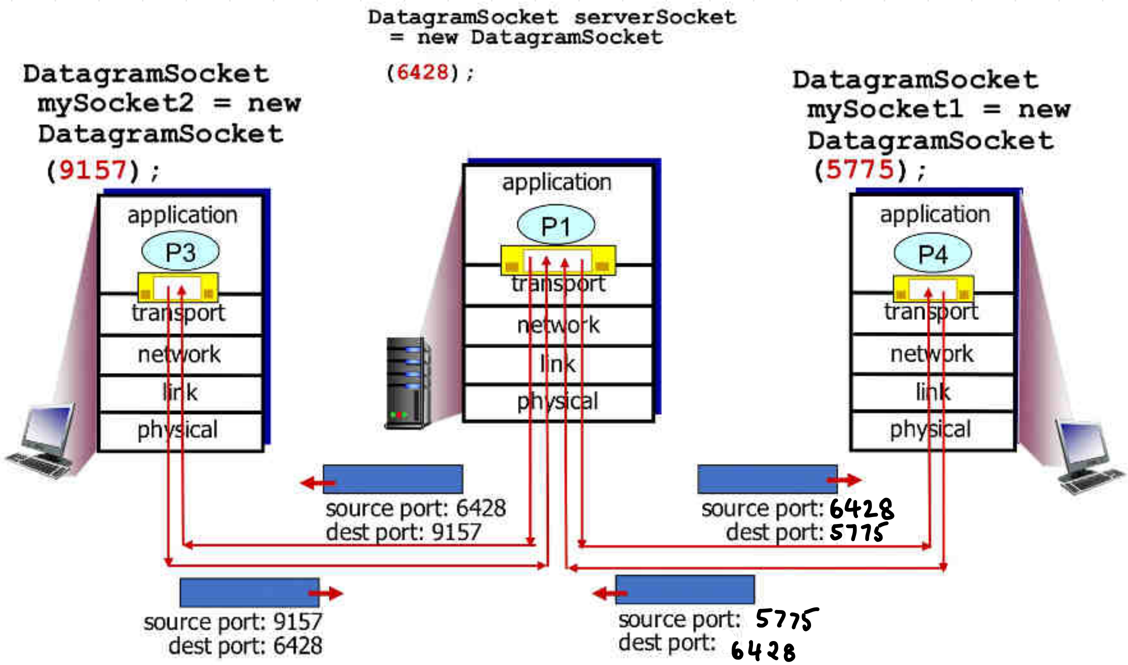
```
#!/usr/bin/python2

from socket import *

serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = raw_input('Input lowercase sentence: ')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

print modifiedMessage
clientSocket.close()
```

UDP segment

UDPServer.py

```
#!/usr/bin/python2

from socket import *

serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"

while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

UDP segment



CONNECTION-ORIENTED DEMULTIPLEXING

- TCP socket: 4 tuple
  - source IP
  - source port
  - dest IP
  - dest port
- Demux: receiver uses 4 values to direct segment to socket
- Server host: support multiple TCP sockets simultaneously (own 4-tuple)
- Web servers: diff socket for each client
  - non-persistent HTTP: different socket for each req
  - unlike connectionless

Example

TCPClient.py

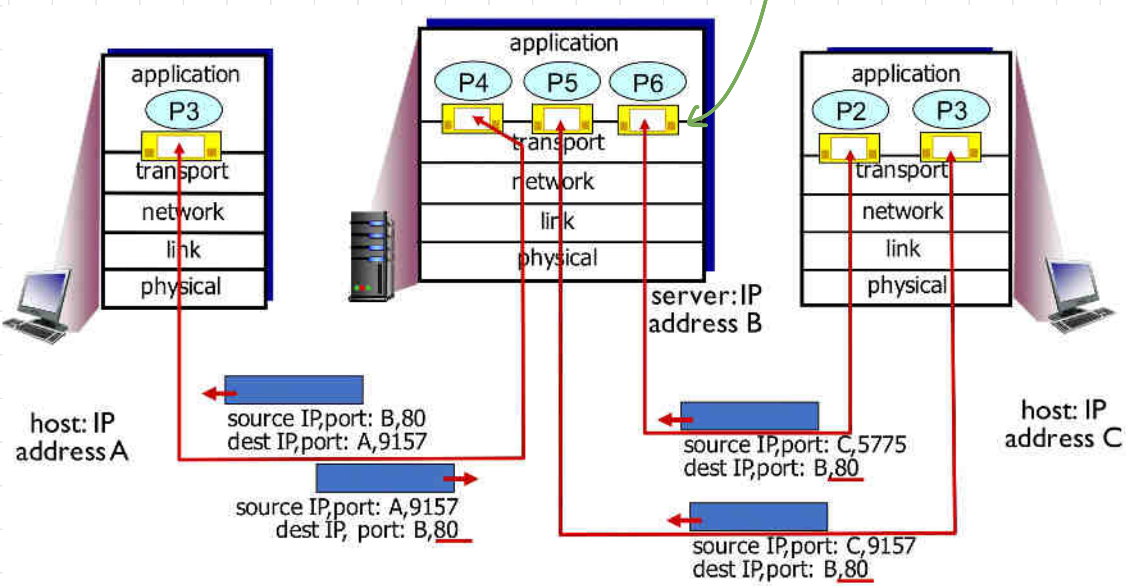
```
#!/usr/bin/python2

from socket import *

serverName = 'localhost'
serverPort = 12000

clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence: ')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server: ', modifiedSentence
clientSocket.close()
```

TCPServer.py

```
#!/usr/bin/python2

from socket import *

serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'

while 1:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
connectionSocket.close()
```

- TCP server application has welcoming socket that waits for connection establishment requests from TCP clients on port 12000

demultiplexed to
different sockets



Comparison

- UDP: demux only using dest port no.
- TCP: demux using 4 tuple
- Based on segment (TCP), datagram (UDP) header values

CONNECTIONLESS TRANSPORT LAYER PROTOCOL - UDP

- 'no frills', 'bare bones' (does not add too much)
- connectionless, unreliable, out of order, 'best effort'; no guarantee
- no handshake; each UDP segment independent of others
- 8 byte header overhead per segment

Use of UDP

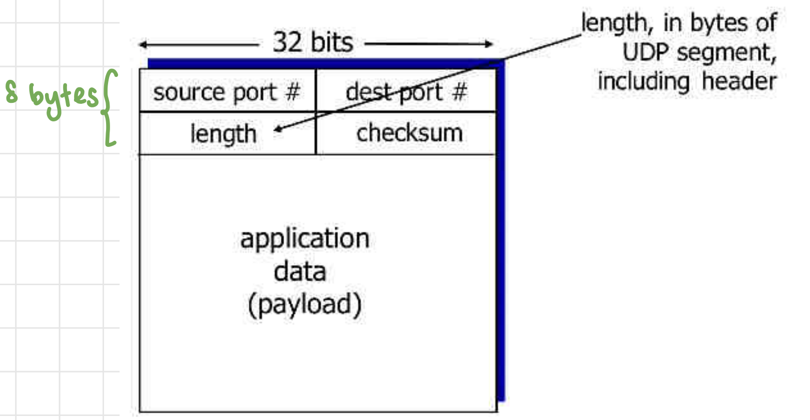
- no RTT delay due to connection (round trip time)
- no buffer/seq/ack/c-c parameters for connection state at sender and receiver
- small header size (8 bytes, not 20)
- no congestion control; possibility of loss but no speed limit at sender

Applications

- streaming multimedia
- DNS
- SNMP
- HTTP/3

- Reliability over UDP — HTTP/3
  - add reliability at app layer

UDP Segment Header



UDP segment format

UDP Checksum

- detect certain errors (flipped bits)
- sender: treats UDP segment (including header fields and IP addresses) as sequence of 16-bit ints
- Checksum: 1's comp sum of segment content, value put into UDP checksum field
- Receiver adds checksum to computed checksum w/o 1's comp to get string of 1's if no errors found
- Some errors not detected

Q: checksum example

```
    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
  ---------------------------------
  1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
                                  1
        wraparound
  ---------------------------------
sum       1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum  0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1    1's comp
```

Q: Calculate checksum

```
    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 0 0 1 1 1 1 0 0 0 0 1 1 0 0
  ---------------------------------
  1 0 1 0 0 1 0 1 0 1 1 0 0 0 0 0 1
                                  1
  ---------------------------------
sum       0 1 0 0 1 0 1 0 1 1 0 0 0 0 1 0
checksum  1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 1
```

Verify with receiver side

```
    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 0 0 1 1 1 1 0 0 0 0 1 1 0 0
  ---------------------------------
  1 0 1 0 0 1 0 1 0 1 1 0 0 0 0 0 1
                                  1
  ---------------------------------
sum              0 1 0 0 1 0 1 0 1 1 0 0 0 0 1 0
header checksum  1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 1
  ---------------------------------
                 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

note: CRC – cyclic reliability check — algorithm

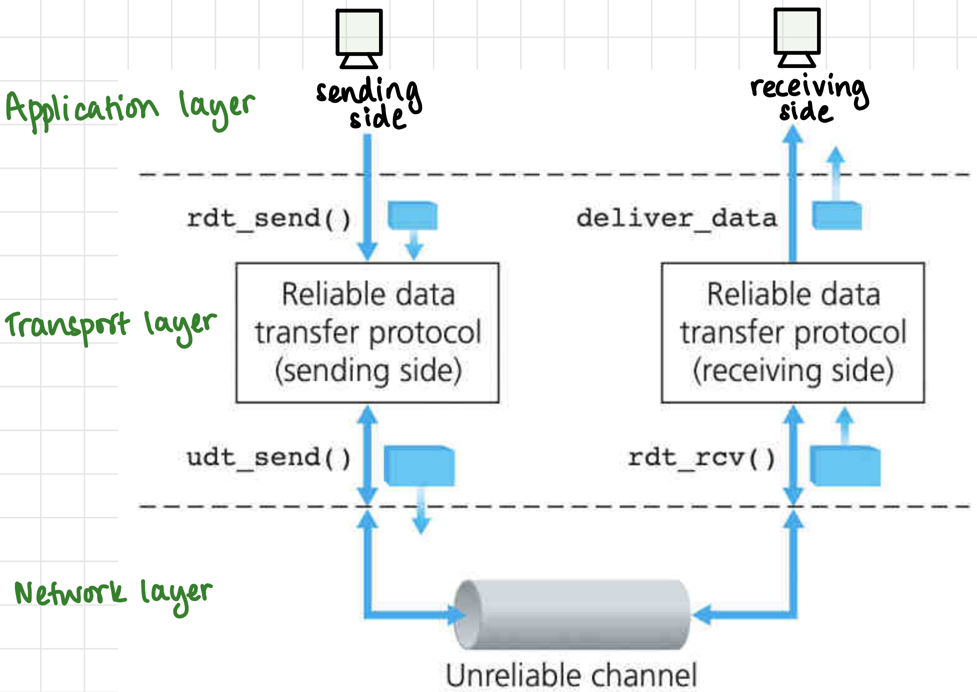
Principles of Reliable data transfer

- Unreliable channel below transport layer

- Complexity of reliable data transfer protocol depends on characteristics of unreliable channel

- Sender and receiver know nothing about state of the other; ack needed

- Functions:
  - rdt_send(): called from app layer: passes data to be delivered by receiver's above (app) layer
  - rdt_rcv(): called once packet arrives from receiving end of channel
  - udt_send(): called by rdt to transfer packet over unreliable channel
  - deliver_data(): called by rdt to deliver data to app layer



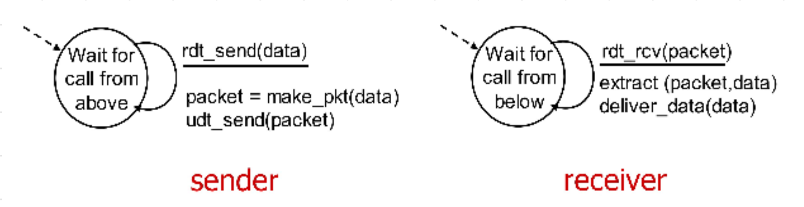
BUILDING A RELIABLE DATA TRANSFER PROTOCOL

rdt 1.0

- underlying channel assumed to be perfectly reliable (no errors/losses)
- event causing state transition: above horizontal line
  action taken when event occurs: below horizontal line

FSM

separate FSMs for sender and receiver



rdt 2.0

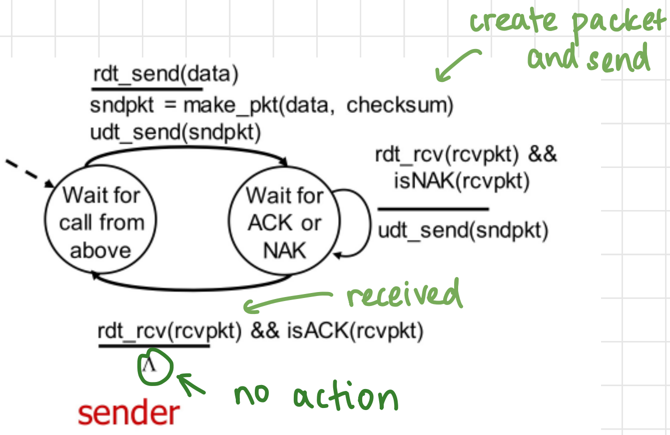
- underlying channel may flip bits (network layer) — bit errors
- checksum
- recover from errors (checksum)

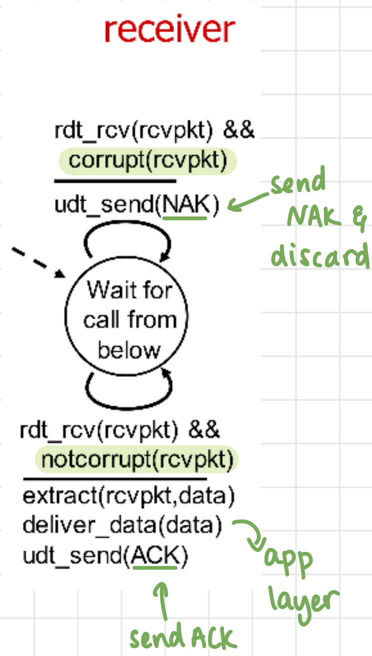
Acknowledgements

- ACKs: receiver explicitly tells sender that pkt received OK
- NAKs: receiver explicitly tells sender that pkt had errors; sender must retransmit prev. sent data

- stop and wait for ACK/NAK - one packet at a time
- Automatic Repeat Request (ARQ) Protocols

FSM





FLAW

- ACK / NAK corrupted
- multiple retransmissions - duplicate packets received when ACK/NAK corrupted
- Solution
  - packet seq no
  - receiver can discard duplicates

rdt 2.1

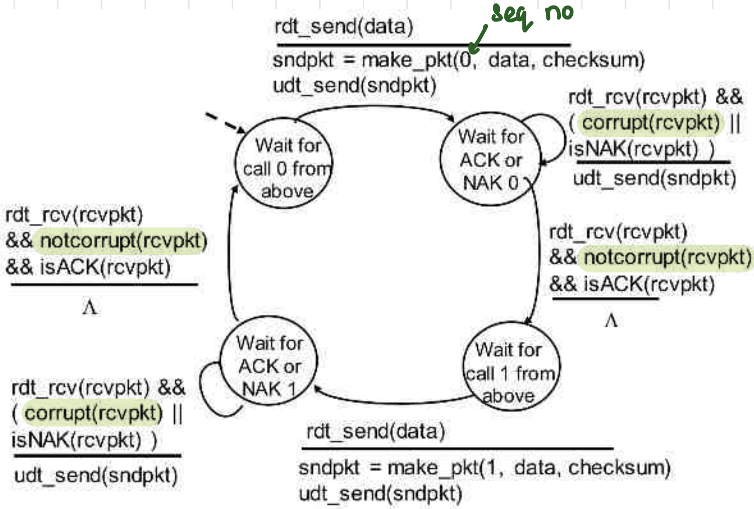
sender

- seq no to pkt
- only 2 seq no.s
- check if ACK/NAK corrupt
- twice as many states

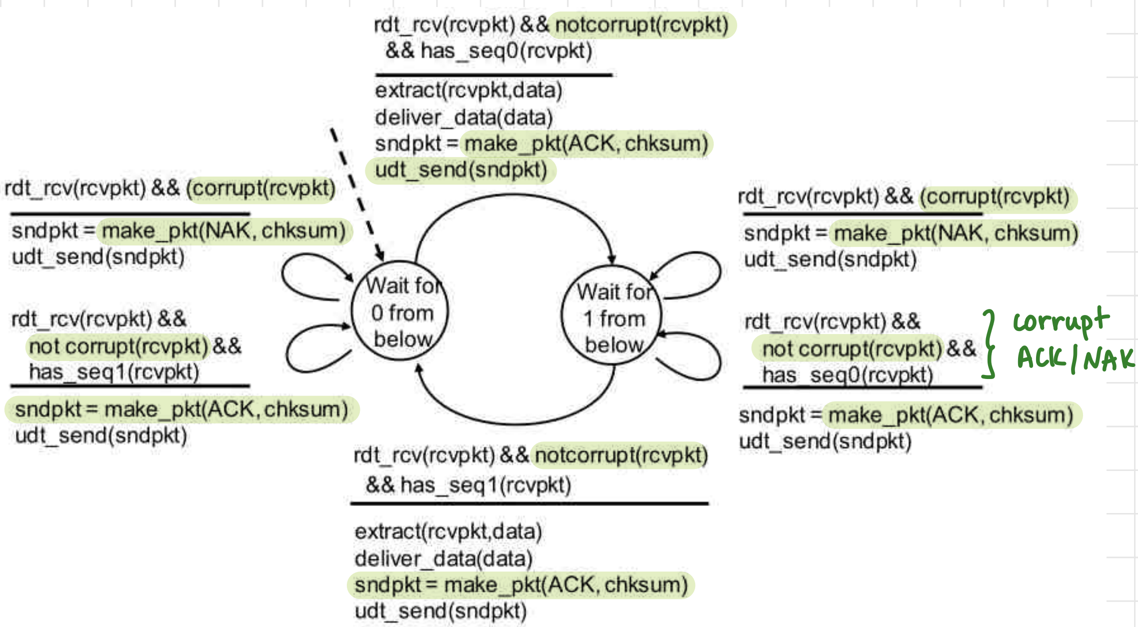
receiver

- check for duplicates
- does not know if ACK/NAK received OK

sender

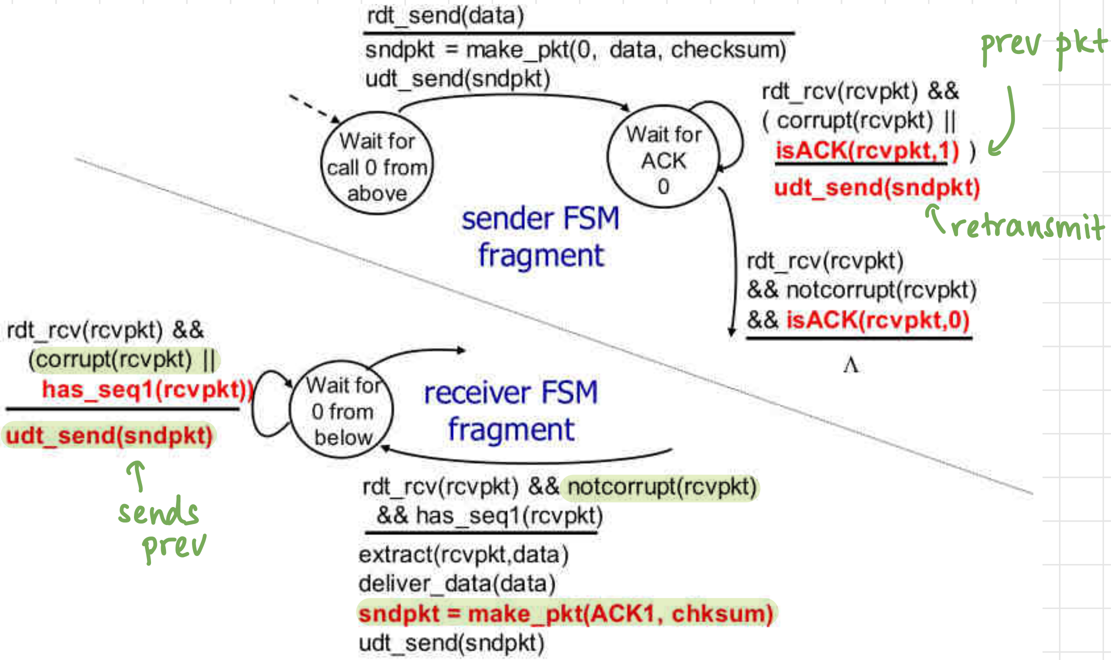


receiver



rdt 2.2

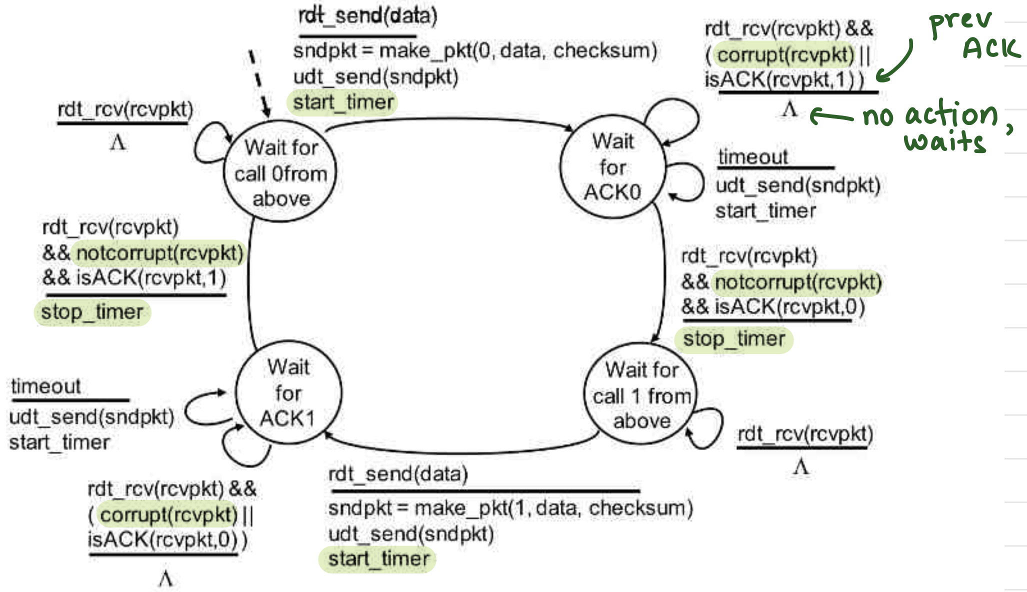
- NAK-free protocol
- only ACK
- ACK for prev packet



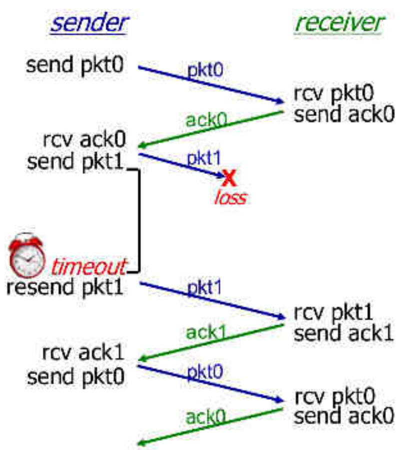
rdt 3.0

- underlying channel can also lose packets
- checksum, seq no, ACKs, retransmission
- sender waits for timeout time before retransmitting (due to loss or no ACK or delay)
- receiver specifies seq no
- more than RTT
- also called alternating-bit protocol

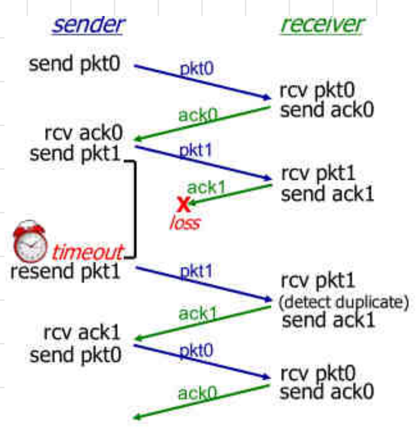
sender



in action

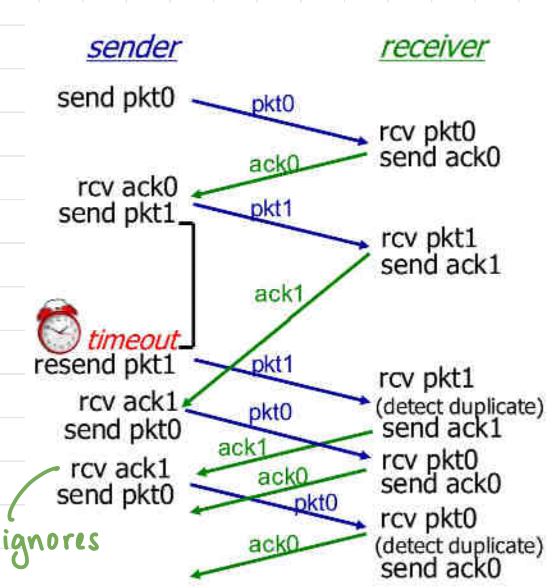


packet loss



ack loss

duplicate ACK
(ignores)



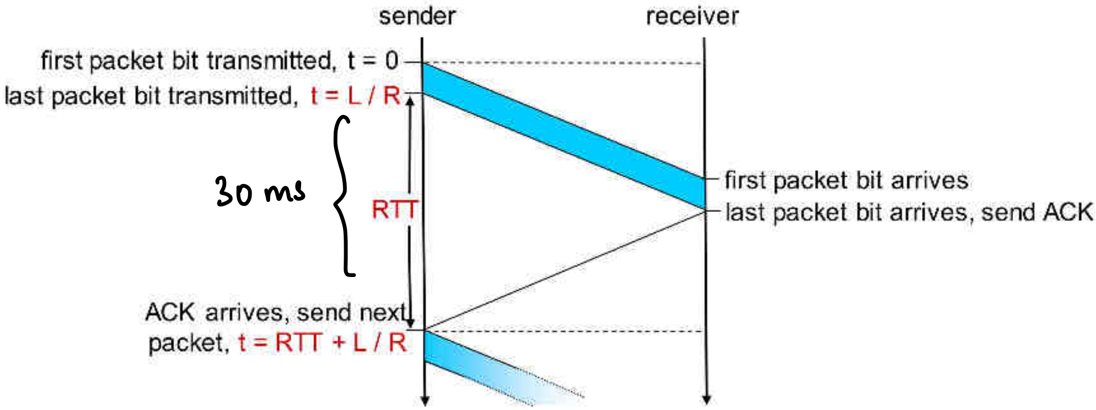
premature timeout

PERFORMANCE

- U_{sender} = utilisation = fraction of time sender busy sending

- eg: R = transmission channel = 1 Gbps link
  D_{prop} = prop delay = 15 ms
  L = packet = 8000 bits

- $D_{trans} = \frac{L}{R} = \frac{8000}{10^9} = 8 \times 10^{-6}\,s = 8\,\mu s$

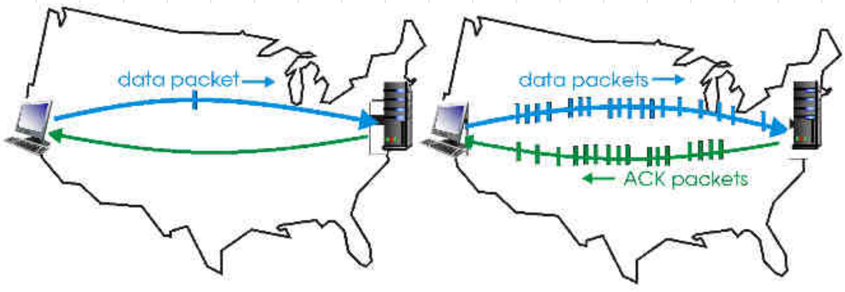


$$RTT = 2 \times D_{prop}$$

$$\boxed{U_s = \frac{L/R}{L/R + RTT}} = \frac{8 \times 10^{-3}}{30 + 8 \times 10^{-3}} = 0.00027$$

Solution: Pipelining

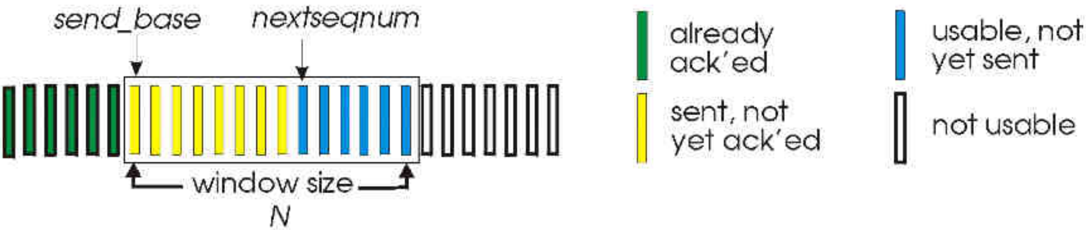
- range of seq increased
- buffers at both ends
- go-back-N, selective repeat



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

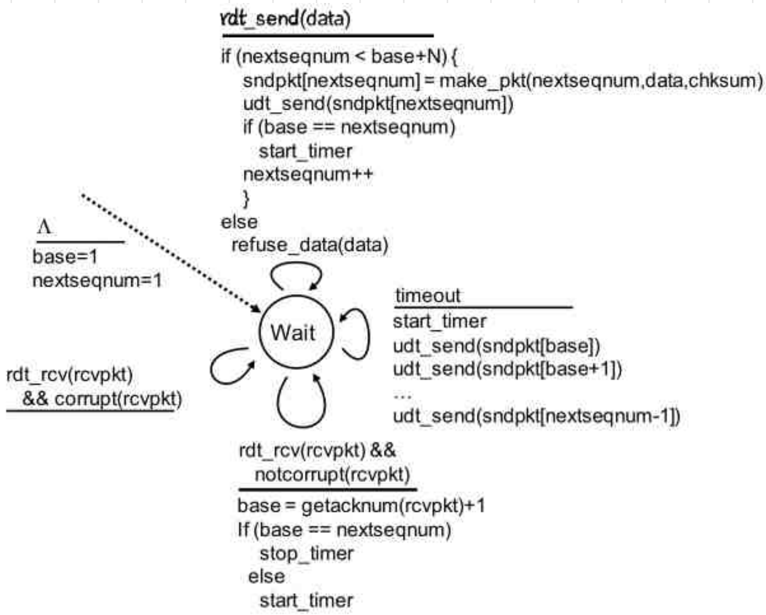
GO-BACK-N

- Sender can have upto N un-acked packets (consecutive); window of size N (N > 1)
- K-bit sequence number

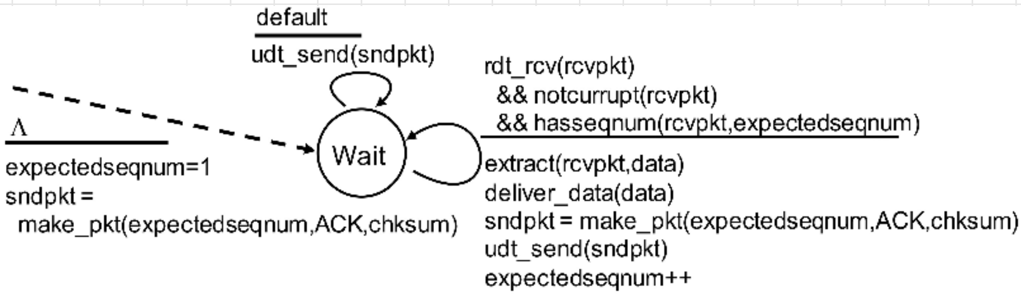


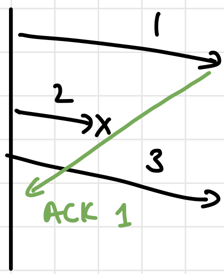
- Cumulative ACK(n) → all packets upto and including N
- timeout (n): retransmit #n and all higher ones

Sender fsm



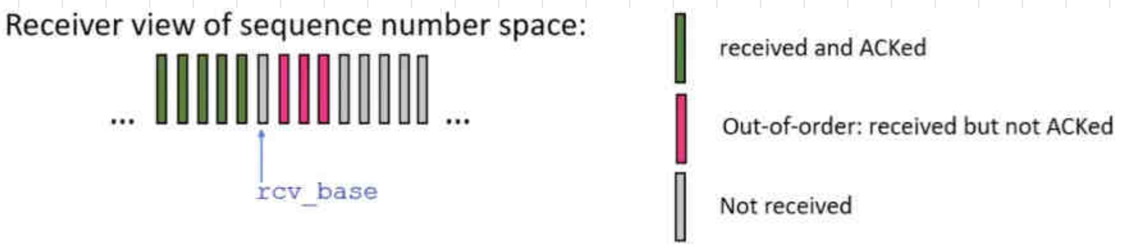
receiver fsm



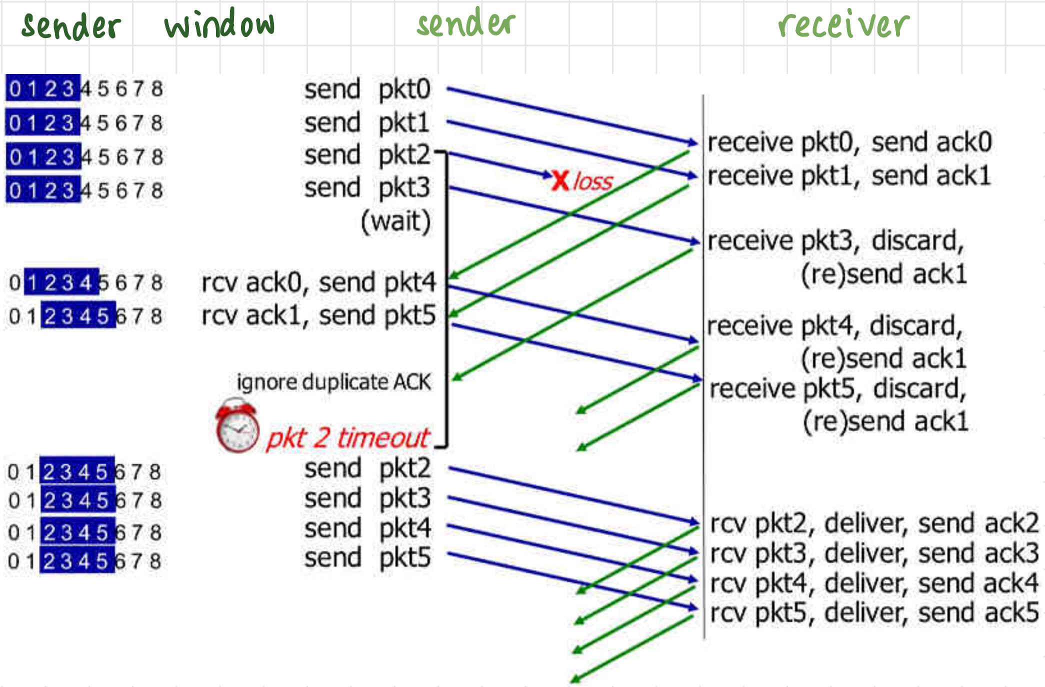


no ACK 3

$N_{receiver} = 1$



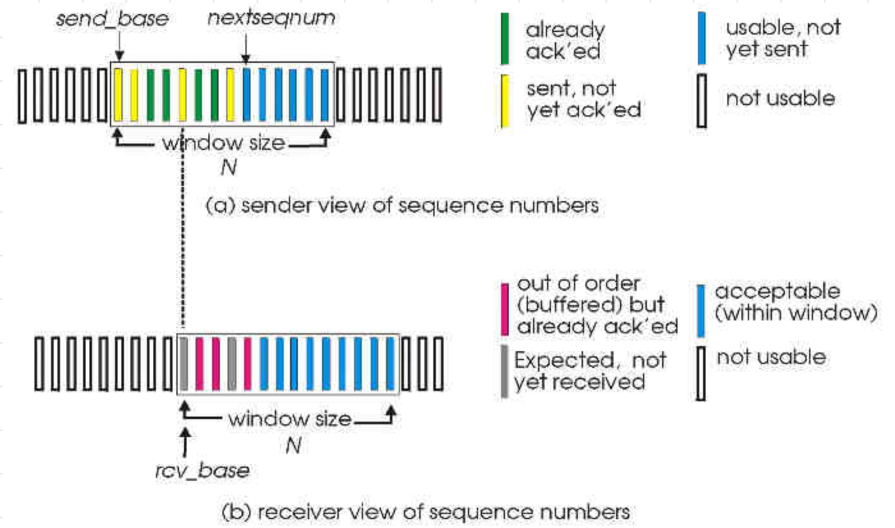
GB4 In Action



selective REPEAT

- Receiver individually acknowledges every packet (not cumulative)
- Buffers received packets to order before sending to app layer
- Window : N consecutive
- Timer maintained for each un ACKed packet

Selective Repeat



Sender

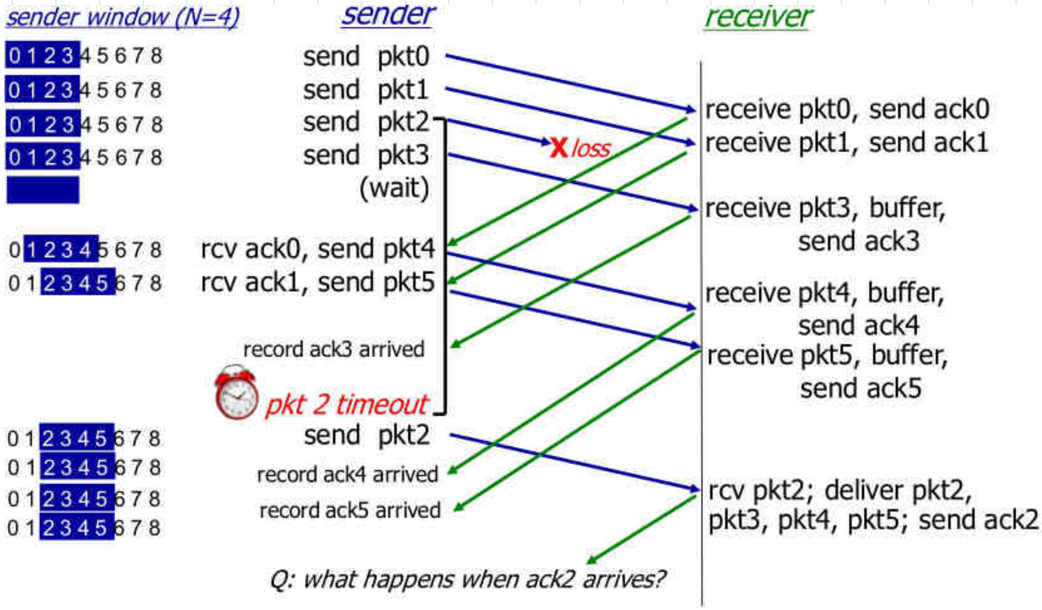
- data from above
  - if next seq # in window, send pkt
- timeout(n)
  - resend #n, restart time
- ACK(n) in [sendbase, sendbase + N]
  - mark n as received
  - if n smallest unACKed, advance window by 1

receiver

- packet n in [rcvbase, rcvbase + N-1]
  - send ACK(n)
  - out order: buffer
  - in order: deliver

- packet n in [rcvbase - N, rcvbase - 1]
  - Ack (n)
  - resent (eg: lost ACK)

- otherwise
  - ignore



Q: In GB3, if every 5th packet is lost, have to send 10 packets, how many transmissions?

1 2 3 4 5 6 7
ign ign

5 6 7 8 9
ign ign

7 8 9 10
ign

9 10

18

Q: Selective repeat, if every 5th packet is lost, have to send 10 packets, how many transmissions? window = 3

1 ✓ 2 ✓ 3 ✓ 4 ✓ 5 6 ✓ 7 ✓

5 ✓ 8 ✓ 9 10 ✓

9 ✓

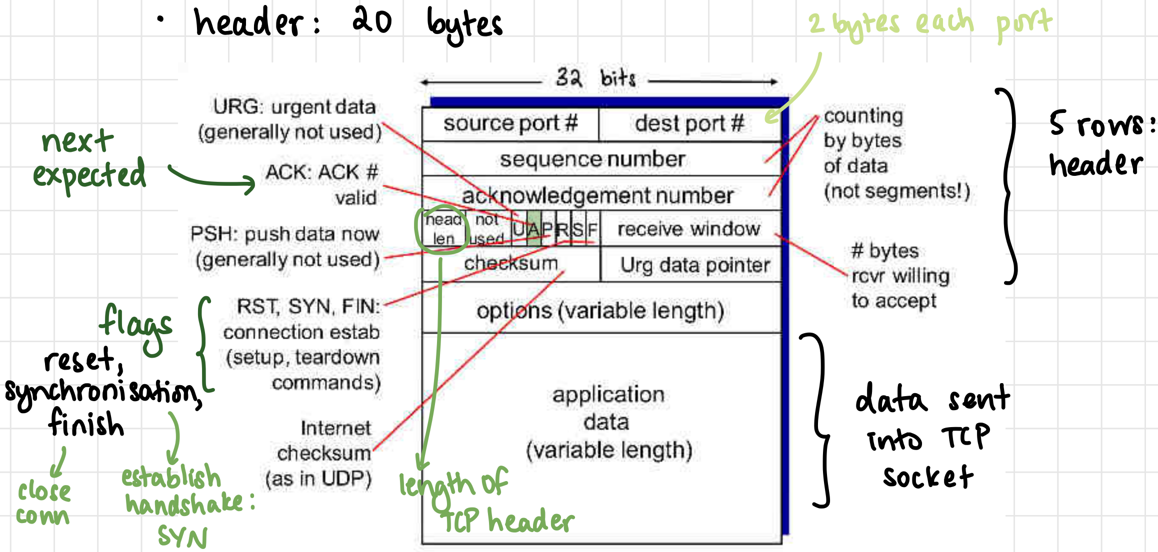
(12)

TRANSMISSION CONTROL PROTOCOL

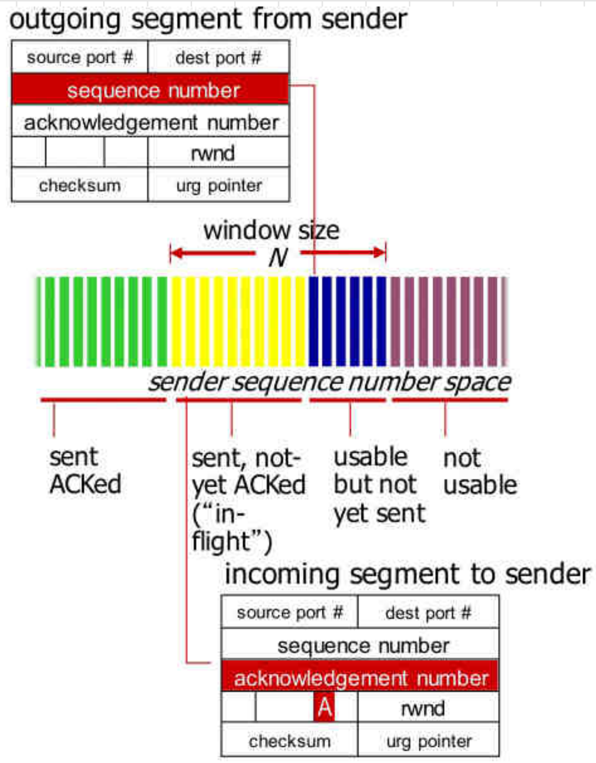
- point to point (one sender, one receiver)
- reliable, order
- full duplex
- MSS : maximum segment size
- cumulative ACKs
- pipelining ; congestion, flow control
- connection-oriented; handshaking

TCP Segment Structure

- header : 20 bytes



Sequence number



Sequence number

- byte stream "number" of first byte in segment's data
- byte numbering, not segment number

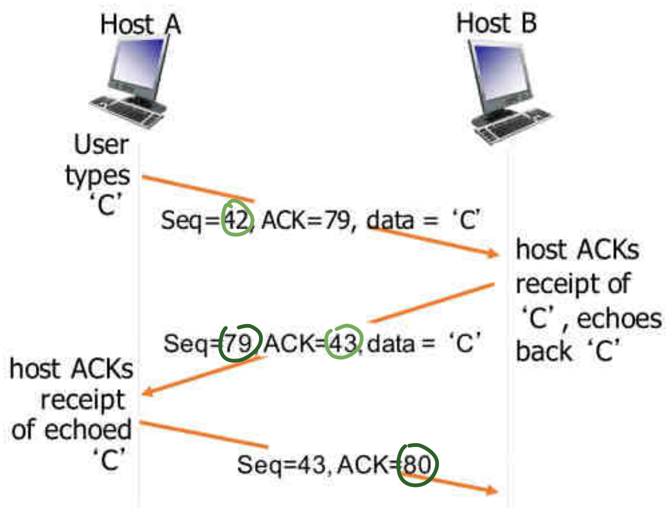
ACKs

- seq no of expected next byte
- cumulative ACK

Out of Order

- Up to implementer

Simple Telnet (port 23 - check unit 2)



simple telnet scenario

Timeout

- TCP timeout > RTT (varies)
- premature timeout; unnecessary retransmissions
- too long: slow rxn to segment loss

Sample RTT

- Measured time between segment transmission until ACK receipt (ignore retransmissions)
- Varies with every segment

Estimated RTT

- Exponential Weighted Moving Average (EWMA)
- Influence of past sample decreases exponentially fast
- Typical: $\alpha = 0.125$

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

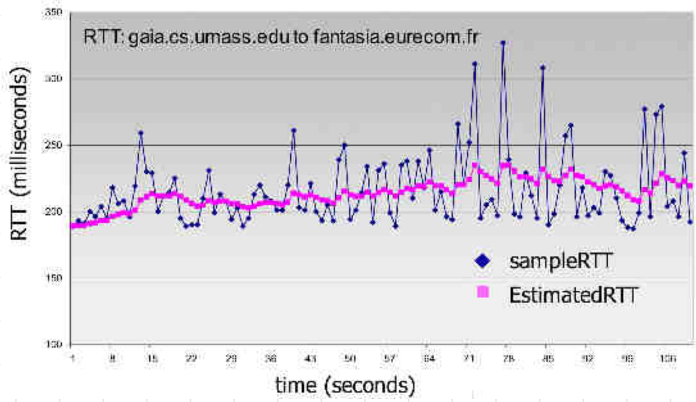
- Timeout interval: Estimated RTT + safety margin
  - if large var in Estimated RTT, larger safety margin

$$\boxed{\text{Timeout Interval} = \text{EstimatedRTT} + 4 * \text{DevRTT}}$$

- Deviated RTT: EWMA of SampleRTT's deviation from Estimated RTT

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

$\beta = 0.25$ usually



TCP Sender

1. Event: data received from app
   - create segment with seq no (byte stream no. of first data byte)
   - start timer if not already on
     - TimeOutInterval expiration period
     - for oldest un-ACKed segment

2. Event: timeout
   - retransmit segment
   - restart timer

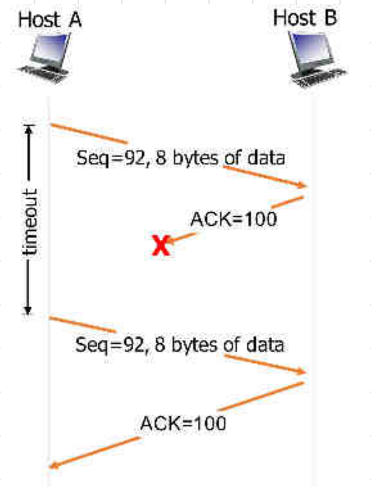
3. Event: ACK received
   - update what is known to be ACKed
   - restart timer if more unacked

TCP Receiver

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

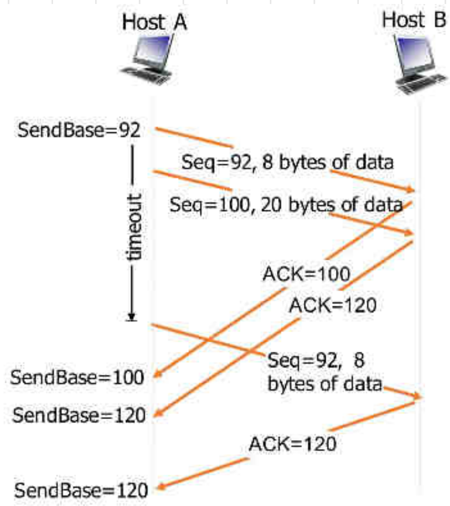
Retransmission Scenarios

1) Lost ACK



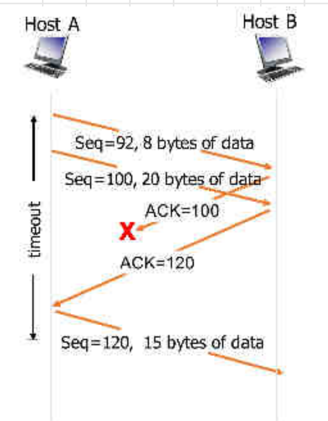
lost ACK scenario

2) Premature Timeout



premature timeout

3) Cumulative ACK covers for prev lost ACK

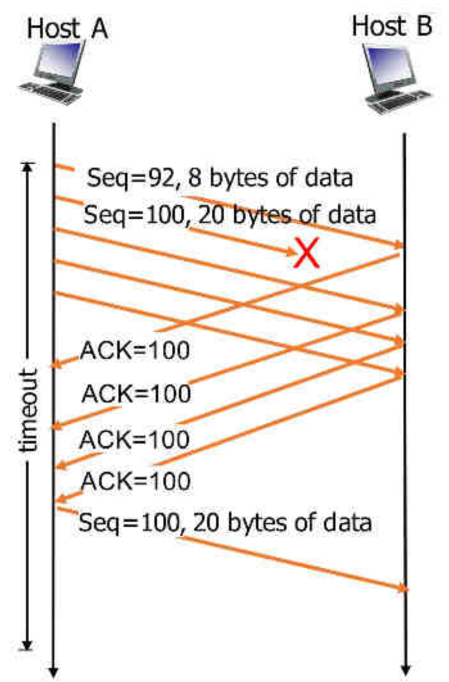


cumulative ACK

TCP Fast Retransmit

- Receipt of 3 duplicate ACKs — retransmit before timeout

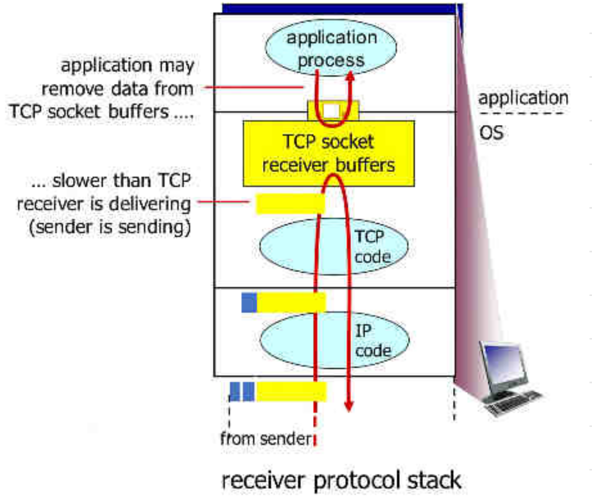
- Resend unACKed segment with smallest seq no



fast retransmit after sender receipt of triple duplicate ACK

TCP Flow Control

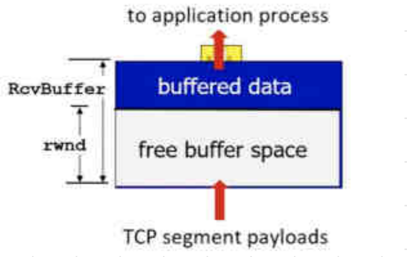
- Receiver controls sender so that sender does not overflow receiver's buffer by transmitting too fast



- TCP header: rwnd field

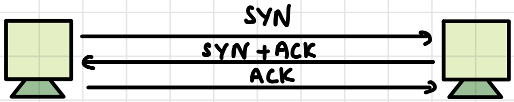
- Advertises free buffer space in rwnd
  - RcvBuffer size set via socket options
  - Autoadjusted by OS
  - Typically - 4096

- sender limits amount of unAcked data to the received rwnd

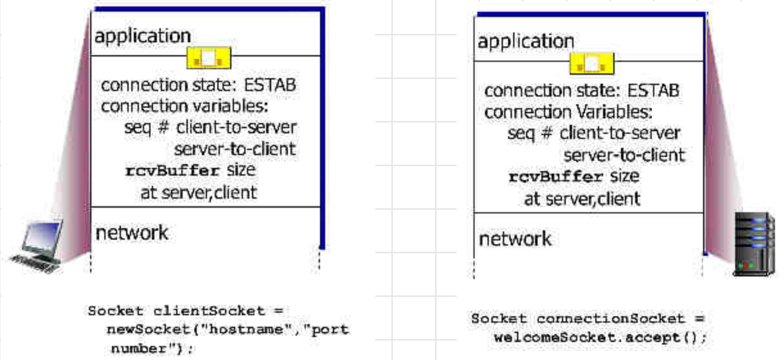


TCP receiver side buffering

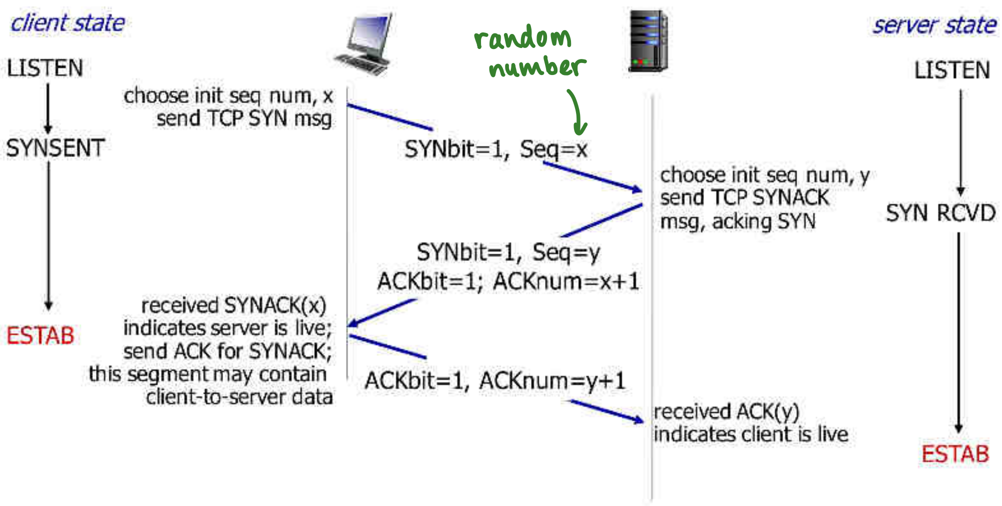
TCP 3-Way Handshake



- Establish handshake
- Agree on connection parameters

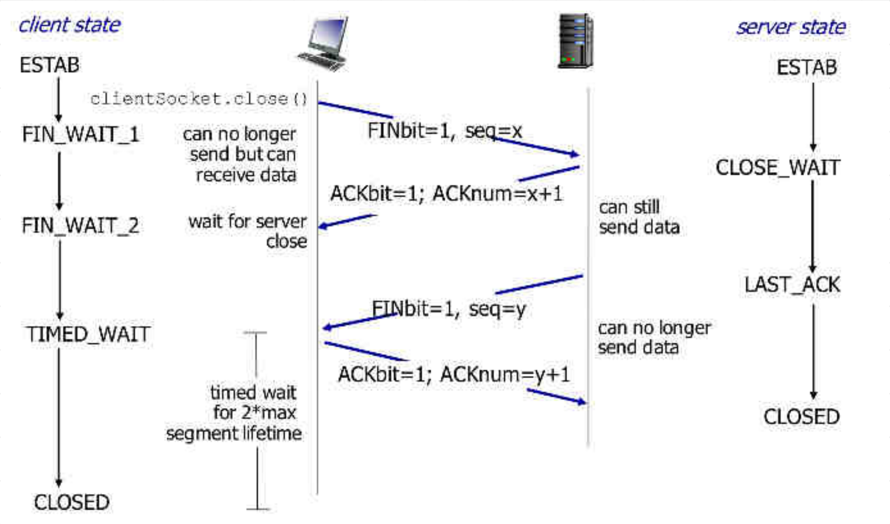


handshake



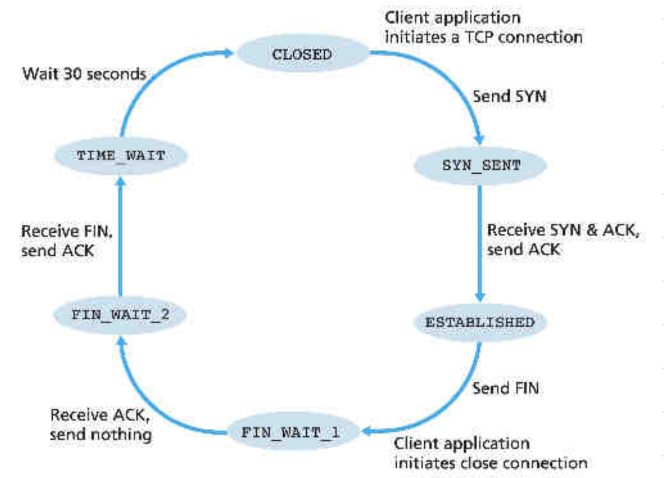
CLOSING CONNECTION

- send TCP segment with FIN=1
- respond to received FIN with ACK

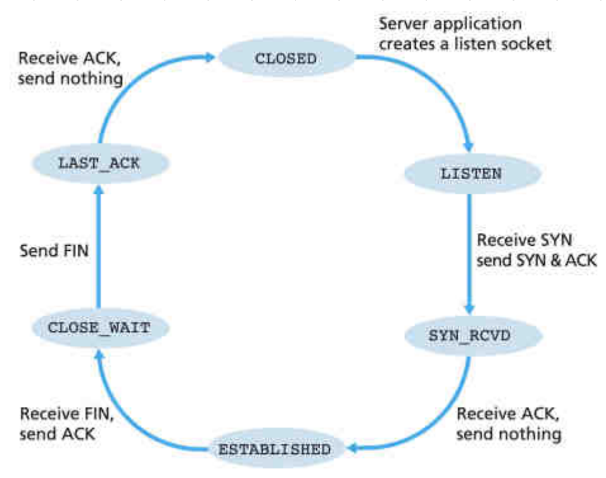


Sequence of TCP States

Client



Server



SYN-FLOODING ATTACK

- Large no. of SYN segments sent by attacker to a server with different fake source IP addresses
- Servers unnecessarily start allocating buffers and resources for all SYN requests and also sends back SYN + ACK segments
- Server eventually runs out of resources; may crash
- Denial-of-service attack (to genuine clients)

Congestion control

- Congestion: too many sources sending too many packets faster than network can handle

- Different from flow control

- Consequences: lost packets (buffer overflow at routers), long delays (queueing in router buffers)

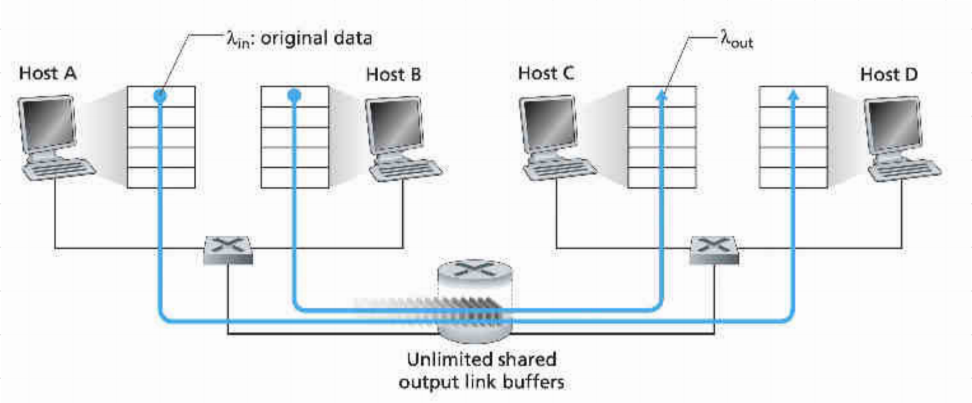
- Causes & costs of congestion: 3 scenarios of increasing complexities

Scenario 1 : Two Senders, a Router with Infinite Buffers

- Host A sending data at rate λ_{in} bytes / sec

- Host B sending data at rate λ_{in} bytes / sec

- Router outgoing link capacity R, infinte buffer



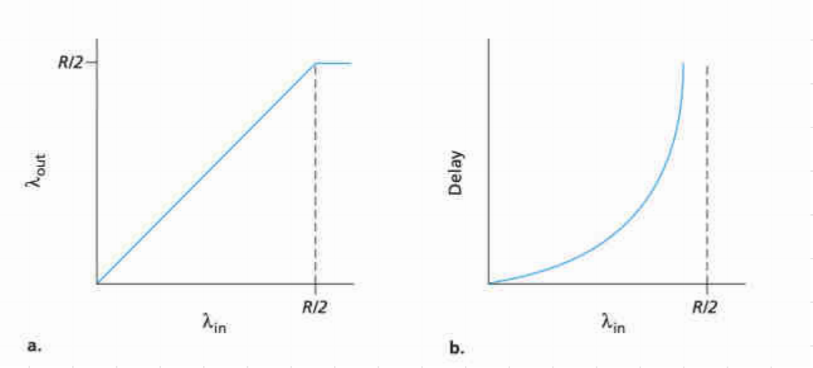
- Graph a: per-connection throughput (bytes/sec at receiver) as a function of λ_{in}

- While $\lambda_{in} \leq R/2$, throughput at receiver's end = λ_{in}

- When $\lambda_{in} > R/2$, the throughput remains R/2 (limit due to sharing capacity)

- Graph b: Average delay for packet to arrive at receiver

- Average no. of queued packets unbounded, delays unbounded as $\lambda_{in} \to R/2$



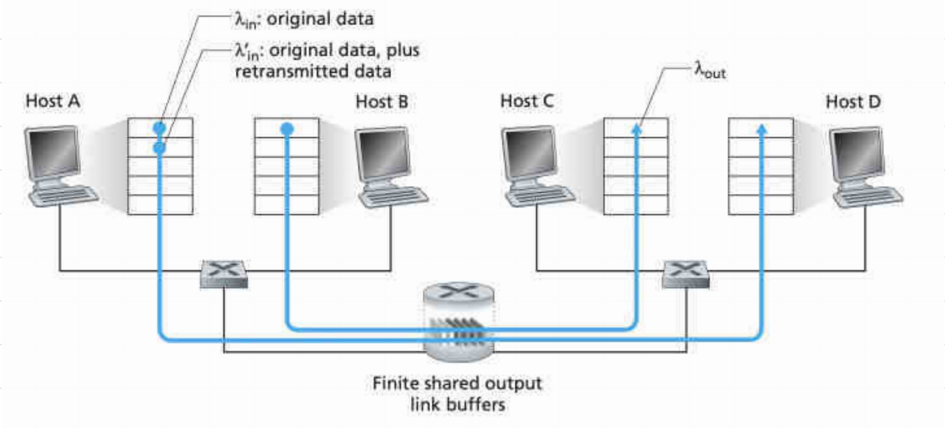
Scenario 2 : Two Senders, a Router with Finite Buffers

- Packets arriving to full buffer are dropped

- Retransmissions for lost/dropped packets

- λ_{in}: rate at which application sends data into socket

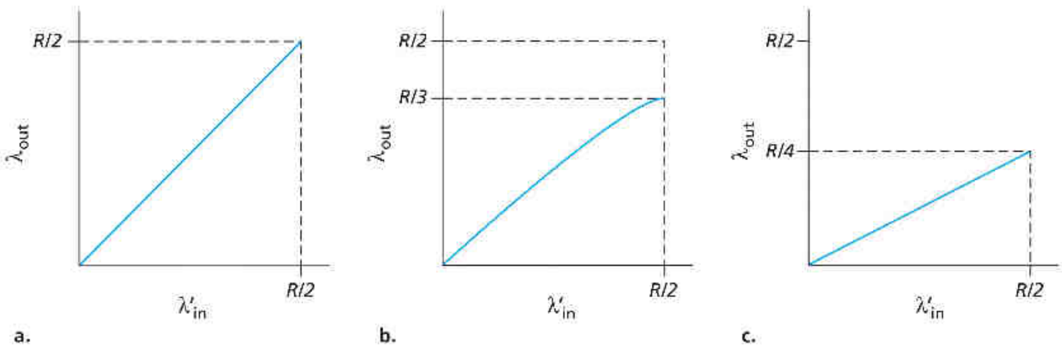
- λ'_{in}: rate at which transport layer sends segments (original data + retransmissions) — offered load to the network



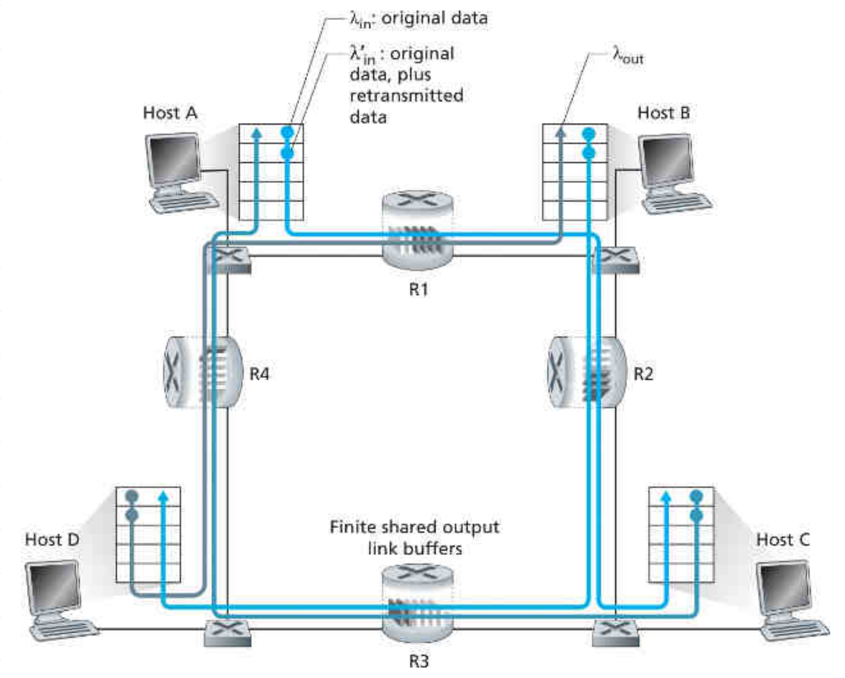
- Graph a: ideal case of sender transmitting only when buffers are free — $\lambda_{in} = \lambda'_{in}$

- Graph b: retransmission done only when packet is known for certain to be lost (large timeout; not very practical) — in graph, 1/3 rd of packets are retransmitted in each half)

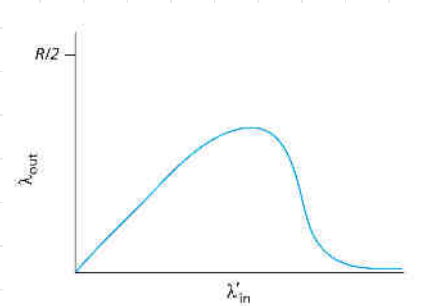
- Graph c: premature timeout for retransmission, duplicates at receiver end



Scenario 3 : Four Senders, Routers with Finite Buffers, Multihop Paths



- Low traffic, as λ_{in} increases, λ_{out} increases
- High traffic, hosts compete at routers



CONGESTION CONTROL

- Sender keeps track of congestion window variable (cwnd) such that the amount of un-ACKed segments at the sender cannot exceed min{cwnd, rwnd}

- Sender's send rate $\approx \frac{cwnd}{RTT}$ bytes/sec (ignoring rwnd)

- Size of cwnd changes based on packet loss (regulated at sender side)

- TCP said to be self-clocking

- TCP sender's rate should be decreased on packet loss as it indicates congestion (cwnd should decrease)

- TCP sender's rate should be increased when an ACK arrives for a previously unACKed packet (cwnd should increase)

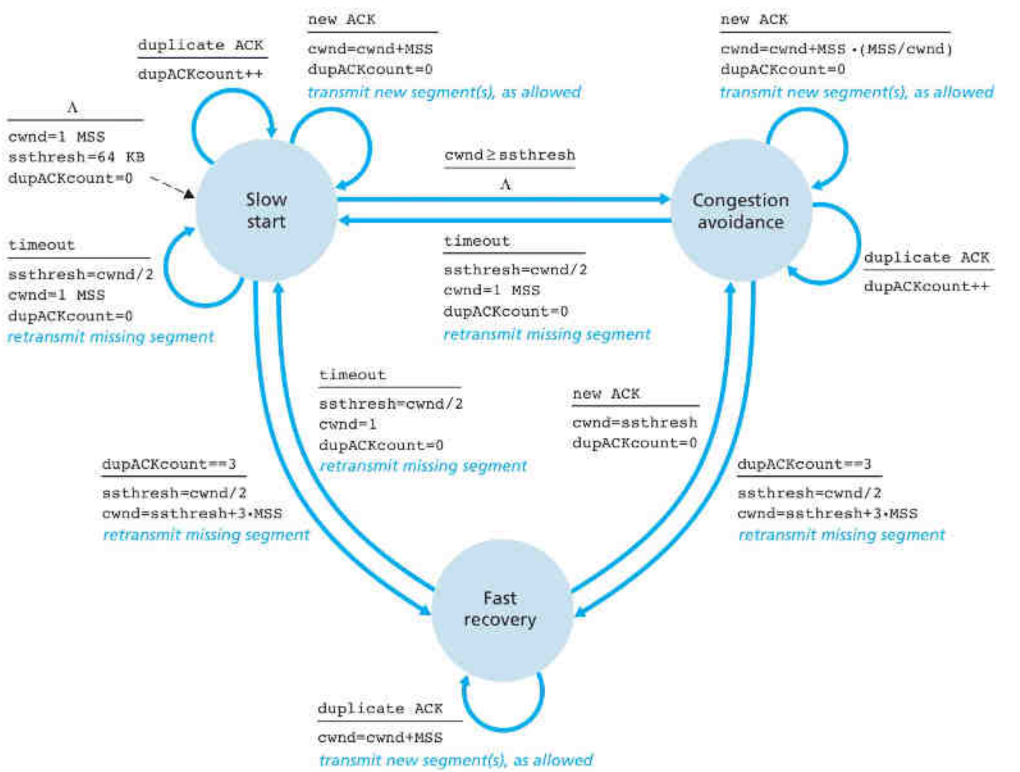
- TCP Congestion Control Algorithm
  - Slow start
  - Congestion avoidance
  - Fast recovery

1. Slow start

- cwnd initialised to 1 MSS, rate = 1 MSS/RTT bytes/sec

- After every ACK, cwnd increases by 1, resulting in doubling of cwnd every RTT — slow start

- First RTT, cwnd = 1; after ACK, cwnd = 2; after both received and ACKed, cwnd = 4 and so on — exponential growth
- If there is a loss event (due to timeout), cwnd reset to 1 and sets threshold value ssthresh to cwnd/2
- When cwnd = ssthresh, slow start ends and congestion avoidance starts
- If 3 duplicate ACKs arrive, TCP enters fast recovery state
- Shown in FSM below



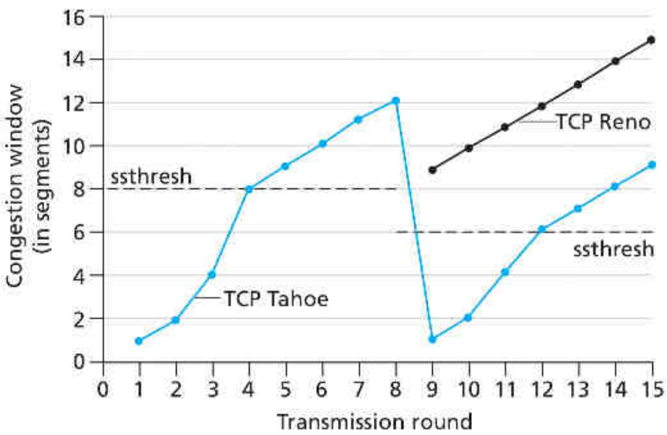
2. Congestion Avoidance

- When this state reached, value of cwnd equal to half its value before congestion encountered
- Value of cwnd increases by 1 MSS every RTT (MSS/cwnd multiples of MSS after every ACK)
- When timeout occurs, ssthresh set to half of cwnd and cwnd = 1 MSS
- When triple duplicate ACK occurs, cwnd is halved and ssthread is set to half cwnd when triple duplicate ACKs were received, then fast recovery state

3. Fast Recovery

- Value of cwnd increased by 1 MSS for every duplicate ACKs until 3
- If ACK for missing segment arrives, TCP deflates cwnd and enters congestion avoidance
- Timeout: cwnd reset to 1, ssthresh = old cwnd/2, enters slow start state
- Old version: TCP Tahoe — whether timeout or triple duplicate, cwnd always reset to 1 MSS and entered slow start

- New version: TCP Reno — incorporates fast recovery



ADDITIVE INCREASE, MULTIPLICATIVE DECREASE

- AIMD form of congestion control
- sawtooth