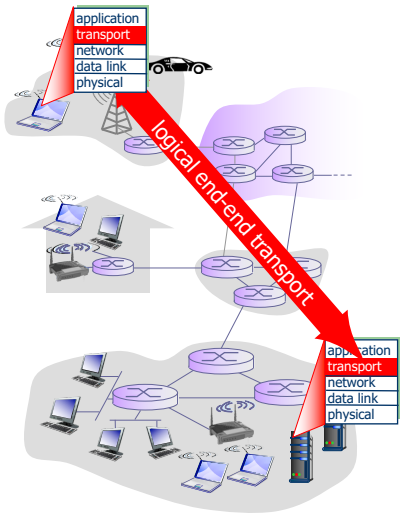
Transport Layer

- Transport Layer Services
- Multiplexing and Demultiplexing
- Connectionless Transport: UDP
- Connection-Oriented Transport: TCP
- TCP Congestion Control

1

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 / UDP



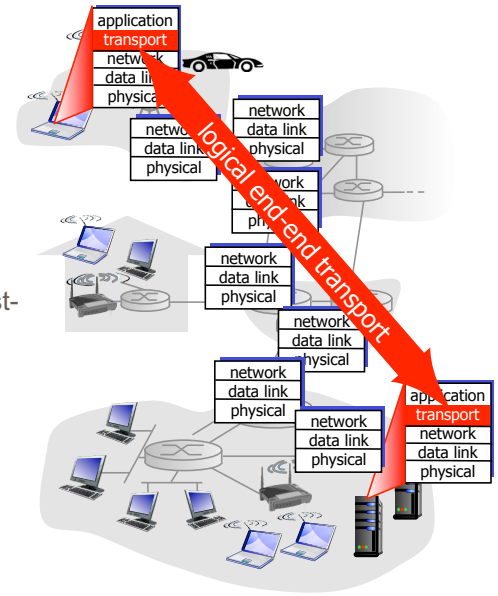
2

The act of requiring a connection leave this vulnerable to eg. DDOS

Internet Transport-Layer Protocols

- TCP - Reliable, in-order delivery
  - Congestion control
  - Flow control
  - Connection setup
- UDP - Unreliable, unordered delivery
  - No-frills extension of "best-effort" IP
    Still provides a port
- Services not available
  - Delay guarantees
  - Bandwidth Guarantees

Q: What type of services can we run over TCP & UDP?

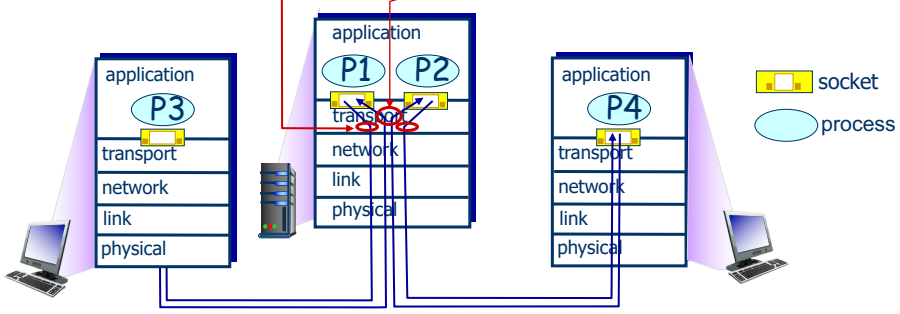


3 - TCP : data application that requires things to be in order
- UDP : voice/video/media. better to drop packets than stutter

Multiplexing and Demultiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



Q: How many addresses does a typical host have?

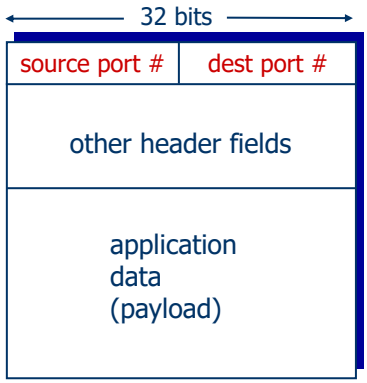
4 3, link layer (MAC addr), network (IP addr), transport (port)

5

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 number

Host uses IP addresses and port numbers to direct segment to appropriate socket



TCP/UDP segment format

5

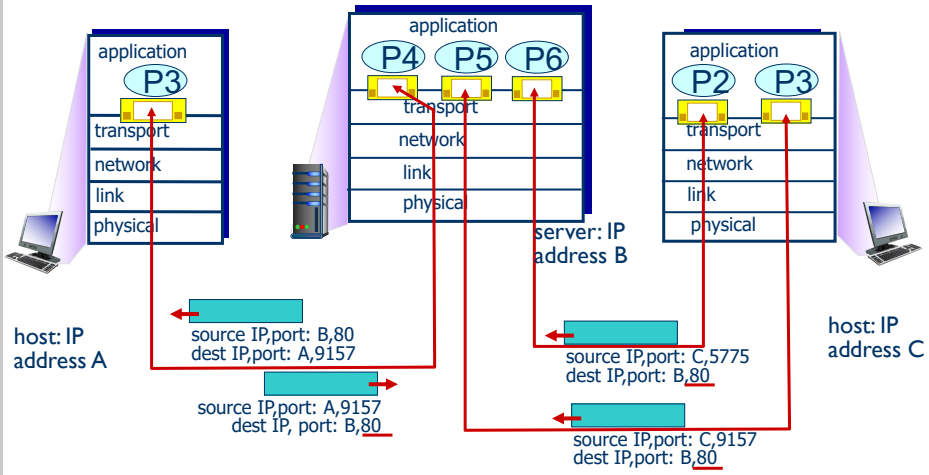
6

Connection-Oriented Demux

- TCP socket identified by 4-tuple
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Receiver uses all four values to direct segment to appropriate socket
  - Demultiplexing
- 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 a different socket for each request

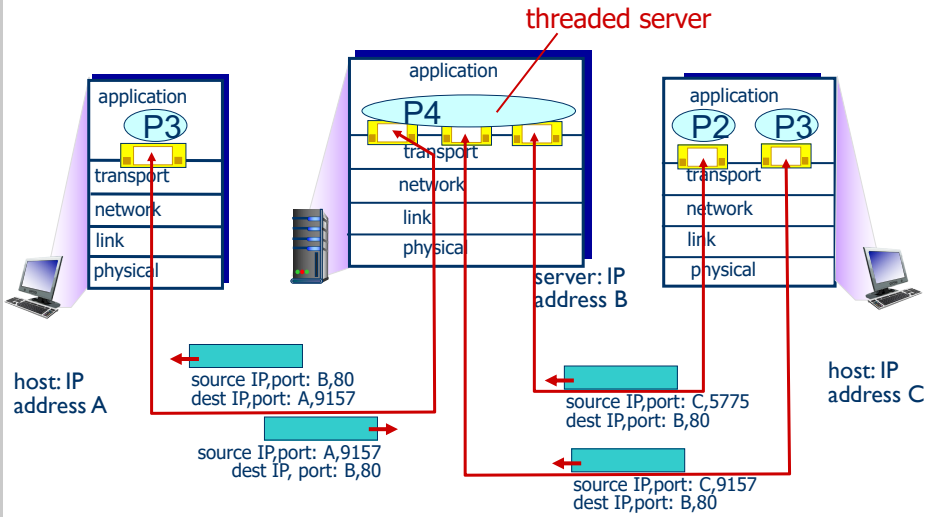
6

Connection-Oriented Demux Example



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

Connection-Oriented Demux Example II



Q: Why use threads?

Much lighter weight

9

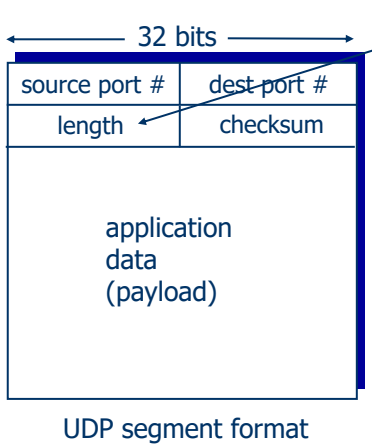
User Datagram Protocol (UDP)

- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be
  - Lost
  - Delivered out-of-order to app
- Connectionless
  - No handshake between sender and receiver
  - Each UDP segment handled independently of others
- Uses
  - Streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS Just send another request if there's an issue
  - SNMP network management protocol

9

10

UDP Header



UDP segment format

Why is there a UDP?

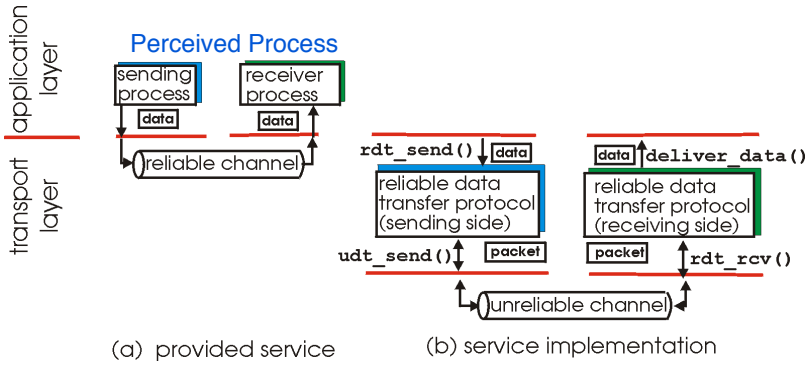
- No connection establishment
  - Which can add delay
- Simple - No connection state at sender, receiver
- Small header size
- No congestion control
  - UDP can blast away as fast as desired
  - This can cause issues on the network

10

11

Principles of Reliable Data Transfer

- Important in application, transport, link layers



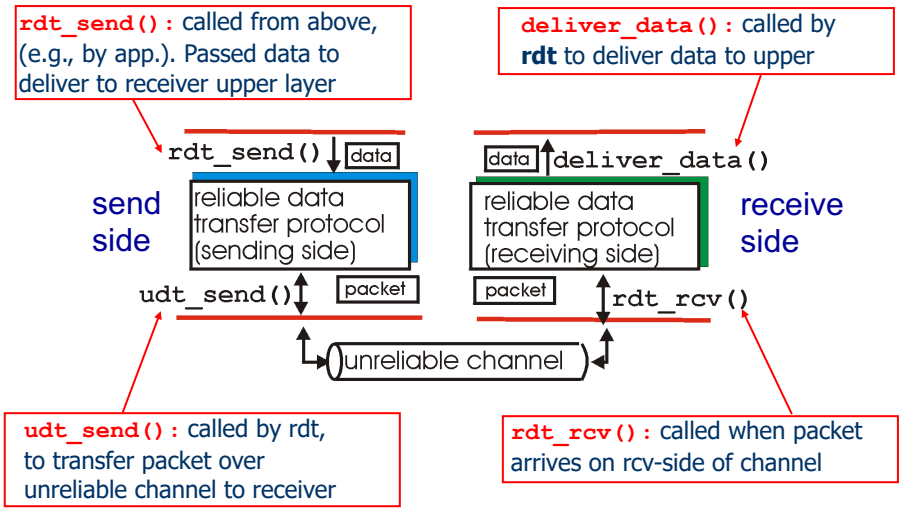
- Characteristics of the unreliable channel will determine complexity of *reliable data transfer* protocol

rdt - reliable data transfer
udt - unreliable data transfer

11

12

Reliable Data Transfer (RDT)

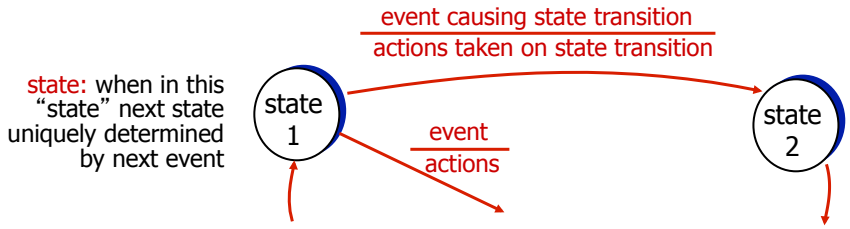


12

13

RDT

- We will incrementally develop sender, receiver sides of reliable data transfer protocol
- Consider only unidirectional data transfer
  - Control info flows in both directions
- Use finite state machines (FSM) to specify sender, receiver



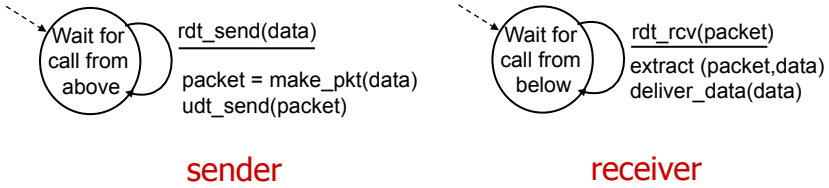
13

14

Not a real world scenario

rdt1.0: Reliable Transfer Over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- Separate FSMs for sender and receiver
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



sender receiver

14

15

rdt2.0: Channel with Bit Errors

- Underlying channel may flip bits in packet
  - Checksum to detect errors
- 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
  - Feedback
    - Control msgs (ACK, NAK) from receiver to sender

Must be contained in header

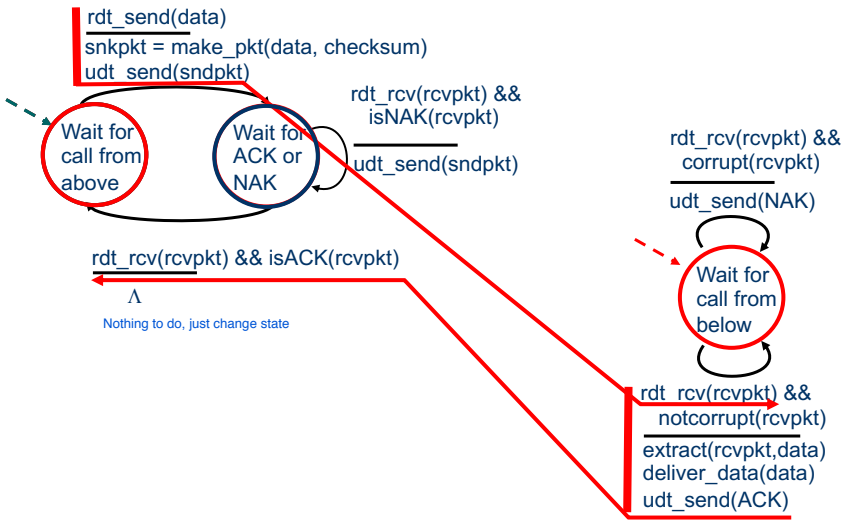
Buffering of all packets that haven't been ACK'd on sender side

Can use 2 bits for ACK/NAK ie each has one field, or use same bit, 1 = ACK, 2 = NAK

15

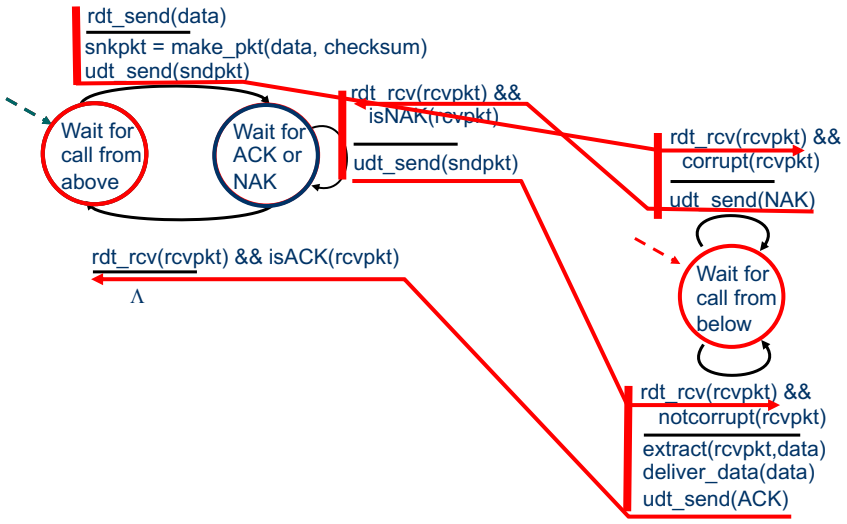
16

rdt2.0: Operation with No Errors



16

rdt2.0: Error Scenario



17

rdt2.0: Fatal Flaw

- What happens if ACK/NAK corrupted?
  - Sender does not know what happened at receiver!
  - Cannot just retransmit

    — Possible duplicate

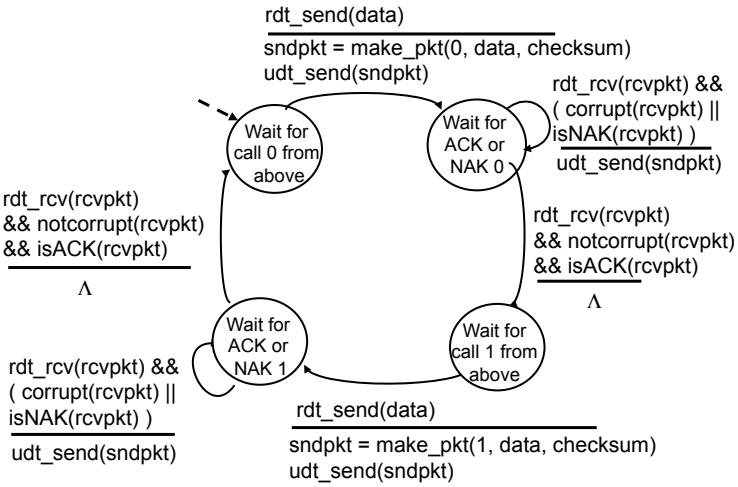
- Handling Duplicates
  - Sender retransmits current pkt if ACK/NAK corrupted

    — sender adds sequence number to each pkt 32-bit seq no. in TCP Realistically how many frames?
  - Receiver discards (does not deliver up) duplicate pkt

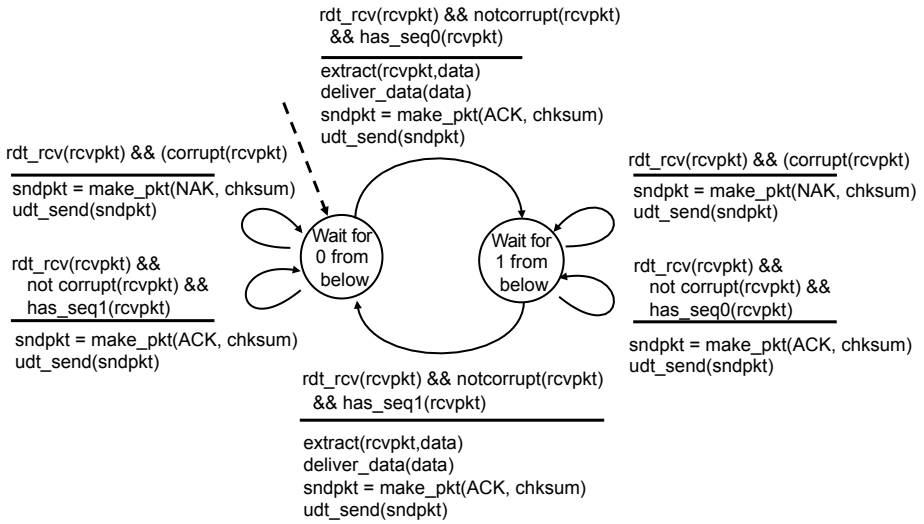
- Stop and Wait
  - Sender sends one packet, then waits for receiver response

18

rdt2.1: Sender Handles Garbled ACK/NAKs



rdt2.1: Receiver Handles Garbled ACK/NAKs



21

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 "expected" pkt should have seq # of 0 or 1
- Receiver
  - Must check if received packet is duplicate
  - Note: receiver cannot tell if its last ACK/NAK received OK at sender

21

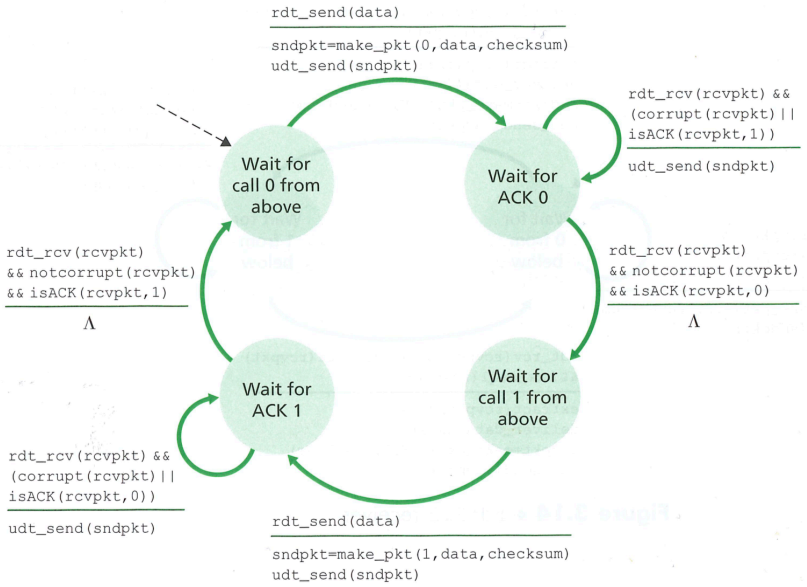
22

rdt2.2: NAK-free Protocol

- Same functionality as rdt2.1
  - Using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
- Duplicate ACK at sender results in same action as NAK
  - Retransmit current pkt

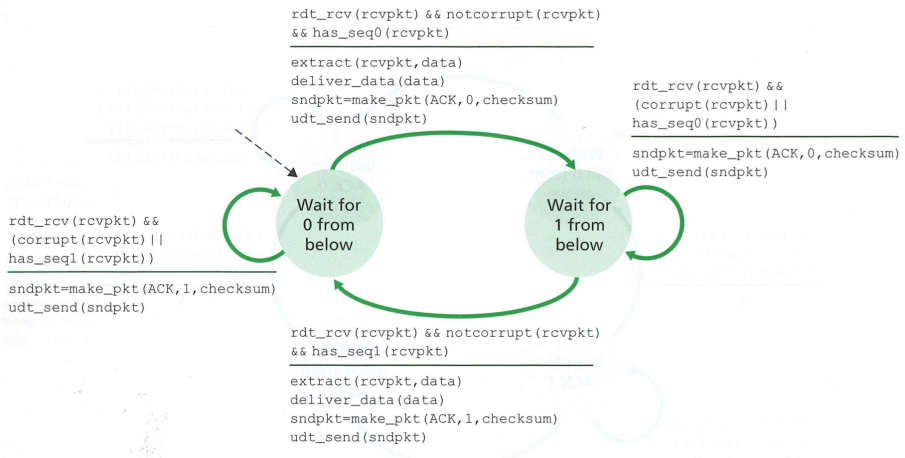
22

rdt2.2: Sender FSM



23

rdt2.2: Receiver FSM

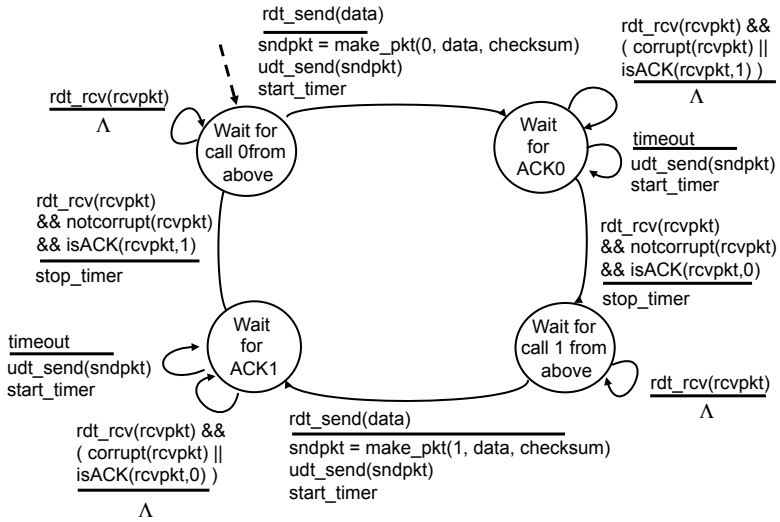


24

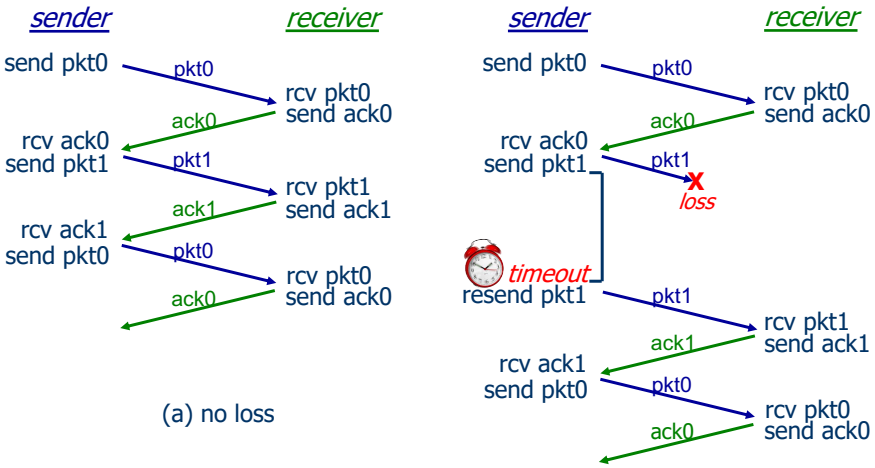
rdt3.0: Channels with Errors and Loss

- New Assumption
  - Underlying channel can also lose packets (data, ACKs)
  - Checksum, seq #, ACKs, retransmissions will be of help
    - But not enough
- Approach
  - Retransmits if no ACK received in this time
  - If pkt (or ACK) just delayed (not lost)
    - Retransmission will be duplicate, but seq #'s already handle this
    - Receiver must specify seq # of pkt being ACKed
  - Requires countdown timer

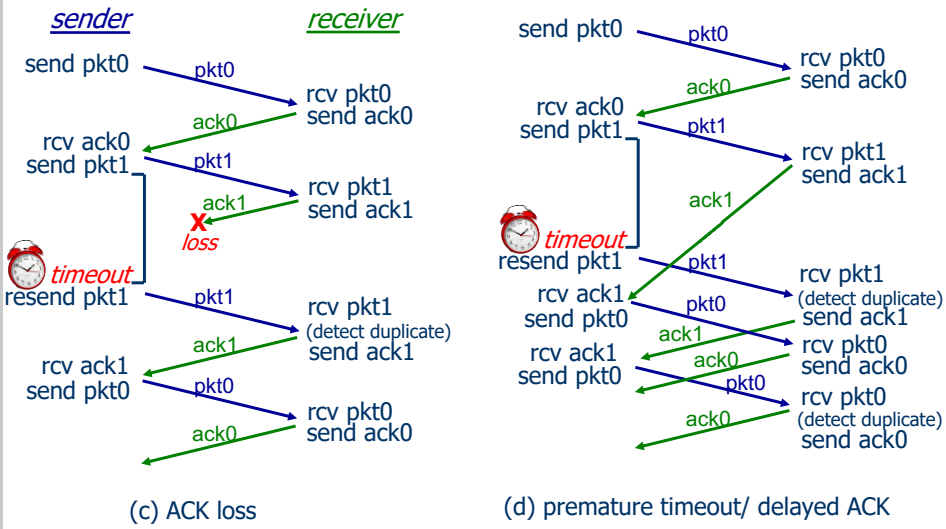
rdt3.0: Sender



rdt3.0: In Action



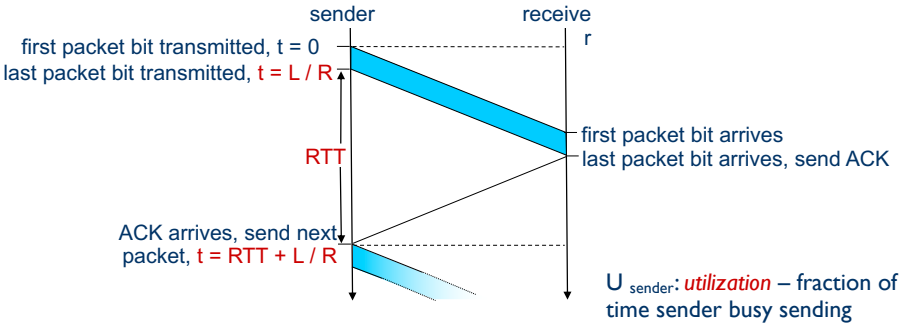
rdt3.0: In Action II



rdt3.0: Stop-and-Wait Operation

- rdt3.0 performance stinks
  - e.g. 1 Gbps link (R), 15 ms prop. delay, 8000 bit packet (L)

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$



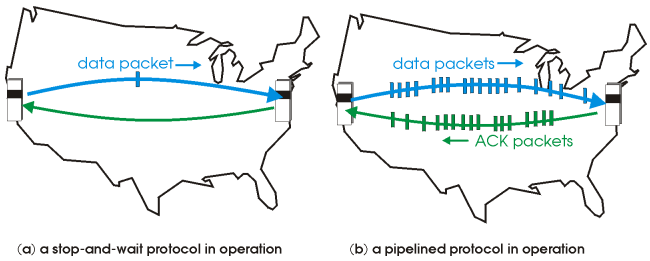
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- Effective thruput of 267 kbps over a 1 Gbps link

29

Pipelined Protocols

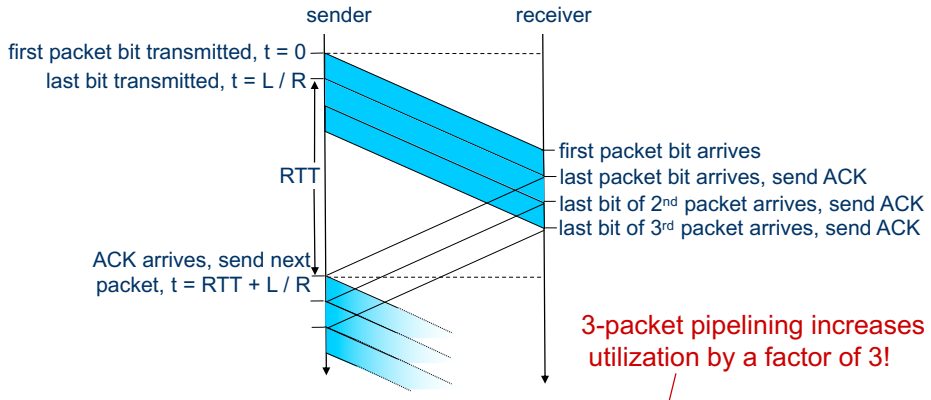
- Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts



- Two generic forms of pipelined protocols
  - Go-Back-N and Selective Repeat

30

Pipelining: Increased Utilization



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

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

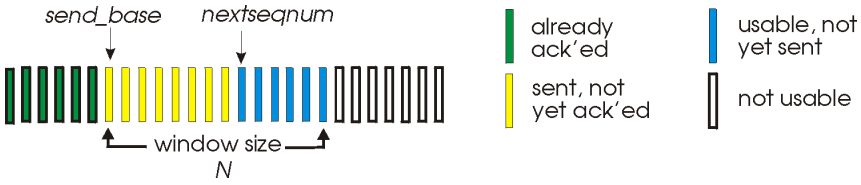
Q: What are the implications of pipelining?

Pipelined Protocols: Overview

- Go-Back-N (GBN)
  - Sender can have up to *N* unacked packets in pipeline
  - Receiver can send cumulative ACKs
  - Sender has timer for oldest unacked packet
- Selective Repeat (SR)
  - Sender can have up to *N* unacked packets in pipeline
  - Sender maintains timer for each unacked packet
    - When timer expires, retransmit only that unacked packet

GBN: Sender

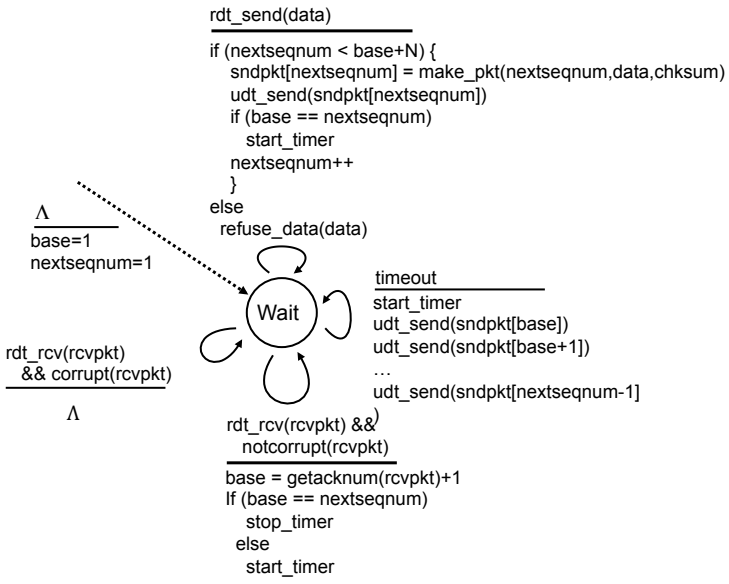
- *k*-bit seq # in pkt header
  - "Window" of up to *N*, consecutive unacked pkts allowed
    - Seq #: [0, $2^k - 1$]



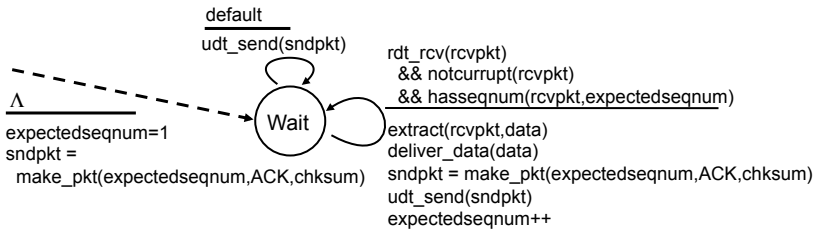
    - Pkts already transmitted: [0, send_base-1]
    - Pkts not yet ACKed: [send_base, nextseqnum-1]
    - Pkts that can be transmitted

- Timer for oldest in-flight pkt
  - Timeout(*n*): Retransmit packet *n* and all higher seq # pkts in window

GBN: Sender Extended FSM

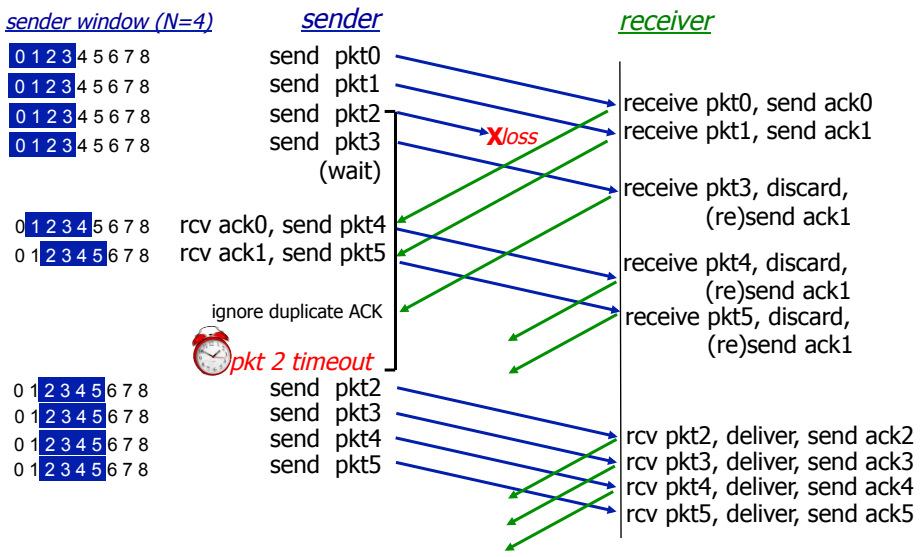


GBN: Receiver Extended FSM



- ACK-only: always send ACK for correctly-received pkt with highest in-order seq #
  - Need only remember **expectedseqnum**
- Out-of-order pkt
  - Discard: no receiver buffering!

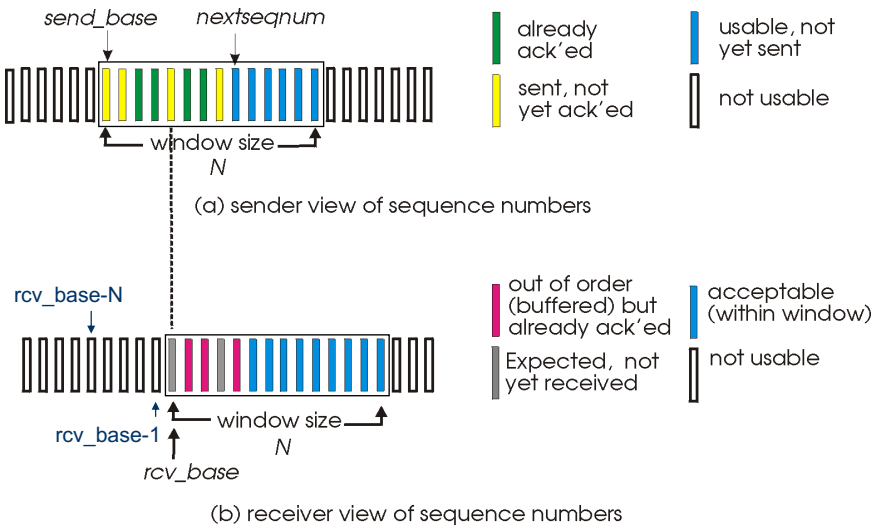
GBN in Action



Selective Repeat

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

SR: Sender & Receiver Windows

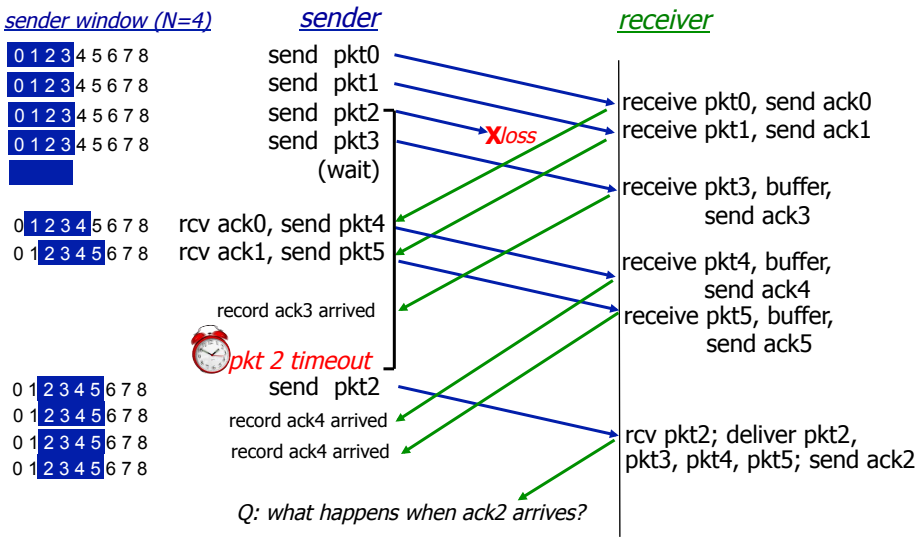


Selective Repeat

- Sender
  - Data from above
    - If next available seq # in window, send pkt
  - Timeout(*n*)
  - ACK(*n*) in [send_base, send_base+*N*]
    - Mark pkt *n* as received
    - If *n* smallest unACKed pkt (seq # == send_base), advance window base to next unACKed seq #
- Receiver
  - Pkt *n* in [rcv_base, rcv_base+*N*-1]
    - Out-of-order: buffer
    - In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
  - Pkt *n* in [rcv_base-*N*, rcv_base-1]
  - Otherwise
    - Ignore

39

Selective Repeat in Action

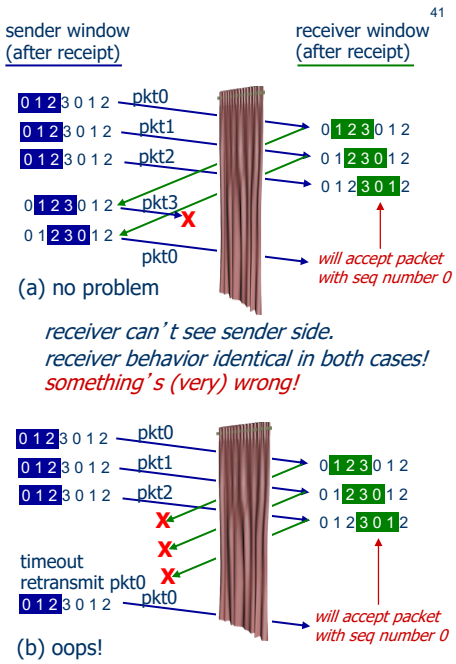


40

41

SR: Dilemma

- Example
  - Seq #'s: 0, 1, 2, 3
  - Window size = 3
- Receiver sees no difference in two scenarios!
- Exercise: What should be the relationship between seq # size and window size to avoid problem in (b)?



41

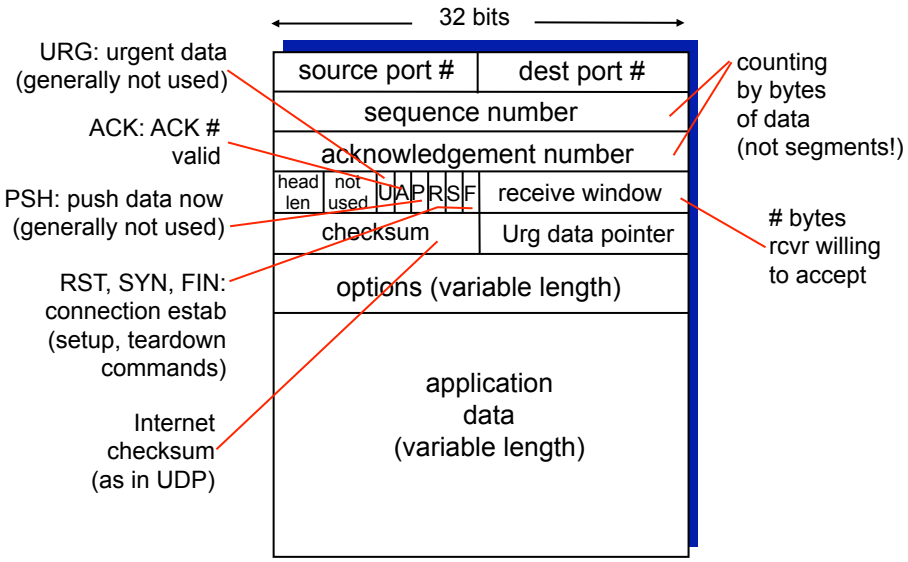
42

TCP: Overview

- Point-to-Point
  - One sender, one receiver
- Reliable, in-order byte steam
- Pipelined
  - TCP congestion and flow control set **window size**
- Full Duplex Data
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- Connection-Oriented
  - Handshaking - exchange of control msgs
    - Initialize sender, receiver state before data exchange
- Flow Controlled

42

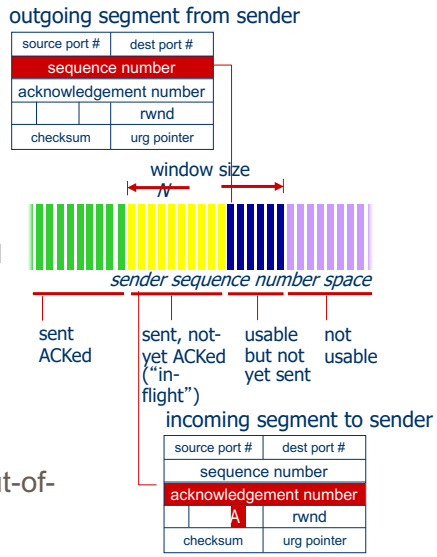
TCP Header



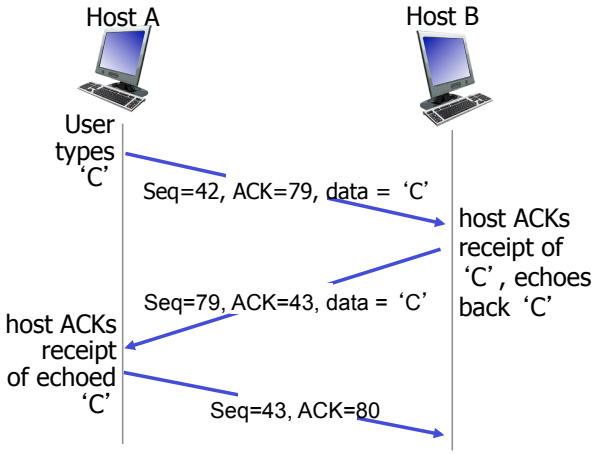
TCP Seq. Numbers & ACKs

- Sequence Numbers
  - Byte stream "number" of first byte in segment's data
- Acknowledgements
  - Seq # of next byte expected from other side

Q: How does receiver handles out-of-order segments GBN or SR?



Piggybacking



simple telnet scenario

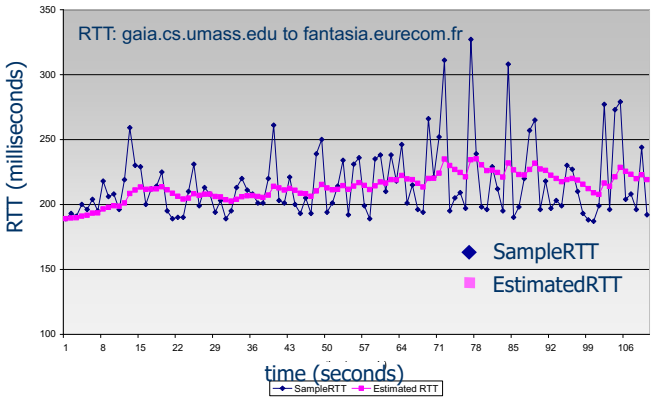
TCP Round-Trip Time (RTT)

- How to set TCP timeout value to recover from lost segments?
  - Longer than RTT - but RTT varies
  - Too short
  - Too long
- How to estimate RTT?
  - SampleRTT: measured time from segment transmission until ACK receipt
    - Ignore retransmissions
  - SampleRTT will vary due to congestion and load at routers
    - Want EstimatedRTT - smoother
      - Average several recent measurements, not just current SampleRTT

Average RTT

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

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



47

Variations in RTT (DevRTT)

- Estimate SampleRTT deviation from EstimatedRTT

DevRTT = (1-β * DevRTT +
β|SampleRTT-EstimatedRTT|
(typically, β= 0.25)

- Timeout Interval: **EstimatedRTT** plus "safety margin"

TimeoutInterval = EstimatedRTT + 4*DevRTT

estimated RTT

"safety margin"

- For TCP initial value of TimeoutInterval = 1 second

48

TCP Reliable Data Transfer

- TCP creates rdt service on top of IP's unreliable service
  - Pipelined Segments
  - Cumulative ACKs
  - Single Retransmission Timer
- Retransmissions triggered by
  - Timeout Events
- Let us initially consider simplified TCP sender
  - Ignore Duplicate ACKs
  - Ignore Flow Control & Congestion Control

49

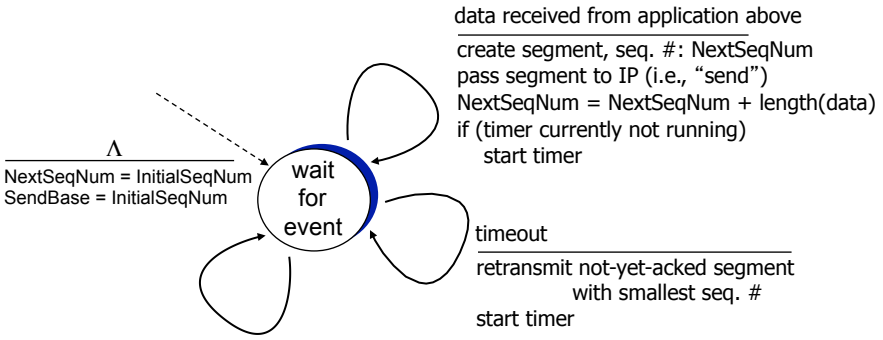
TCP Sender Events

- Data rcvd from app
  - Create segment with seq #
- 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 ACK acknowledges previously unacked segments
    - Update what is known to be ACKed
    - Start timer if there are still unacked segments

50

51

TCP Sender (Simplified)

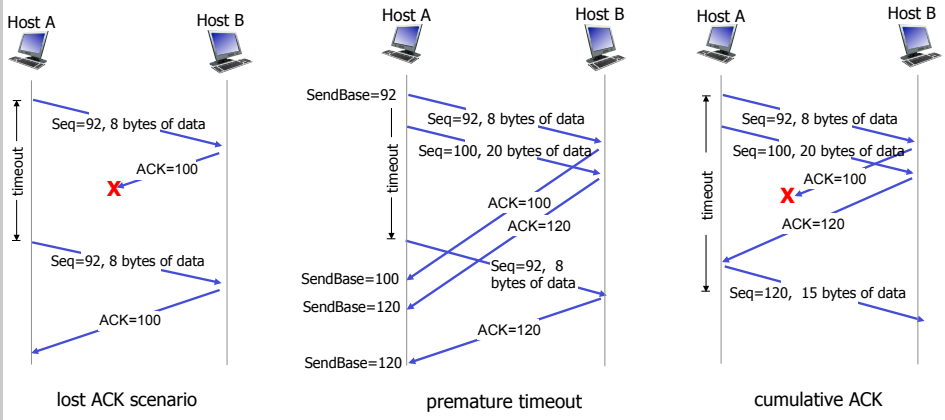



```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
   }
```

51

52

TCP: Retransmission Scenarios



52

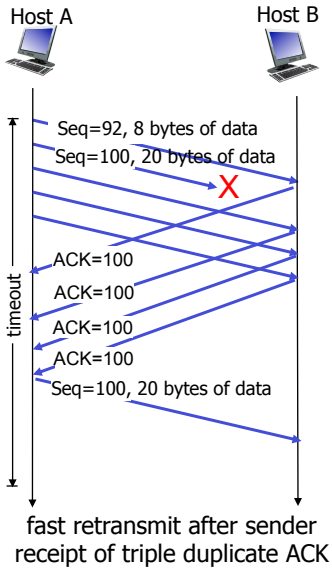
TCP ACK Generation

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
	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
	Immediately send ACK, provided that segment starts at lower end of gap

Q: Is TCP a GBN or SR protocol?

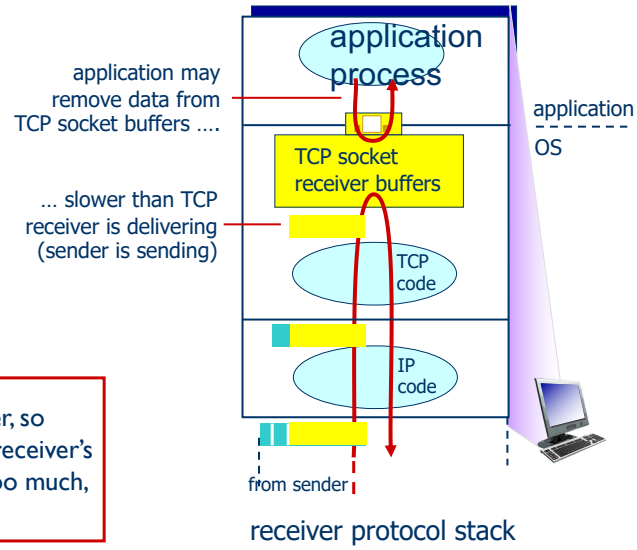
TCP Fast Retransmit

- Time-out period often relatively long
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
- TCP Fast Retransmit
  - If sender receives 3 ACKs for same data - "Triple Duplicate ACKs"
    - Resend unacked segment with smallest seq #
  - Likely that unacked segment lost, so do not wait for timeout



55

TCP Flow Control



flow control

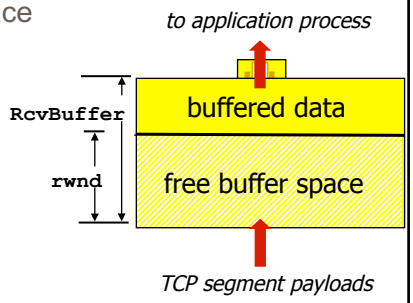
Receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

55

56

TCP Flow Control II

- Receiver "advertises" free buffer space by including *rwnd* (receive window) value in TCP header of receiver-to-sender segments
  - RcvBuffer size set via socket options
  - rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]
- Sender limits amount of unacked ("in-flight") data to receiver's *rwnd* value



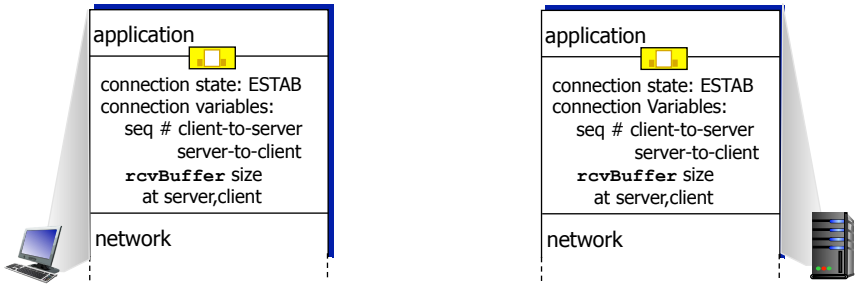
receiver-side buffering

Q: What happens when receiver window is full and it has advertised *rwnd*=0 and has no data to send?

56

Connection Management

- Before exchanging data, sender/receiver "handshake"
  - Agree to establish connection (each knowing the other willing to establish connection)

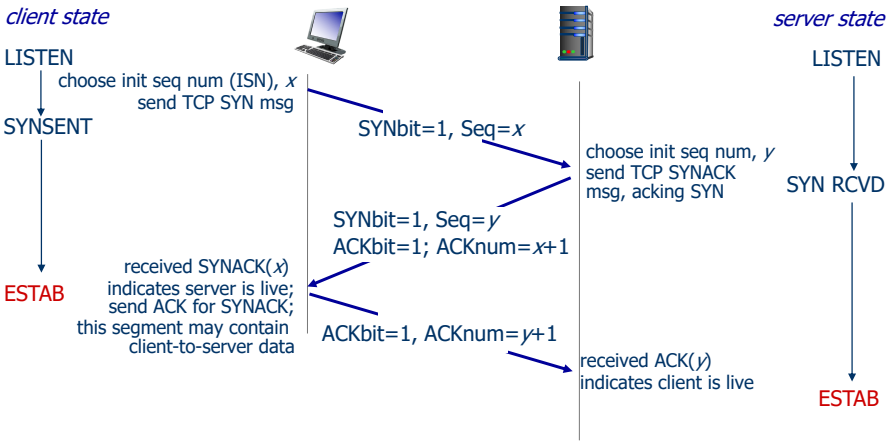



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

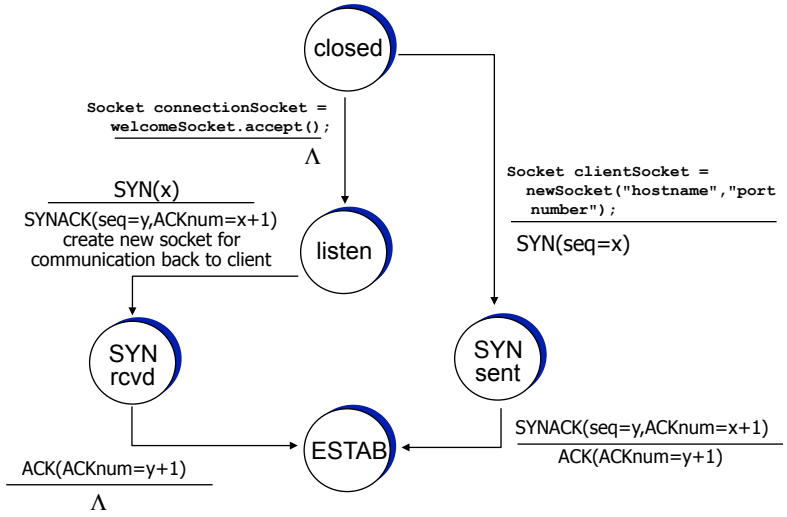
57

TCP: 3-way Handshake

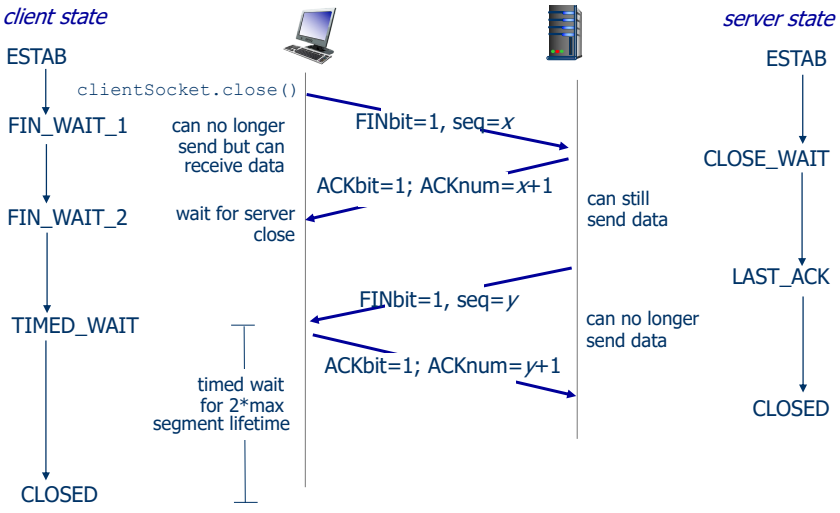


58

TCP: 3-way Handshake FSM

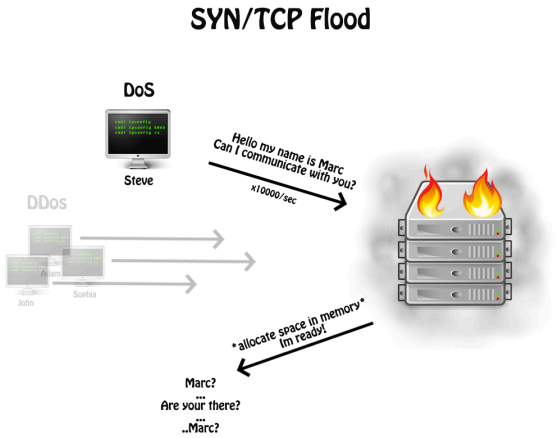


TCP: Closing a Connection



SYN Flood Attack (DoS)

- Attacker sends large number of TCP SYN segments
  - Without completing the third handshake step

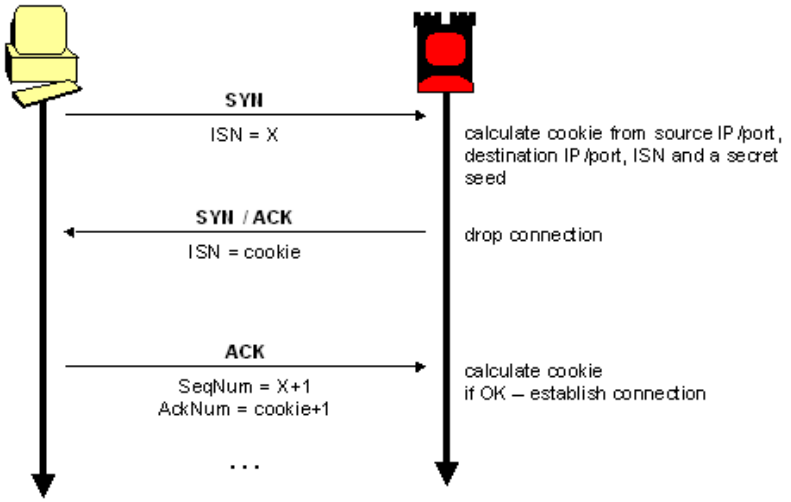


SYN Cookies

- Server does not know if the SYN segment is coming from a legitimate user
- Server creates an initial sequence number (ISN) or “cookie” from the hash of
  - Src IP addr & Port
  - Dest IP addr & Port
  - Secret Seed
- Server sends SYNACK
- A legitimate client will return an ACK segment
  - Use the cookie information (ISN+1) in the ACK

63

SYN Cookies II



63

64

Principles of Congestion Control

- Informally:
  - "Too many sources sending too much data too fast for the *network* to handle"
    - Different from flow control!
- Manifestations
  - Lost packets
  - Long delays

64

65

Approaches Towards Congestion Control

- End-to-End Congestion Control
  - No explicit feedback from network
  - Approach taken by TCP
- Network-Assisted Congestion Control
  - Routers provide feedback to end systems
  - Explicit rate for sender to send at

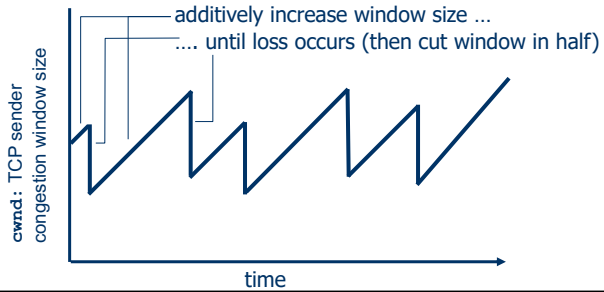
65

66

Additive Increase Multiplicative Decrease (AIMD)

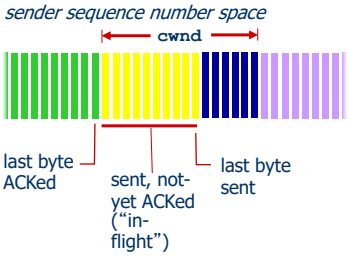
- Sender increases transmission rate (window size)
  - Probing for usable bandwidth, until loss occurs
- Additive Increase
  - Increase congestion window (*cwnd*) by 1 MSS every RTT until loss detected
- Multiplicative Decrease
  -

AIMD saw tooth behavior: probing for bandwidth



66

TCP: Congestion Control Details



- Sender limits transmission

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{cwnd}$$

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

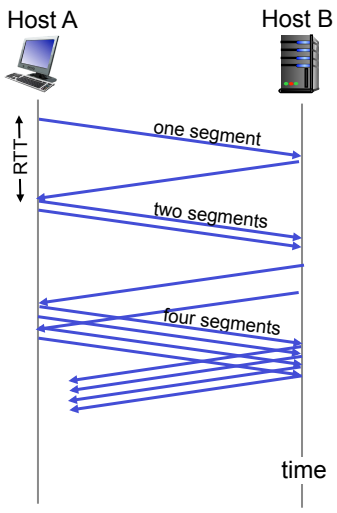
TCP sending rate

- Send *cwnd* bytes
- Wait RTT for ACKs
- Then send more bytes

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

TCP: Slow Start

- When connection begins, increase rate exponentially until first loss event
  - Initially *cwnd* = 1 MSS
  - Done by incrementing *cwnd* for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



69

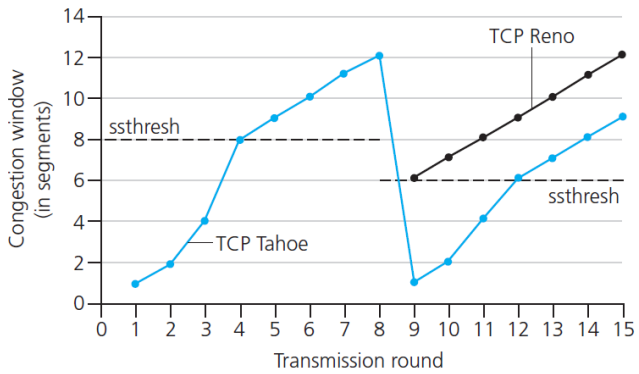
TCP: Congestion Avoidance

- Loss indicated by timeout
- Window then grows exponentially (as in Slow Start) to threshold (*ssthresh*)
  - Then grows linearly
- Loss indicated by 3 duplicate ACKs: TCP Reno
  - Dup ACKs indicate network capable of delivering some segments
- TCP Tahoe always sets *cwnd* to 1
  - Timeout or 3 duplicate ACKs

69

70

TCP: Congestion Avoidance



Q: When should the exponential increase switch to linear?

70

TCP: Congestion Control Summary