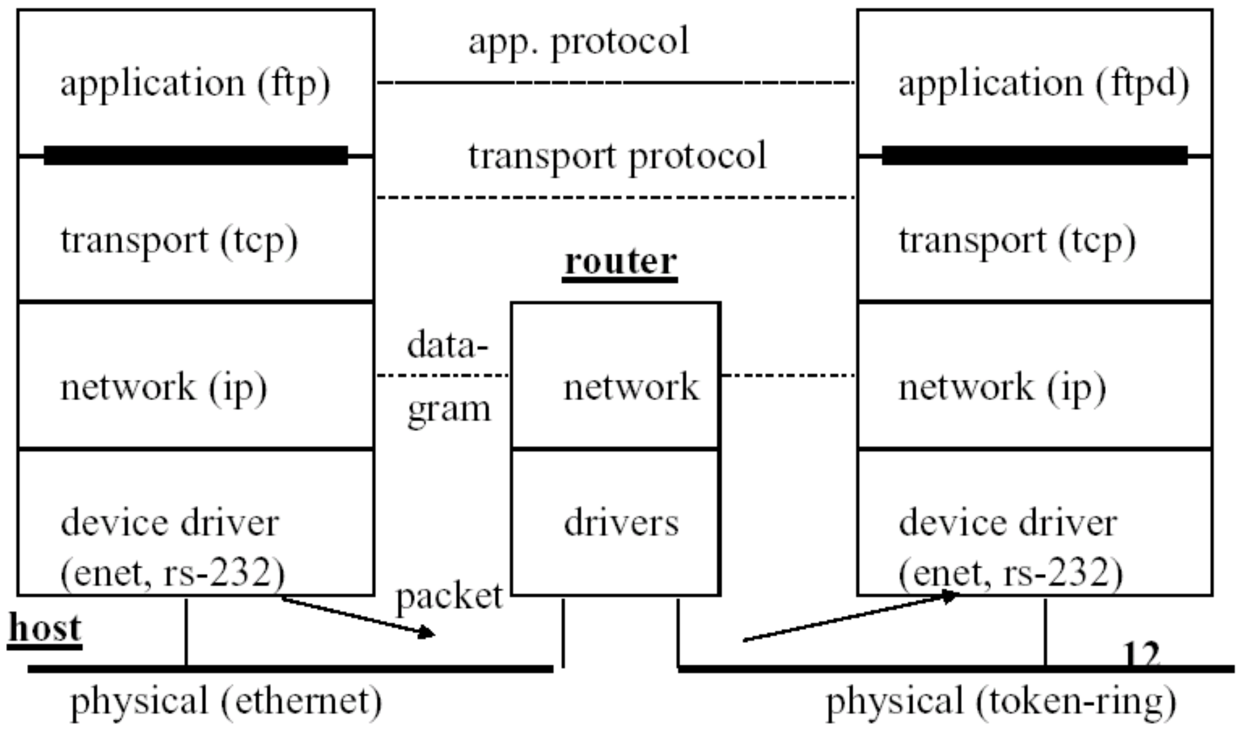
Transport layer

Outline

- UDP
- TCP
  - Header
  - Opening and closing connections
  - Some TCP protocol mechanisms
    - Flow control
    - Timers and retransmission
    - Congestion control
  - TCP performance
  - Conclusions

TCP/IP Reference Model



UDP

- UDP - User Datagram Protocol
- RFC 768
- UDP == “ip with ports”
- client/server both “bind” to a port and send and receive messages via a port
- port is 0..64k-1
- well-known ports associated with servers

UDP contd.

- UDP provides unreliable connectionless delivery
  - no error or flow control
- there is a checksum, but it is configured on/off per host
- checksum is over ip pseudo header, udp header, and data
- if 0 value is sent, means checksum off

UDP Encapsulation

ethernet hdr	ip header	udp header	data

8 bytes (no options)

UDP Header

0 … 15	16 … 31
src port #: 16 bits	dest port #:16 bits
UDP length: 16 bits	checksum: 16 bits
data ...	

length includes header, minimum is 8

Pseudo Header

- udp (and tcp) optionally performs checksum across whole payload, with ‘ip pseudo-header’
  - 32 bit source ip address
  - 32 bit dest. ip address
  - 1 byte zero
  - 1 byte proto = 17 (UDP)
  - 2 bytes UDP length == 12 bytes in all
- original idea was to include IP addresses (etc) into checksum as a form of authentication (little used)

Applications that use UDP

- regular, broadcast oriented
  - routing daemons (rip and routed)
- streaming apps
  - do not want tcp error and flow control
- multicast apps (eg audio conferencing)
  - tcp can’t handle, so udp only alternative
- short message-oriented (don’t want connection overhead)
  - snmp, dns
- applications must use own recovery mechanisms as appropriate

TCP Introduction

- TCP - Transmission Control Protocol
- reliable, connection-oriented stream protocol (UDP is not)
- delivers to the receiver the exact stream sent
- required because underlying network (IP) is imperfect
  - loss, out of order due to routing, or corruption
  - TCP service makes it look reliable

Introduction

- RFC 793 and host requirements RFC 1122
- TCP has own jargon:
  - **segment**: a TCP unit of transfer
  - **MSS: maximum segment size**: max segment one TCP side can send another, negotiated at connection time (or default)
  - **ports**: for identifying end-points
  - **socket:** for identifying connections

TCP Properties

- stream orientation: stream of octets (bytes) passed between sender and receiver
- byte stream is full duplex: two streams
  - two independent streams joined by **piggybacking**
- piggybacking: one data stream carries control info for the other data stream (going the other way)
- unstructured stream
  - TCP arbitrarily divides streams into segments
  - doesn't show segment boundaries to applications

TCP Properties

- unstructured stream
  - but you can still structure your i/o calls as "messages" or structures if you want
- connection oriented
  - connect – data transfer - disconnect
  - client connects and server listens/accepts
- TCP provides flow control
  - receiver 'paces' the sender so cannot be overwhelmed

TCP Properties

- **error and loss handling is in TCP**
  - end to end (critical TCP function)
- congestion detection end to end
  - backs off if it thinks net is congested
- complex protocol
  - can treat telnet (interactive) and ftp (bulk transfer) differently + acks/timers, etc.

TCP Sliding Windows

- TCP uses *byte count* (not packet count) for sequencing: each byte in the stream has it's own sequence number
- For example, using 20-byte segments:

  First segment – sequence number 0

  Second segment – sequence number 20

  Third segment – sequence number 40

  Etc.

TCP Sliding Windows

- receiver controls sliding window size
  - window ‘advertisement’ in ACK packets
  - sender adjusts window size accordingly
- can stop all sending by advertising window size of 0
  - **flow control**

Flow Control

- flow control occurs because the receive side controls the window size
- if window advert = 0, the sender cannot send data
  - re-opened by a subsequent ACK, but:
  - sender will send window probe (1 byte of data) to see if window is open (ack might be lost). (Separate timer for this function called 'persistence timer')
- sender can also control flow rate via its window size
- this is *end to end* flow control

TCP Encapsulation

e.2=14	20	20	1460
ethernet	ip header	tcp header	data (but maybe not)

TCP header may have options, but default size is
20 bytes

TCP Segment Header

<table>
<tr><td colspan="8">0 … 15</td><td colspan="2">16 … 31</td></tr>
<tr><td colspan="8">source port: 16 bits</td><td colspan="2">dest. port: 16 bits</td></tr>
<tr><td colspan="10">sequence number: 32 bits</td></tr>
<tr><td colspan="10">acknowledge number: 32 bits</td></tr>
<tr><td>hlen:4</td><td>resv:6</td><td>U</td><td>A</td><td>P</td><td>R</td><td>S</td><td>F</td><td colspan="2">window size: 16</td></tr>
<tr><td colspan="8">TCP checksum:16</td><td colspan="2">urgent pointer: 16</td></tr>
<tr><td colspan="10">TCP options (if any)</td></tr>
</table>

Header Explained

- header sent with every TCP segment
- segment may be just a header: a control message (SYN/FIN/ACK) with no data
- view TCP as 2 send/receive data streams with control information (eg ACK) sent back the other way (piggybacking)

Header

- source port: 16 bits, the TCP source port
- destination port: 16 bits
- sequence number: 1st data octet in this segment (from send to recv): 32 bit space
- ack: if ACK flag set, next expected sequence number (piggybacking; i.e., we are talking about the flow the other way)

Header

- hlen: number of 32 bit words in header (usually 5)
- reserved: not used
- flags
  - URG: - urgent pointer field significant
  - ACK:- ack field significant (this pkt is an ACK!)
  - PSH: - push function (send now!)
  - RST: - reset (give up on) the connection (error)
  - SYN: - initial synchronization packet (start connect)
  - FIN: - final hang-up packet (end connect)

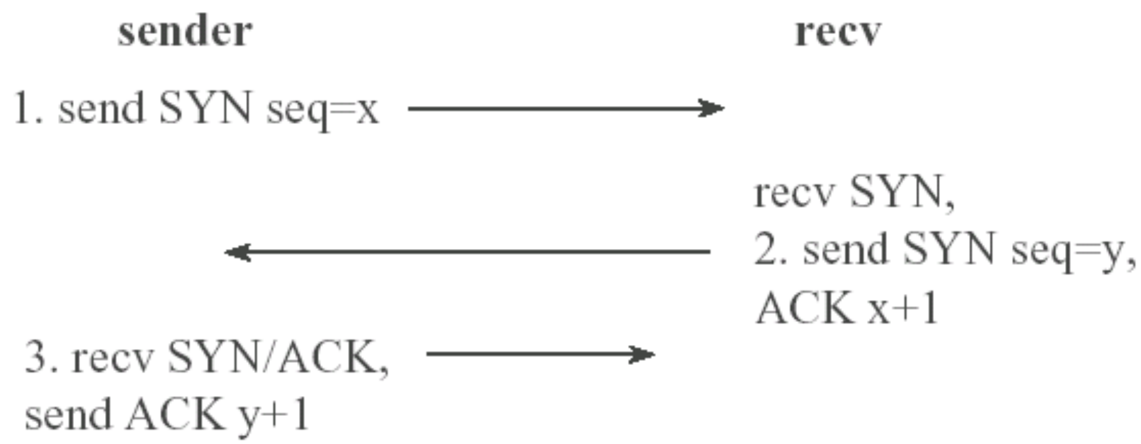
Header

- window: advertises size of window that recv-side will accept (flow control)
- checksum: 16 bits, pseudo-header, tcp header, and data
- urgent pointer: offset from sequence number, points to data following urgent data (URG flag must be set)
- options - e.g., Max Segment Size (MSS)

TCP Open / Close

- TCP distinguishes **passive** and **active** open
- servers usually do passive open, means they LISTEN
- clients usually do active open, means they connect
- reach ESTABLISHED state after 3-way handshake

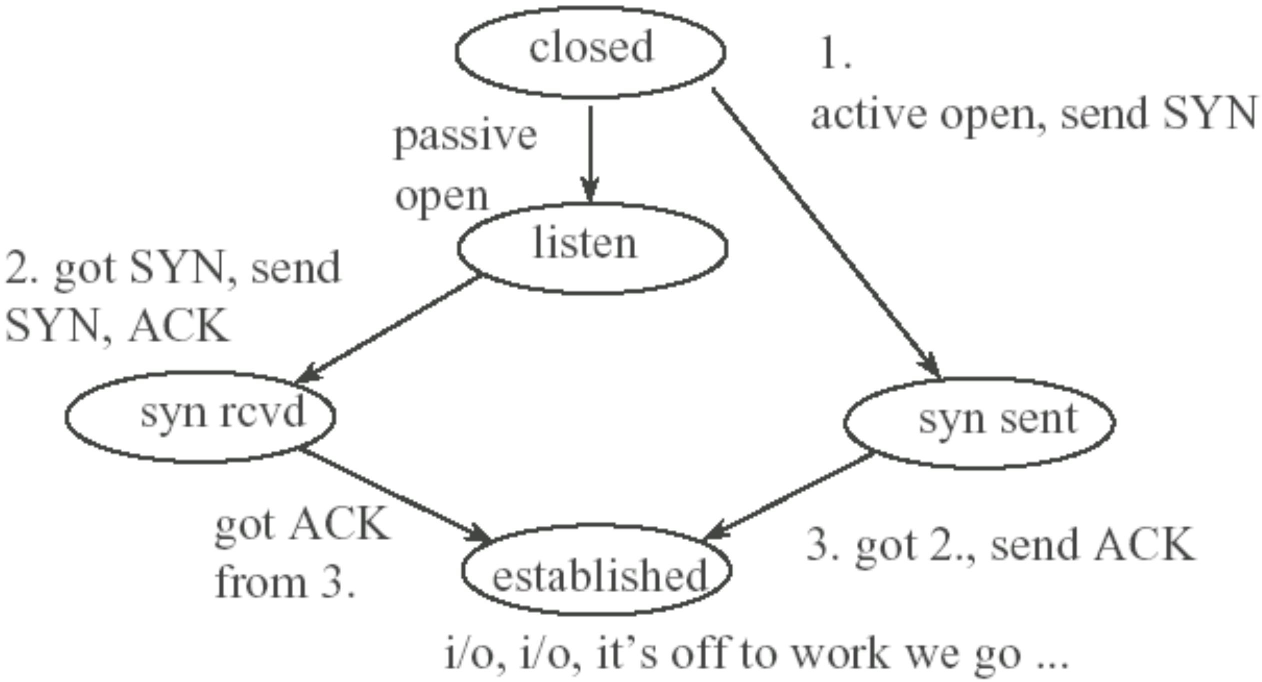
Three Way Handshake



both sides can SYN at the same time and it will work

results include established connection, initial sequence numbers exchanged, ACKS ack next expected byte (cumulative)

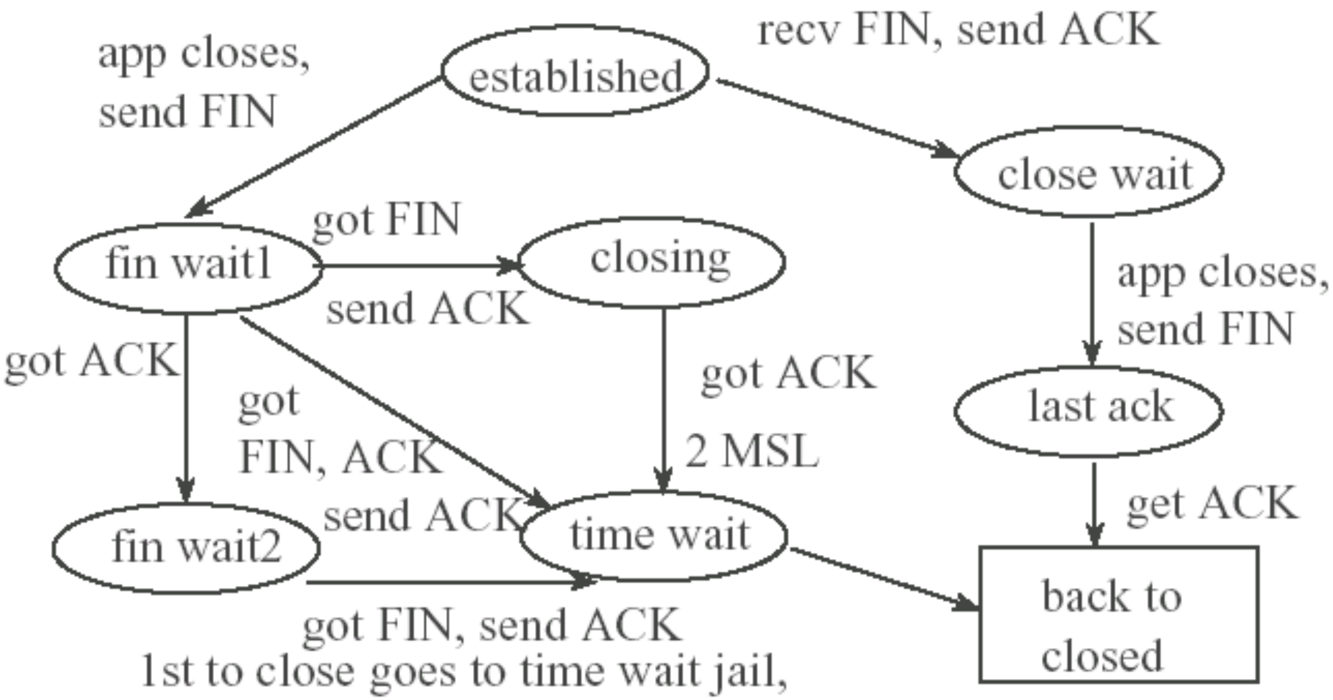
Open State Machine



Closing TCP Connection

- connections are full duplex and it is possible to shutdown one side at a time
  - initiated when application does ‘close’
- really just two 2-way handshakes (send FIN, recv replies with ACK per channel)
- open/close handshakes withstand most loss and duplicate scenarios, but
- interesting problem: how do you make sure last ACK got there (can’t ACK it...)

Close State Machine



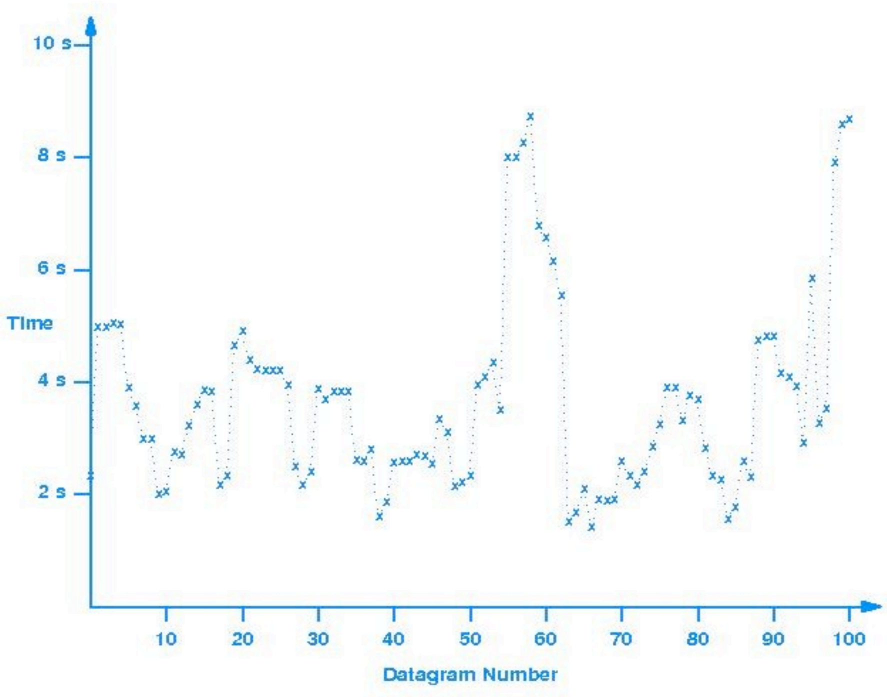
Timeout and Retransmission

- Round Trip Times (RTTs) on the Internet can be highly variable during a session
- So, how to decide a timeout period value?
- if too long, response not good if timeout occurs
- if too short, spurious retransmissions
  - (and increased congestion, which increases RTT)
- TCP uses an **adaptive retransmission algorithm**

Plot of RTTs for 100 Datagrams

Nowadays delay is lower.

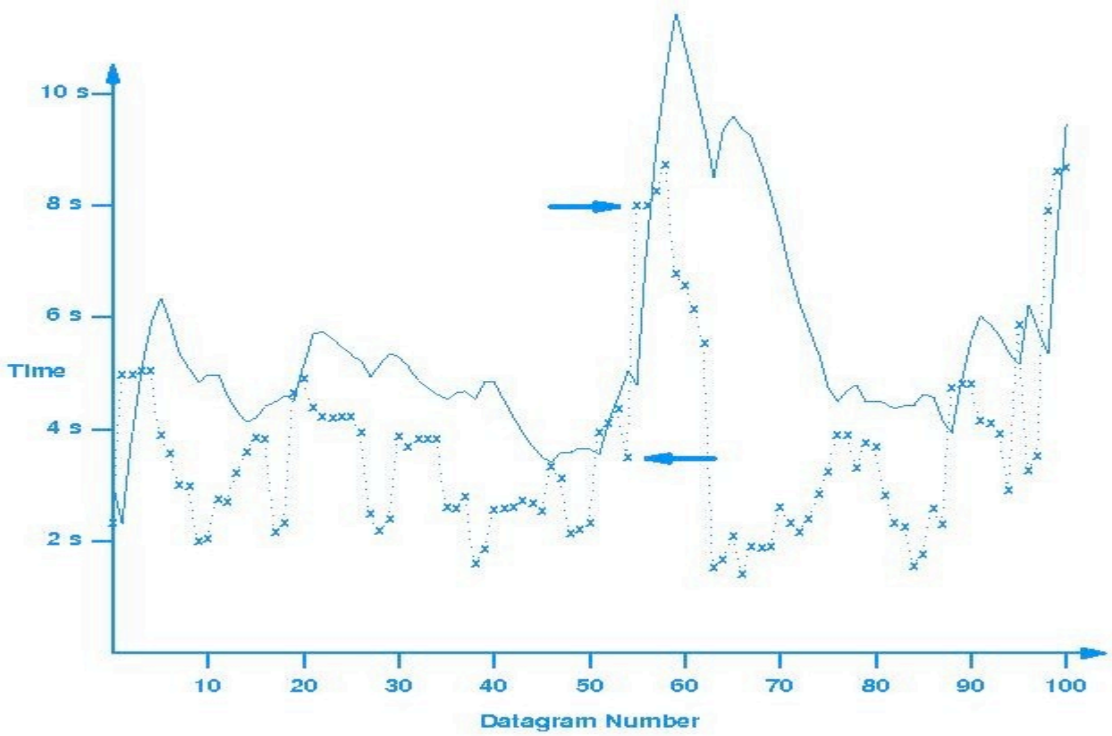
But still same variance



Adaptive Timeout and Retransmission

- Start with a default value of TO
- Update TO using ACKs to measure Sample RTTs
  - Difference: time ACK x received vs segment x sent
- Simple Algorithm:
  - First measure RTT (SRTT = Sample RTT))
  - Estimate mean RTT (ERTT) through
    - ERTT = (1-a)*Old ERTT + a*SRTT   (0=<a<1)
  - Recompute timeout period
    - TO = z*ERTT   (z originally = 2)
  - (Choice of 'a' determines responsiveness to change)

TCP Retransmission Timer Estimates vs Plotted RTTs



Adaptive Timeout and Retransmission

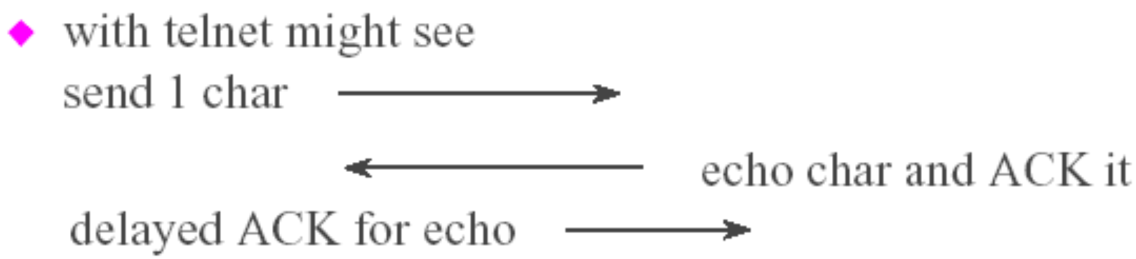
- Does not cope well with high levels of RTT variance (common in the Internet)
- Complex Algorithm:
  - As above for SRTT and ERTT
  - **Estimate Deviation (DRTT) as**
    - **DRTT= (1-b)*DRTT + b*(SRTT-ERTT)**
  - **TimeOut=ERTT+4*DRTT**

Timer Backoff (Karn's Algorithm)

- acks can be *ambiguous* with regard to RTT estimates due to retransmitted segments
- only use *unambiguous* acks for RTT
  - where segment has been transmitted once only
- if no acks at all are received tcp will use a modified form of exponential backoff of the timeout period
- if ack packets start showing up, backoff is removed

Delayed Acks

- don’t send ACK immediately, wait for reverse flow data to show up so that ACK can piggyback (free ride – no need for an extra TCP segment)
- but don’t wait for ever: delay typically is 200 ms (max 500ms)
- this is a receiver-side timer



Nagle Algorithm

- traditional telnet inefficient: we have 40 bytes of header for 1 echoed byte of data
- rfc 896: Nagle algorithm = delayed transmit
- while waiting for an ACK, can accumulate more data in sender buffer before sending it
  - data is ‘clumped’ before sending
- accumulated data is sent when, either
  - an ACK is received, or
  - maximum segment size is accumulated

Nagle contd.

- only affects sender who sends small data amounts
  - fast senders keep the buffer full
- algorithm is said to be “self-clocking”: you can go as fast as round trip latency will allow since you wait for return ACK
- some applications do not want nagle algorithm, they want to send small data chunks (eg mouse clicks) immediately
  - TCP_NODELAY socket option turns this off
  - uses PUSH flag in TCP header

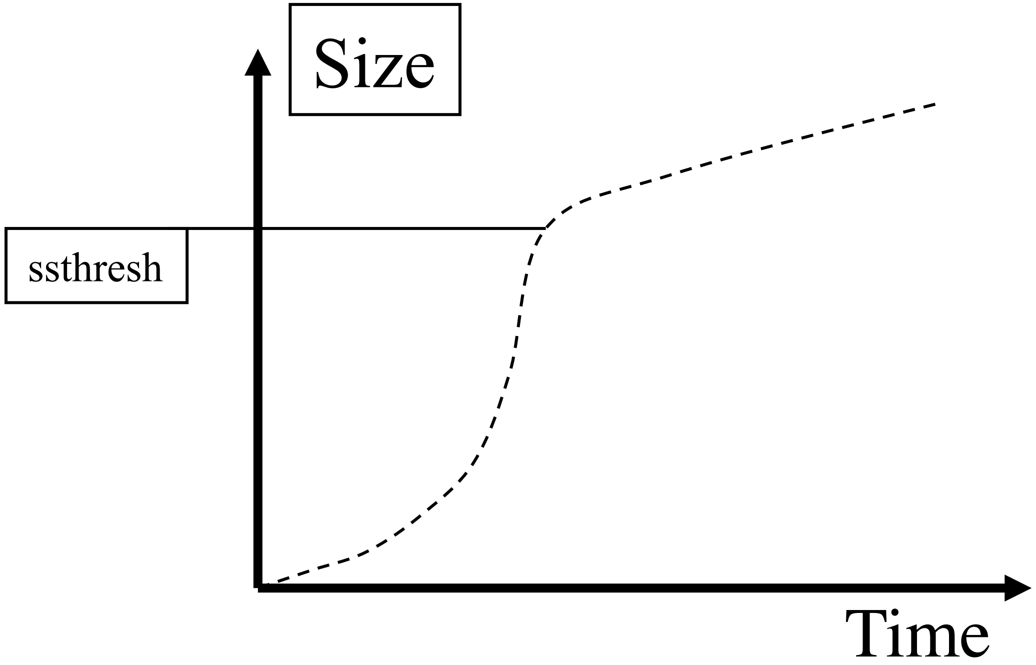
Congestion Control

- routers may drop packets as buffers run out = congestion
- routers don't have effective mechanisms to indicate congestion to sender
- sending apps send as fast as they can
  - early Internet collapse
- TCP now assumes packet loss to be due to congestion (assumes packet damage to be rare)

Congestion Avoidance

- TCP uses **slow start** and **multiplicative decrease** to deal with congestion
  – Van Jacobson (1988) outlined these ideas
- slow-start roughly: whenever starting a connection or recovering from congestion,
  – start congestion window at the size of 1 or a few segments
  – increase window size by one with each ACK (**additive increase**)
  – At a threshold, change to linear increase and open up by 1/window segments per ack. (congestion avoidance phase)
  – often called open up by one segment / round

TCP Congestion COntrol



Congestion Control

- multiplicative decrease - upon loss of a segment,
  – reduce the congestion window by half
  – down to a minimum of one segment.
- For those segments that remain in the send window, backoff the retransmission timer exponentially because of congestion.

Retransmissions

- TCP uses both ‘Go back-N’ and selective repeat for retransmissions
- If timeout: Go Back-N and go into slow start (go back to the segment that caused the timeout and retransmit from there)
- Improvement: if three repeat acks (same sequence number) then selective repeat
  - Infers that a segment is lost, so
  - fast retransmit, before a timeout occurs

Routers and Congestion RED

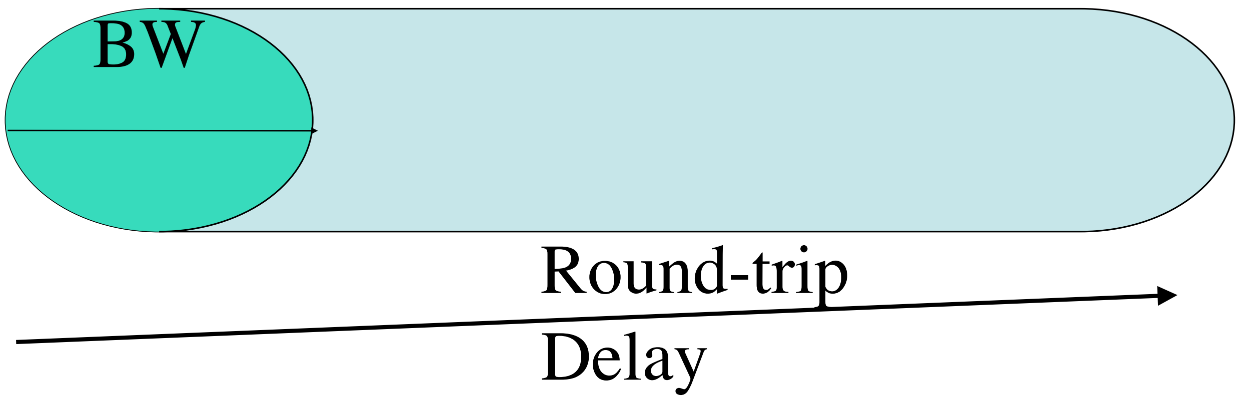
- routers might use a simple queue-drop mechanism
  - Buffers all full: drop packets at end of queue, call this a "**tail-drop**" policy
  - on heavily multiplexed router TCP connections may lose more than one packet and be forced into slow-start
- routers may use **Random Early Detection** (or RED) – basically: randomly discard packets in queue at a certain saturation point
  - thus avoid tail-drop synchronisation

TCP Performance

- mid 80’s: performance limited by speed of sending hosts
- now can achieve good utilisation
- eg 2 hosts on an ethernet can get about 94% utilisation
  - assuming zero propagation delay
- but window size (buffer size) can be the bottleneck

TCP Performance

- Bandwidth/Delay product determines optimal buffer size
- Buffer size too small => throughput constraints



TCP Performance

- Simple example: determine effective throughput and Utilisation of a pipe,
  - Raw BW = 10 Mbps, RTT = 1000 ms, Window = 64 kbyte (maximum)
- Data transmitted before blocked = 64k*8 = 512 kbit
- Unblock in 1s: effective throughput = 512k/1 = 0.512Mbps
- Pipe capacity (BW/delay product) = 10Mbps*1 = 10 Mbit
- U = 0.512 / 10 = 5.1%

TCP over Wireless

- TCP still needs to evolve!
- Wired: loss assumed due to congestion
- Wireless: loss often due to radio characteristics
- TCP makes wrong assumption
- Reacts wrongly – unnecessary backoff
  - Further optimisations have been proposed