Corrections made on 2/26/98 are listed in purple; 3/18 in green.

CS/EE 5516 -- Computer Networks
Lecture 10: TCP Protocol
Reference: Comer, *Internetworking with TCP/IP, 2nd. Edition, Vol. I*, Ch. 12

Comparison of IP, UDP, and TCP:

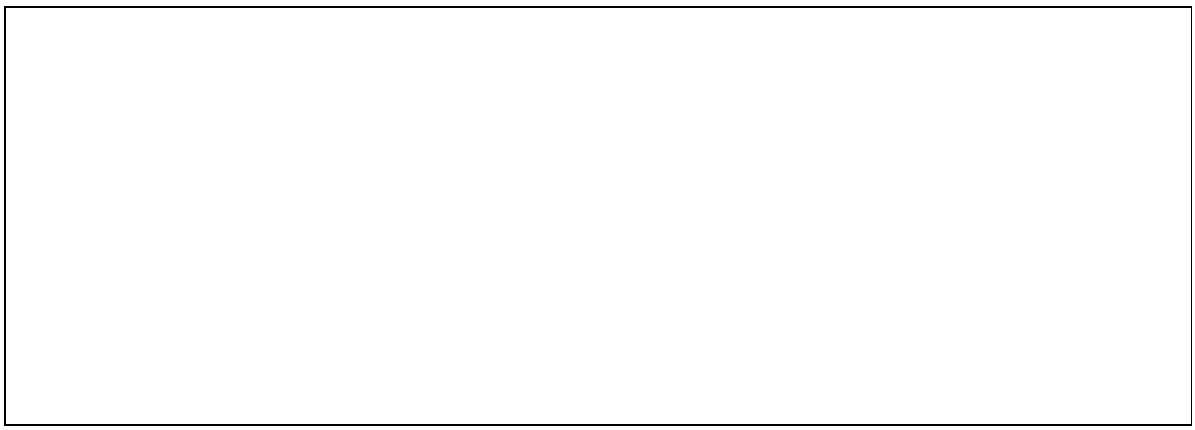
Stevens Fig 5.5:

	IP	**UDP**	**TCP**
connection-oriented?	no	no	yes
message boundaries?	yes	yes	no
data checksum?	no	opt.	yes
positive ack?	no	no	yes
timeout and rexmit?	no	no	yes
duplicate detection?	no	no	yes
sequencing?	no	no	yes
flow control?	no	no	yes

TCP differs from go-back-n with balanced link initialization protocol as follows:

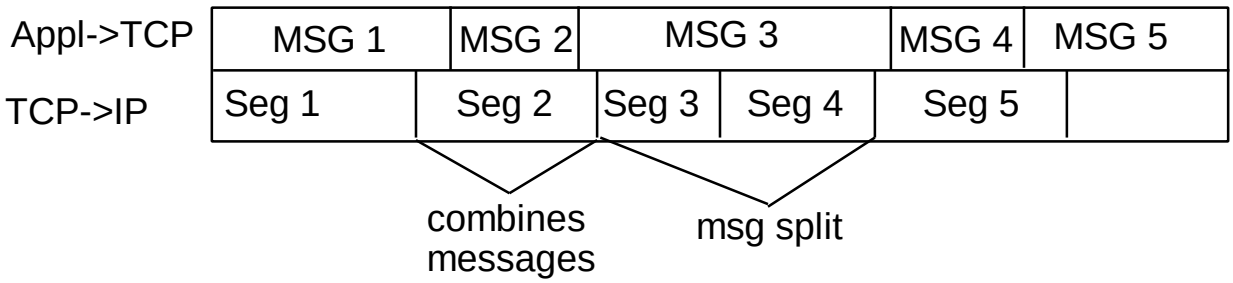
1. n varies
2. retransmission time value varies
3. sequence numbers refer to bytes in a message
4. a message of arbitrary length is fragmented into segments (receiving TCP does *not* reassemble)
5. TCP performs congestion control
6. There is exactly one packet type used for all transfers: data, acks, init, and disc
7. Two traffic types: normal and urgent data. Example of urgent data: C in Unix

Terminology:



How TCP partitions a message into segments:

Segments, Streams, Sequence #'s:



MSS (maximum segment size) is usually no larger than the MTU-2*20. (The term 2*20 is for the TCP and IP headers.) For Ethernet, MSS=1500-2*20 = 1460.

TCP header format (Comer Fig. 12.7)

- 20 byte header if OPTIONS not used (So 1500-20-20=1460 is MSS for ethernet)
- There are no separate ACK/DATA segments. TCP normally does not generate an ACK for received data. Thus ACK is *piggybacked* on DATA.
  - Done simply by ACKNUM field in every TCP header. This is the number of the octet that the source expects to receive next (in other words, one more than the largest, contiguous byte number received).
  - When TCP receives incoming segment, it waits for outgoing data, and piggybacks ACK. If no outgoing data for a while, TCP will generate a zero-data length outgoing segment in which to piggyback the ACK.
- HLEN (header length in 32 bit words) is required due to variable length header
- Code bits (they help distinguish data/ack/init/disc packets):
  - URG urgent pointer field valid
  - ACK ack field is valid
  - PSH this sesgment requests a push
  - RST reset connection
  - SYN synchronize sequence numbers
  - FIN sender has reached end of byte stream
- Note: no special control segments used to establish, release connection; use ACK, RST, SYN, FIN bits in normal segment
  - Finite state machine used for connection est/release (Comer Fig. 12.13)
- WINDOW is receiver advertisement
- URGENT POINTER - specifies position in data where urgent data ENDS, if URG bit is set

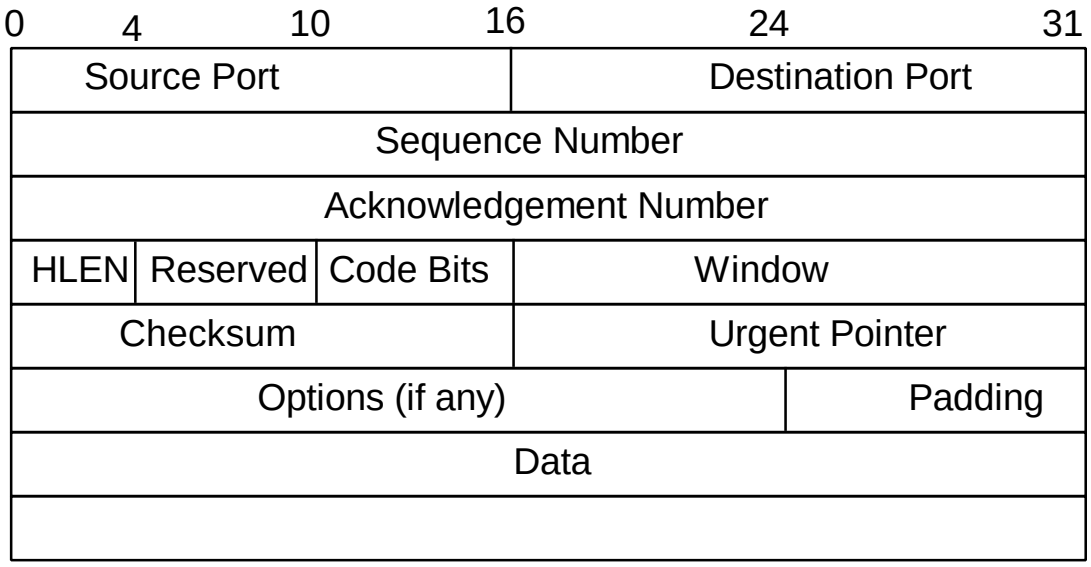


Figure 12.7 Format of TCP header, followed by data. Segments are used to establish connections as well as carry data and acknowledgements

Bit (left to right)	**Meaning if bit set to 1**
URG	Urgent pointer field is valid
ACK	Acknowledgement field is valid
PSH	This segment requests a push
RST	Reset the connection
SYN	Synchronize sequence numbers
FIN	Sender has reached end of its byte stream

(Figure 12.8)

Variable Window Size (n in Go Back-n) (Comer 12.10)

TCP window solves 3 problems (vs. Comer's 2 reasons)

1. Throttles fast sender
2. Provides efficient transmission
3. Provides congestion management mechanism

Congestion = intermediate hops are saturated with traffic

- Good TCP implementation help reduce congestion

- Poor TCP implementation can introduce packets to subnet during congestion period and cause internet breakdown.

What does the "window advertisement" in the TCP header mean?

- The i-th ack sent contains:
  - RN_i = acknowledgement: octet that receiver next expects
  - W_i = *window advertisement*: receiver's current free buffer size
    Initially: W_0 = receiver's buffer size, in bytes

- Each time an ack is sent:

$$W_i = \begin{cases} W_{i-1} & \text{if } RN_i = RN_{i-1} \\ \max(\text{current free buffer space}, W_{i-1} - (RN_i - RN_{i-1}) & \text{if } RN_i > RN_{i-1} \end{cases}$$

  Thus the receiver does *not* contradict previous advertisements
  (e.g., reduce the sender's upper window edge)

- Sender sets its upper window edge to a value <= $RN_i + W_i$
  - Thus sender sets *n* in go-back-*n* to a value <= W_i
  - The "<" is due to congestion management, explained later.

- Sender only increases its upper window edge if receiver chooses $W_l - W_{l-1} > RN_l - RN_{l-1}$.

Example (illustration of receiving TCP)

Suppose that …

- the original window advertisement when connection opened was 8
- the application on the receiving host does *not* remove any data from receiver TCP

Step i-1: Receiver is waiting for octets 4, 6, and 7.
Thus:

- RN_{i-1} = 4
- W_{i-1} = 4

0	1	2	3		5		

Step i: Receiver gets octet 4.
Thus:

- RN_i = 6
- W_i = 2

0	1	2	3	4	5		

Alternate step i:
If the receiver's layer 5 removed bytes 0 and 1, then

- W_i = 4

Q. Receive can advertise a window size of zero to stop sender. Can you think of any exceptions to this rule that are allowed?

A.

- Sender can periodically try sending data in case subsequent non-zero advertisement was lost.
- Sender can send data with urgent flag set to inform receiver that urgent data is available.

Q: Why is $RN_i + W_i$ monotonically increasing?

A:

- RN_i obviously is monotonically increasing.
- W_i is not obviously monotonically increasing. However,W_i can decrease by at most the amount RN_i increases

TCP ACK and Retransmission (Comer [12.15])

Recall:

- RN = next octet expected by receiver
- SN, RN header fields refer to octet # in stream, not a segment number

Q. Why does TCP use octet number, not segment number, for RN & SN?

A. TCP spec allows retrasmitted segment to include more data than original copy! (Perhaps retransmitted packet did not originally contain a full frame's worth of data, and at time of retransmission, sender's layer 5 passed down more data.)

Timeouts and Retransmission *(Comer [12.16])*

1) Unlike DLC, TCP is used over various delay/BW networks. There's no *a priori* knowledge of a "good" timeout value.
2) Unlike DLC, congestion requires dynamic changes to retransmission timeout (TO) value
3) Every connection has its own TO value [Fig 12.10].

Q. Why does every connection have its own TO value?

A. Two different connections may be between two different hosts in the Internet, and thus round trip time (RTT) is probably different.

TCP's Adaptive Retransmission Algorithm

- TCP monitors each connection, and deduces reasonable TO value

- TO = ß * RTT (ß = 2 in early TCP spec)

- RTT_i is estimated round trip time of a segment, after segment *i* was ack'd.
  Each time ack is received:

  $$RTT_i = \alpha * RTT_{i-1} + (1 - \alpha) * RTT_{last\_seg} \qquad (1)$$

  $$a \begin{cases} \text{near } 1 \Rightarrow \text{immune to single segment with long delay} \\ \text{near } 0 \Rightarrow \text{RTTi tracks changes to RTTlast - set rapidly} \end{cases}$$

- Typical values [Karn & Partridge 1987]:
  - $\alpha = 0.875$
  - $\beta = 2$

Alternatives to pre-1987 TCP's α, ß values

1) if you see a RTT_{last_set} that's bigger than your estimate RTT_i, switch to a *smaller* α to adapt more quickly to development of congestion. (Idea due to [Mills].)

2) Vary ß based on observed variance in RTT_{last_seg}. (Due to [Jacobson] - more later.)

Why choosing ß is hard [Karns and Partridge 87]

- Must balance conflict between:
  - Individual user throughput
    - A small β (β slightly larger than 1) detects packet loss quickly.
  - Overall network efficiency (
    - A large β avoids unnecessary retransmissions.
    - Ideally, choose β so that TO is an *upper bound* on RTTlast_seg

- A bad choice of RTT is the *median* of RTT samples:

  Then 1/2 of all packets will be timed out and retransmitted, thereby increasing network load.

- Mills says RTTlast_seg has Poisson distribution, but with brief periods of high delay.

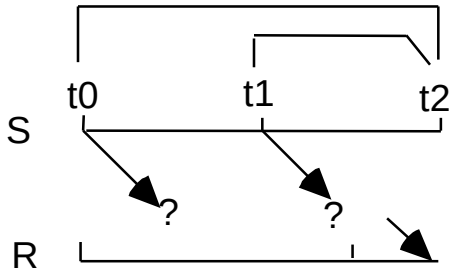
Accurate Measurement of RTT samples ([Comer 12.17])

1) Why can't you just subtract time segment is sent from time ack is received to compute RTT?

A. If loss occurs, there is no longer a 1:1 correspondence between sent segments and acks. This is *acknowledgement ambiguity*.

Example:

1) Sender transmits segment at time t0.
2) Timeout pops.
3) Sender re-transmits segment at t1.
4) Sender receives ack for a segment at t2.
   Is ack for segment sent at t0 or t1?



Should RTT be

a) t2 - t0
b) t2 - t1

Problem with (a):

- Could cause RTT_i to grow w/o bound:
  - You send first segment at t0.
  - Datagram containing segment is lost.
  - You send second segment at t1.
  - You get ack for second segment at t2.
  - You use t2-t0 as RTT sample. That's too long.
  - Now you send *second* segment at t3.
  - Datagram containing second segment is lost.
  - You now wait a long time before retransmitting – namely for t2-t0.

- You get ack at t4. You now use t4-t3 >> t2-t0 as sample.
- Continuing like this, RTT grows without bound!

Problem with (b):

- RTT estimate is too small if a loss did occur; timer will pop too early in future
  - You send first segment at t0.
  - Timer pops and you retransmit at t1.
  - Suddenly the ack for datagram sent at t0 arrives at time t2; you use t2-t1 (which could be nearly 0!) as RTT sample. That's *much* too short.
  - You now send second datagram at t3.
  - Your timer is too small, so you retransmit very soon after t3 – the second segment has no hope of being ack'd before your timer pops.

- Every segment is now transmitted twice, even though no loss occurs!

- Steady state that's been observed for this situation:

$$RTT = (1/2)\, RTT_{actual} + \varepsilon$$

Ambiguity Problem Solution:

- Ignore RTT samples of any packet that was retransmitted.

- Problem:
  - Works ok if actual RTT changes *slowly*.

  - But *sudden spike* in actual RTT will cause retransmission for all subsequent packets:

    - The high RTT of retransmitted packets is ignored.
    - Hence sender is stuck with too small a timeout value
    - This in turn wastes network capacity when it can least afford it - during periods of congestion.
    - It's like pouring gas on a fire!

[C 12.18] Karn's algorithm and timer backoff (Karn's, Partridge, 1988)

- Virtually all TCP implementations (old and new) *increase timeout* upon retransmission.

  If 2 timeouts for *same* packet occur, timeout is increased still further.

- Increasing timeout = *backoff*

- Example (early TCP)

  - BSD 4.3 has table of factors for each successive retransmission
  - Simply double timeout

- Alternative to backoff:

  - Set timeout excessively high, so that no packet could survive retransmission.

  - This is a bad solution:
    It gives poor user throughput over a lossy path.

Karn's Algorithm:

- When an ack arrives for a packet sent >1 time, do not use it to compute RTT. Instead, back off timer.

- Retain backed off time for subsequent packets until a packet is sent and ack'd w/o retransmission.
  - At this point, recalculate TO from RTT_{last_seg} using formula (1).
  - This insures RTT_i will gradually converge to new, higher actual round trip time.

Q: How quickly does RTT converge to true value?
A: In worst case, need just 6 good samples of RTT_{last_seg} for RTT_i to converge on actual value, for ß = 2, α = .87. [Karn, Partridge 88]

TCP Modifications due to [Jacobson, 1987]

- New TCP spec due to congestion collapse (fall 86) in Internet:
  Throughput of 400 yard, 3-hop path dropped from 32kbps to 40bps.

- New principle: Conservation of packets:

  - "A new packet should not enter until an old one leaves"

  - New packets are "clocked in" by returning acks.
    - Self clocking systems automatically adjust to BW & delay variance and have a wide dynamic range
    - (TCP spans 1200bps to 800mbps)

Observation:

Congestion collapse occurs if conservation of packets is violated. This occurs due to one of 3 reasons:

1) Connection doesn't get to equilibrium
2) Sender injects new packet before old packet exits.
3) Equilibrium can' t be reached because of resource limits along path.

Solution to 1 - Slow start

Pre-1987:

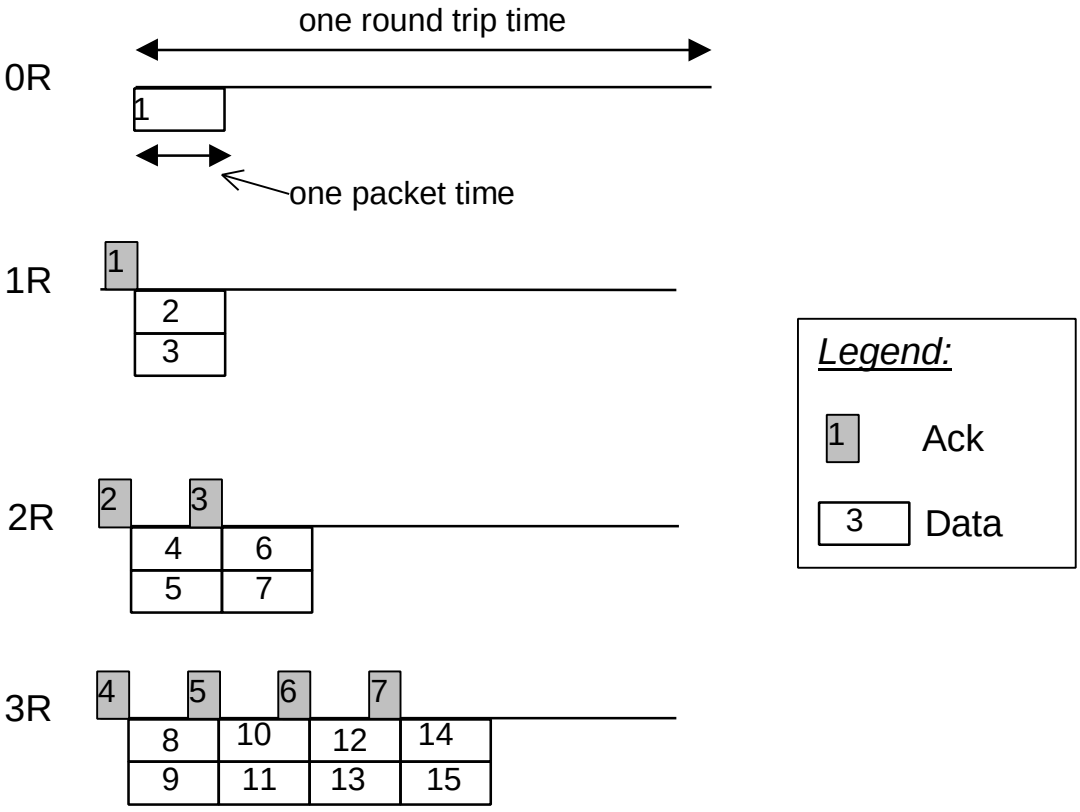
- If you suddenly start a file transfer for a host connected to a 10Mbps Ethernet and through a 56 Kbps gateway to the destination, you begin flooding Ethernet at 10 Mbps: 200 x (gateway BW).
- This causes continuous retransmissions.
- See [Jacobson, Fig. 3] – two Sun's on Ethernets connected by 230.4Kbps microwave link:
  - Only 35% of available BW was used
  - Packets 54 to 58 were sent 5 times each!
- See [Jacobson, Fig. 4] – Fig. 3 after applying slow start algorithm (described below).
  - Achieves 16 Kbps out of possible 20Kbps.
  - Twice the throughput of Fig. 3

So, there needs to be a way to "discover" bottleneck path BW. This is slow start:

- Add a congestion window, CWND, to each connection.
- When starting or restarting after a retransmission, set CWND to one packet.
- On each ack for new data, increase CWND by one packet.
- The value of *n* (in Go Back *n*) at sender is min(CWND, receiver's advertised window in last header received).

Example [Jacobson 1988, fig 2]

Opening a window of size 16:



- Horizontal direction is time.

- Continuous time line has been chopped into one-round-trip-time pieces stacked vertically with increasing time going down the page.

- As each ack arrives, two packets are generated:
  1. one for the ack (the ack says a packet has left the system, so a new packet is added to take its place)
  2. one because an ack opens the congestion window by one packet.

- Add-one-packet-to-window policy opens the window exponentially in time.

- If local net is much faster than long haul net, ack's two packets arrive at bottleneck at essentially the same time.
  - These two packets are shown stacked on top of one another (indicating that one of them would have to occupy space in the gateway's short term queue).
  - Thus, the short term queue demand on the gateway is increasing exponentially
  - Opening a window of size W packets will require W/2 packets of buffer capacity at the bottleneck.

Solution to 2 (Sender injects new packet before old packet exits):

- (2) only occurs if there's unnecessary retransmission
- Thus its critical to accurately estimate RTT algorithm
- New idea: estimate variance of RTT, denoted by σ_{RTT}

By queuing theory, RTT and σ_{RTT} both scale proportionally to $1/(1-\rho)$.

- Thus if ρ= 75%, RTT will vary by a factor of $\pm 2\sigma_{RTT}$, or $\pm 2 \cdot 4$, or a range of (-8, +8), or 16. Wow!
- Using early TCP standard's suggestion of ß = 2 means TCP can adapt only to of $\rho \leq 30\%$. So if load ρ goes above 30%, unnecessary retransmissions occur.

Responding to high variance in delay (Comer [12.19])

1989 TCP spec requires estimation of mean as well as variance in RTT. Let

- DEV denote standard deviation
- δ denote a weighting factor between 0 and 1 that controls how quickly the new sample affects the weighted average; typically 1/8

Then

$RTT_i = RTT_{i-1} + \delta (Sample - RTT_{i-1})$ [δ=1 means adapt instantly]

Note: above is faster to compute than

$RTT_i = \alpha * RTT_{i-1} + (1 - \alpha) * Sample$

Use standard deviation in place of ß to estimate Timeout (was ß * RTT_i)

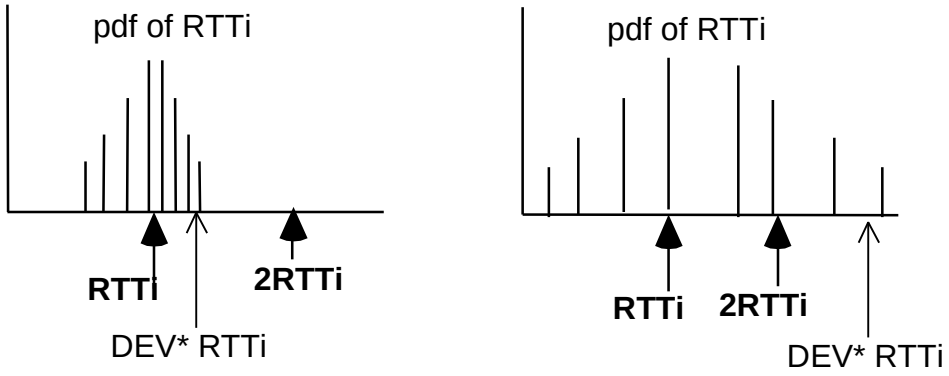
$Timeout_i = RTT_i + 2*DEV_i$,

where

$DEV_i = DEV_{i-1} + \delta (|Sample - RTT_{i-1}| - DEV_{i-1})$

Above formula doesn't use true std. deviation formula to avoid time consuming terms (e.g., squaring an integer)

Why timeout$_i$ = DEV$_i$ * RTT$_i$ is better than ß(=2) * RTT$_i$:



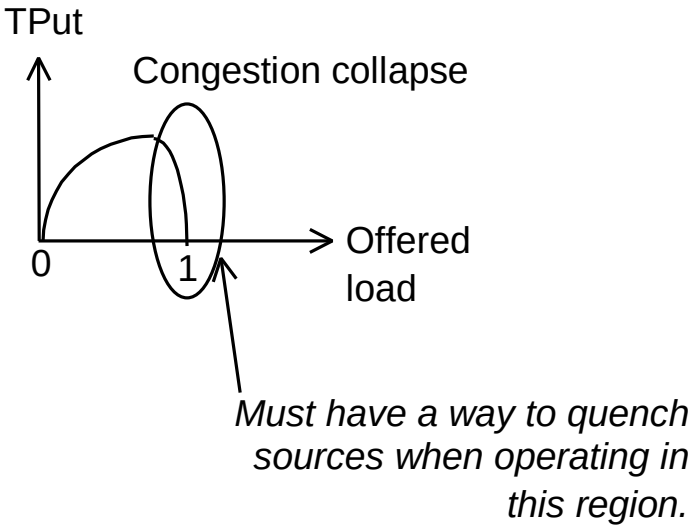
- RTT and σ_{RTT} both grow with network load, denoted ρ ($0 \leq \rho \leq 1$).

Compare Figs. 5 & 6 in [Jacobson]:
- Fig. 5 is ß(=2) * RTT$_i$
- Fig. 6 is DEV$_i$ * RTT$_i$

Summary:

- RTT is updated only for segments not retransmitted
- Timeout is doubled whenever segment timer pops
- When timeout is updated
  - -> higher std. dev leads to large timeout
  - -> small std. dev. leads to small timeout

[Comer 12.20] Dynamic setting of CWND (solution to 2 and 3)



Q: Why does curve decline?
A: Transport protocols retransmit when timers pop.
Long queue delays -> lots of times pop -> most datagrams are retransmissions.

So we use retransmission as a "signal" that network is congested:
- Decrease utilization if signal is received
- Increase utilization otherwise.

Note: retransmission is a good signal because *all* networks deliver it! No special bits need to be added to protocols or implementations.

Multiplicative Decrease:

- In steady state, on non-congested connection:
  CWND = Receiver's window

- On congestion (packet retransmission), queue lengths increase exponentially; thus we must decrease window size exponentially:

  SSTHRESH = SSTHRESH * *D* *(SSTHRESH is explained later)*

CWND = 1

where

$D = 1/2$

Also double timeout value, by Karn's algorithm

- This reduction of CWND throttles sender.

Additive Increase

- If you're sharing network link with one other person, you each will get 50% of BW.
- If she stops sending, how will you know that you can use more network BW?
- You must occasionally try increasing your BW utilization to discover the current limit:
  - Whenever you send an *entire* window w/o retransmission, you increase CWND by 1.
- We increase additively, not multiplicatively, to avoid wild oscillations in CWND:
  - Easy to drive net into saturation, hard for net to recover (think of rush hour traffic).

Summary of Additive Increase/Multiplicative Decrease

- *Additive Increase:* Upon receiving *each* ack, do this:

  CWND = CWND + μ

  where $\mu = 1/CWND$.

  So CWND increases by 1 when a full window's worth has been received w/o retransmission.

- *Multiplicative Decrease:* Upon timeout, half CWND.

- When sending, send the min(CWND, W_i)

How Slow Start and MD/AI work together

(Appendix B of [Jacobson 88])

Introduce a new variable:

SSTHRESH

It's the threshold to switch sender from slow start to MD/AI.

- On ack of non-retransmitted packet:

```
if (CWND < SSTHRESH)
    then /*slow start - open exponentially */
        CWND = CWND + 1
    else /*additive increase/
        CWND = CWND + 1/CWND
```

- On timeout

```
SSTHRESH = CWND/2;          /* multiplicative decrease */
CWND = 1                    /* to initialize slow start */
```

This moves CWND quickly from 1 to size that got us in trouble, then increases slowly to probe for more bandwidth on path.

Summary of growth of CWND:

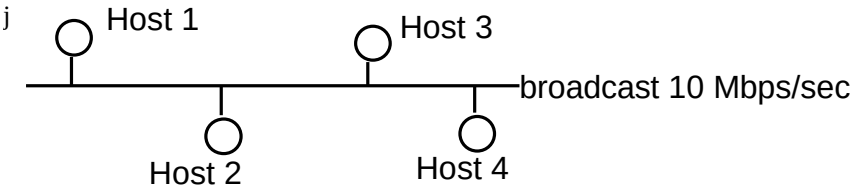
CWND = 1 initially

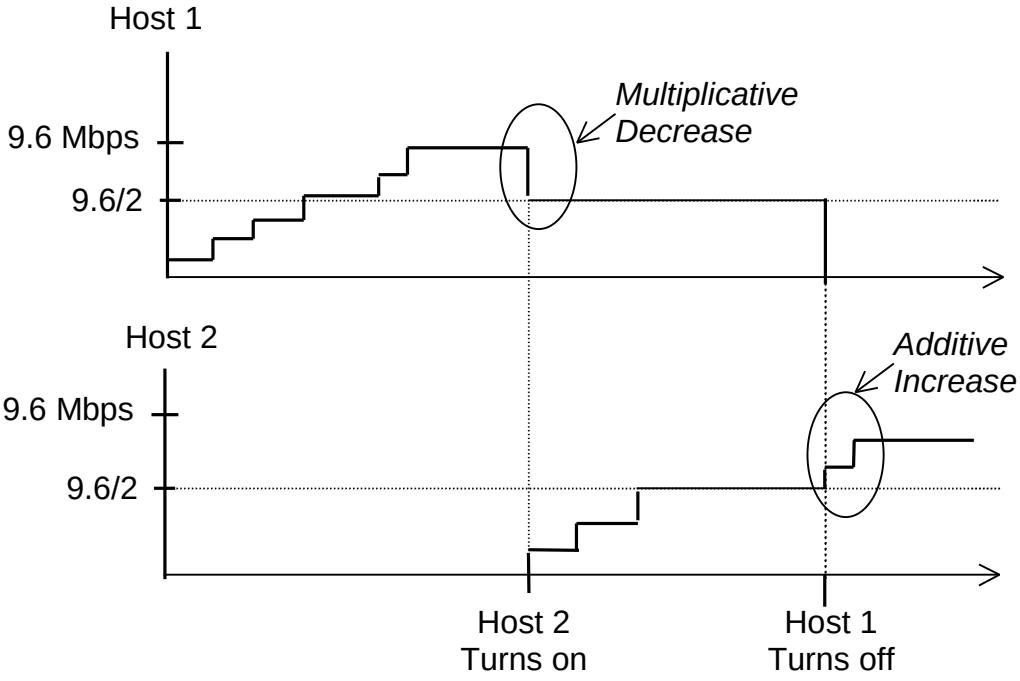
CWND = CWND + 1 each time a seg ack arrives and CWND is < 1/2 its original size

CWND = CWND + 1 if all segs in send window ack'd (if CWND is >= 1/2 its original size)

Multiplicative Decrease Intuition:

j





Automatically stabilizes to give each of m senders 1/mth of BW!

Summary: 1989 spec improved TCP performance by 2-fold to 10-fold with no significant increase in protocol software overhead using:

- Window size improvements:
  - Slow start
  - Mult. decrease
  - Congestion avoidance
- Timer improvements:
  - Measure variance
  - Exponential timer backoff (Karn's algorithm)