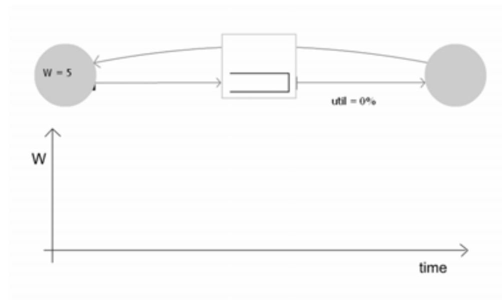
Internet Programming & Protocols Lecture 12

TCP evolution …

TCP Reno fast recovery
TCP NewReno partial ACK's
TCP SACK FACK D-SACK

TCP additive increase multiplicative decrease (AIMD)

cwnd ← cwnd + 1/cwnd for each ACK
Loss ➔ cwnd ← cwnd/2



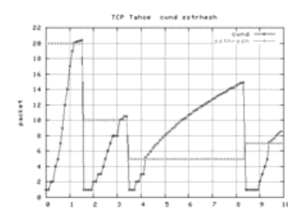
TCP evolution

- RFC 793
  - Crude RTT estimators
  - Initial window blast
  - Lost packet detection with timeout
  - Possible go-back-N on lost packet
  - Too many tiny packets
- Jacobson '88 (Tahoe, 4.3 BSD)
  - Refined RTT estimator
  - Slow-start
  - AIMD congestion control
    - ssthresh ← cwnd/2
    - cwnd ← 1 (slow-start) up to ssthresh (1/2 previous data rate), then linear increase of cwnd each RTT
  - Fast retransmit – 3 dup ACK's retransmit missing packet

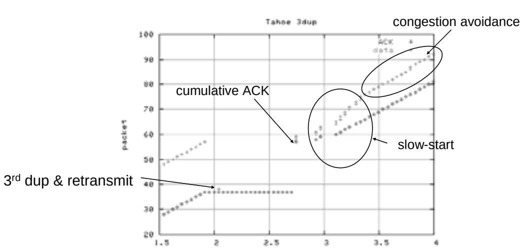
Conservation of packets – a new packet isn't put into the network until an old packet leaves. If sender's follow this principle, the network should be robust in the face of congestion.

Tahoe AIMD

- Cut sending rate in half if packet is lost ssthresh ← cwnd/2
- Set congestion window to 1 and do slow-start til ssthresh is reached
  - cwnd ← 1 then double cwnd every ACK til ssthresh
- Enter congestion avoidance phase (linear)
  - cwnd ← cwnd + 1/cwnd for every ACK
  - Increment cwnd by 1 every RTT



Tahoe fast retransmit



If 3rd dup ACK arrives, assume packet lost, retransmit and do AIMD. Avoid waiting for timeout

TCP Reno -- fast recovery

- Jacobson (email '90), then RFC 2581
- still same AIMD parameters (0.5, 1)
- After packet loss (3 dup) and retransmit, keep ACK clocking (pipe full) by sending new packet for each dup ACK received (if available window will allow)
- Since we keep ACK clocking, we don't need slow-start when congestion event ends (lost packet ACKd)
- 3rd dup ACK, cut cwnd in half, retransmit lost packet
- Sender's usable window *min(awin, cwnd+ndup)*
- Sender effectively waits til half a window of dup ACKs arrive, then sends a new packet for each additional dup ACK (cwnd ← cwnd/2)
  - Tahoe waited til window was empty (full RTT), cwnd ←1
- When a "new" ACK arrives, exit fast recovery with cwnd ← cwnd/2

Reno

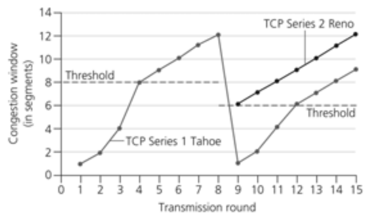
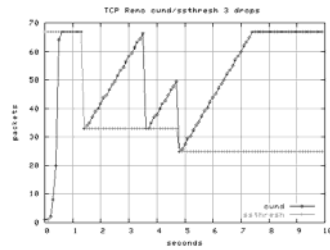


Figure 3.51 ♦ Evolution of TCP's congestion window (Tahoe and Reno)

© Kurose

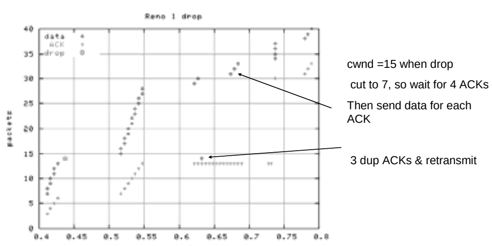
Reno cwnd and ssthresh

- Slow-start to begin flow
- 3 lost packets



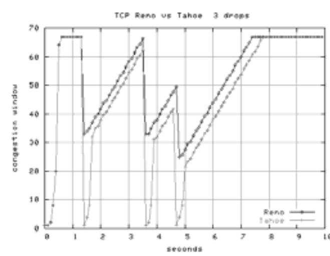
Reno recovery

- The graph illustrates Reno sending new packets for dup ACKs after half the dup ACKs were accounted for. This helps performance some, the real advantage of Reno over Tahoe is starting cwnd at cwnd/2 and not cwnd=1.



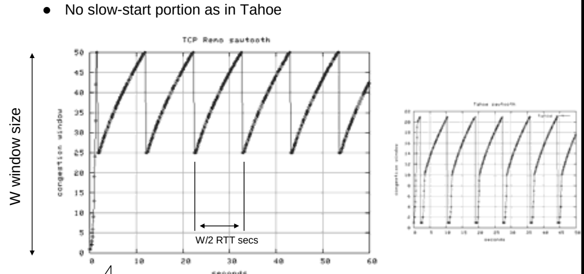
Reno vs Tahoe

- Reno has faster recovery from packet loss (e.g., higher bandwidth)



Reno sawtooth

- TCP Reno sender with send buffer bigger than router queue size
- No slow-start portion as in Tahoe



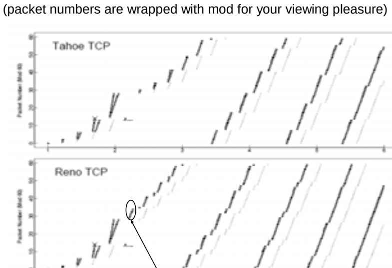
Data rate: (area of box + area of triangle)/time

$(W/2)^2 + ½ (W/2)^2 =$ **3/8 W² per cycle = ¾ W /sec**

Reno performance (one drop)

- Floyd's results Tahoe vs Reno with one packet drop (ns)

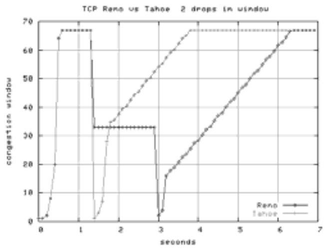
(packet numbers are wrapped with mod for your viewing pleasure)



While Tahoe is idle following retransmit, after cwnd/2 dup ACKs, Reno is transmitting with each dup ACK

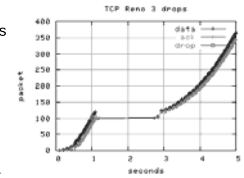
Reno problems

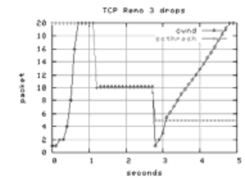
- If there are multiple packet drops in the same “window” (one RTT), Reno usually has to timeout to recover. It’s slower than Tahoe in this case! (Tahoe retransmits a lost packet each RTT)



Reno with 3 drops in the “window”

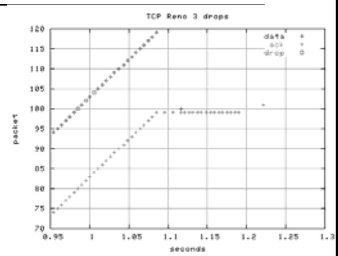
- Window is 20 segments, RTT 100ms
- 20 packets in flight, packets 100,102,104 are lost, all are in the “window” {100 … 119}
- 3 dup ACKs and retransmit 100
- cwnd goes from 20 to 10
- But Reno hangs then, and times out after 1.5 seconds, cwnd ← 1 and slow-start ☹





Reno stuck

- 3 dup ACKS, retansmit 100
- Inflight = 20 (or un ACK’d = 20)
- Dup ACKs for rest of window are received
- Can’t send on each dup ACK because “no room” at the receiver
- After RTT, cumulative ACK for 100-101 arrives



•Un ACK’d window is now 119-102+1 == 18, but cwnd is now only 10

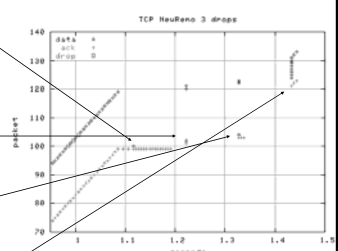
•Reno can’t send til un ACK’d drops to 10, but no more packets are in flight … timeout

NewReno

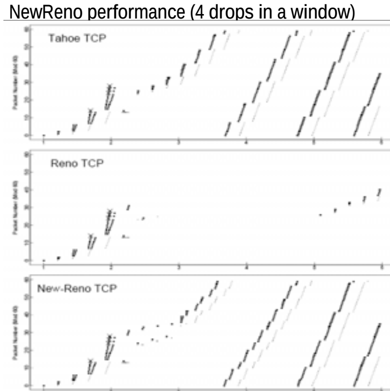
- Hoe (’95) and later RFC 2582
- Modify (fix) Reno recovery
  - If partial ACK arrives during Reno recovery, assume the next packet is lost, and retransmit
  - keeps ACK clock running
  - Stay in recovery til last packet in window ACK’d
- Out of order packet will cause unneeded retransmit, but worth it to fix Reno timeout problem
- If there are multiple drops in the window, one will be retransmitted each RTT

NewReno fix

- Same loss scenario {100,102,104} lost
- 3 dup ACK, retransmit 100
- Cut cwnd in half (20 → 10)
- After 1 RTT, ACK for 100-101, partial ACK, retransmit 102 and 120,121
- One more RTT, ACK for 102-103, retransmit 104 and 122,123
- One more RTT, cumulative ACK for rest of window, exit recovery, cwnd = 10 (note packet burst from cumulative ACK)



NewReno performance (4 drops in a window)



One retransmit per RTT

One retransmit then timeout!

One retransmit per partial ACK (one/RTT)

SACK

- SACK (Selective ACK)
  - First described by Jacobson ('88) RFC 1072
  - Refined in RFC 2018 ('96)
- TCP dup ACKs convey little information
  - ACK clocking may be lost (Reno)
  - Cumulative ACK can cause a burst
  - NewReno sends new packets for each dup if available window allows
- Selective ACK allows receiver to inform sender which segments have arrived successfully
  - Sender can fill in holes even when available window is full
  - NewReno algorithms still decide when to send, SACK says "what" to send
- Uses two TCP options
  - SACK "permitted" (option 4) used in SYN SYN-ACK
    - Receiver should generate SACK's only if it has received a permitted option
  - SACK blocks (option 5) carries SACK info in ACK packet

SACK

```
                  +--------+--------+
                  | Kind=5 | Length |
+--------+--------+--------+--------+
|      Left Edge of 1st Block       |
+--------+--------+--------+--------+
|      Right Edge of 1st Block      |
+--------+--------+--------+--------+
|                                   |
/            . . .                  /
|                                   |
+--------+--------+--------+--------+
|      Left Edge of nth Block       |
+--------+--------+--------+--------+
|      Right Edge of nth Block      |
+--------+--------+--------+--------+
```

Max TCP option space is 10 words

max of 4 SACK block pairs

(only 3, if timestamp option on)

Pair of 32-bit words (sequence numbers) indicating beginning and end of block of received data

First SACK block MUST report the block containing the most recently received segment

Receiver needs to do some bookkeeping to construct SACK blocks

Sender must accumulate SACK block info and use it when retransmitting packets

mark segments in SNDBUF that have been "received"

SACK examples

Sender sends 8 500-byte segments, first segment is dropped

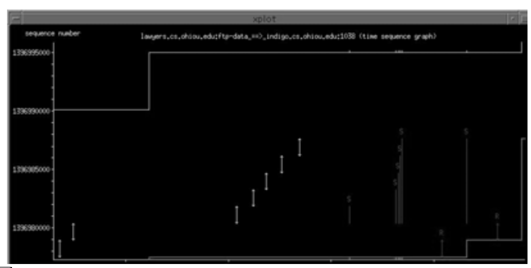
```
                    ----- SACK blocks  -------------------
Triggering   ACK   Left Edge Right Edge
 Segment
  5000     (lost)
  5500      5000     5500     6000
  6000      5000     5500     6500
  6500      5000     5500     7000
  7000      5000     5500     7500
  7500      5000     5500     8000
  8000      5000     5500     8500
  8500      5000     5500     9000
```

2nd example

```
  5000      5500
  5500      (lost)
  6000      5500     6000     6500
  6500      (lost)
  7000      5500     7000     7500        6000   6500
  7500      (lost)
  8000      5500     8000     8500        7000   7500       6000   6500
  8500      (lost)
```


SACK (tcptrace)

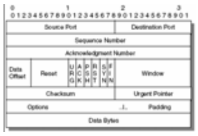
- 2 dropped packets, later packets arrive and are SACK'd
  - Note "growth" of SACK block
- Retransmissions and cumulative ACK



SACK option negotiation

- SACK "permitted", option 4, length 2

```
manitou.33878 > whisper.5001: S 885110161:885110161(0) win 5840 <mss
    1460,sackOK,timestamp 45695408 0,nop,wscale 0> (DF)
  4500 003c f421 4000 4006 a9ec c0a8 0104
  a024 3add 8456 1389 34c1 b591 0000 0000
  a002 16d0 cded 0000 0204 05b4 0402 080a
  02b9 41b0 0000 0000 0103 0300
whisper.5001 > manitou.33878: S 2714686246:2714686246(0) ack 885110162 win 5792
    <mss 1436,sackOK,timestamp 160286560 45695408,nop,wscale 5> (DF)
   4500 003c 0000 4000 3406 aa0e a024 3add
   c0a8 0104 1389 8456 a1ce d326 34c1 b592
   a012 16a0 883c 0000 0204 059c 0402 080a
   098d c760 02b9 41b0 0103 0305
```

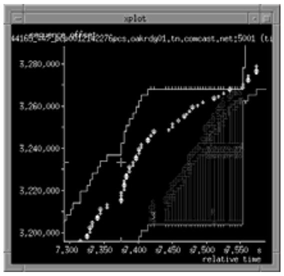


SACKs in an ACK

```
69.252.162.198.5001 > 160.36.58.221.44165: . ack 3204001 win 64080
    <nop,nop,timestamp 24421971 244071026,nop,nop,sack sack 1 {3205425:3215393} >
    (DF) [tos 0x20]
 4520 0040 9231 4000 3406 f0a2 45fc a2c6
 a024 3add 1389 ac85 2862 2e0a 9e3d 1ecd
 b010 fa50 1220 0000 0101 080a 0174 a653
 0e8c 3a72 0101 050a 9e3d 245d 9e3d 4b4d

69.252.162.198.5001 > 160.36.58.221.44165: . ack 3204001 win 64080
    <nop,nop,timestamp 24421977 244071026,nop,nop,sack sack 3
    {3242449:3245297}{3238177:3239601}{3205425:3236753} > (DF) [tos 0x20]
 4520 0050 9243 4000 3406 f080 45fc a2c6
 a024 3add 1389 ac85 2862 2e0a 9e3d 1ecd
 f010 fa50 424c 0000 0101 080a 0174 a659
 0e8c 3a72 0101 051a 9e3d b4fd 9e3d c01d
 9e3d a44d 9e3d a9dd 9e3d 245d 9e3d 9ebd
```

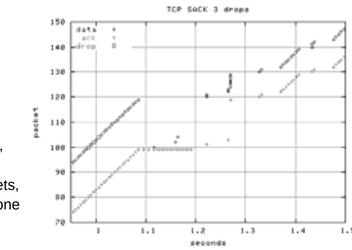
Multiple SACK blocks in an ACK packet



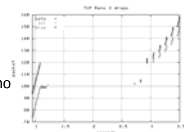
- Loss event with multiple drops
- TCP NewReno + SACK + FACK
  - Some new packets transmitted during event
  - Re-transmits to fill holes based on SACK

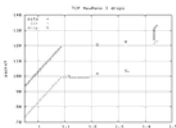
SACK with 3 drops

- Same 3 drop example
- Reno timed out
- NewReno did one retransmit each RTT for partial ACK
- When SACK info arrives, sender can immediately retransmit missing packets, and we exit recovery in one RTT



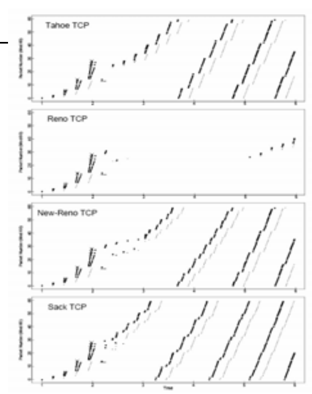
Reno





NewReno

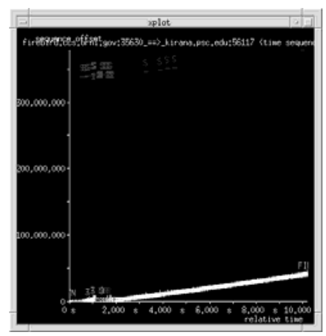
SACK performance



- No advantage for single drop
- Shines when multiple drops
- Example: 4 drops in a window
- SACK is top performer
- Most TCP's today are SACK+NewReno
- Is this too aggressive?
  - Multiple drops but cwnd/2
  - Or divide cwnd by 2 for each drop?

Applet data Flow statistics
53% experience loss
8% drop only one packet
12% experience timeout

Broken SACK ?



Strange SACKs from tcpdump of flow between ORNL and LBL.

Performance suffered because of SACK failure and resultant timeouts.

Bug in Cisco/PIX firewall:

Firewall was modifying TCP sequence numbers outbound and fixing them back when packets returned, but failed to account for sequence numbers in SACK blocks!

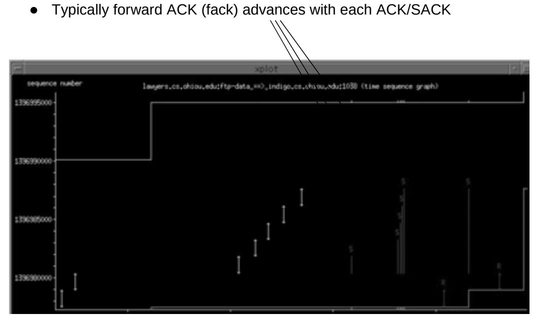
FACK

Example, 5 consecutive packets dropped –only one dup ACK for that event, but SACK info will indicate 5 are missing.

- Forward Acknowledgements (Mathis '96)
- Make better utilization of SACK info
  - SACK + NewReno retransmits a new packet for each dup ACK
  - FACK uses SACK block info to calculate exactly how many more packets may be transmitted and can often transmit more than one packet per dup ACK
  - Performs better than SACK+NewReno when many drops in window
- As SACK blocks arrive during recovery, FACK recalculates "fack"
  - fack == the most forward data held by the receiver
  - snd.una is the last acknowledged byte of the sender (left edge)
  - snd.nxt is the next byte to be sent (right edge)
  - FACK calculates actual data outstanding in the flow, awnd=snd.nxt-fack+retran_data where retran_data accounts for any segments retransmitted during this recovery period
  - When an ACK+SACK arrives these value are revised, and sender can send multiple segments if the following holds: while(awnd < cwnd) sendsomething();
  - Obeys our "conservation of packets" principle

FACK

- Typically forward ACK (fack) advances with each ACK/SACK

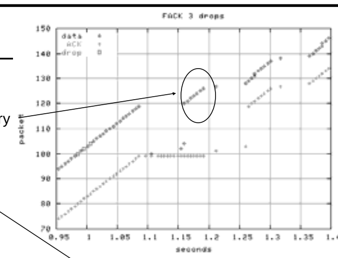


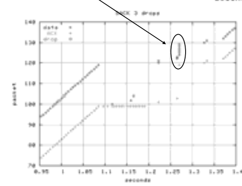
FACK faster retransmit

- FACK adjusts the dup ACK test from if (dupcnt ==3) to
  if ((fack – snd.una) > (3*MSS) || (dupcnt == 3)
- If multiple segments are lost before the 3rd dup ACK arrives, the earlier dup ACKs will carry SACK info that will allow a faster detection of possible loss (or reorder) and a "faster" retransmit
- If exactly one segment is lost, the two algorithms trigger recovery on the same dup ACK
- FACK improves throughput when there are multiple packet drops in one recovery window
- FACK is a little less bursty than SACK (a good thing)
- Linux usually has FACK enabled (it can only work when SACK is supported by both hosts)

FACK with 3 drops

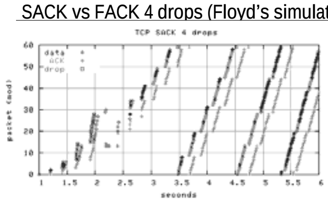
- Same 3 drop example
- Transmits during recovery
- No burst
- Slightly faster

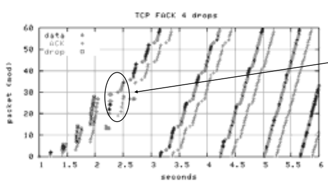




Bursts of packets are a bad thing. TCP avoids bursts with initial slow-start. But we can still have cwnd/2 burst from cumulative ACK after a loss event. (see rate-halving studies)

SACK vs FACK 4 drops (Floyd's simulations)





FACK is sending more packets earlier

D-SACK

- Duplicate SACK RFC 2883 ('00)
- use of the SACK option for acknowledging duplicate packets
- when duplicate packets are received, the first block of the SACK option field can be used to report the sequence numbers of the packet that triggered the acknowledgement
- TCP sender could then use this information for more robust operation where there are
  – reordered packets
  – ACK loss
  – packet replication
  – early retransmit timeouts
- No additional SYN negotiation needed, just use SACK negotiation

D-SACK

- The left edge of the D-SACK block specifies the first sequence number of the duplicate contiguous sequence, and the right edge of the D-SACK block specifies the sequence number immediately following the last sequence in the duplicate contiguous sequence.
- Distinguished from SACK in that segment extent is within ACK'd segment space (past left edge of window)
- Example, several ACK's lost so sender retransmits segment 3000-3499, receiver gets duplicate segment and sends D-SACK

Transmitted Segment	Received Segment	ACK Sent (Including SACK Blocks)
3000-3499	3000-3499	3500 (ACK dropped)
3500-3999	3500-3999	4000 (ACK dropped)
3000-3499	3000-3499	4000, SACK=3000-3500

TCP evolution summary

- Early fixes for tiny grams and silly window syndrome, source quench for congestion avoidance
- Jacobson fixes for congestion collapse: better RTT estimates, slow start, AIMD (no more go-back-N), and fast retransmit (Tahoe)
- Reno added fast recovery
- NewReno fixed timeout problem with Reno (multiple drops in a window)
- SACK/FACK allows sender to retransmit missing packets, faster recovery
- Other tweaks, RFC 1323, window-scaling and timestamps
- Most of these adjustments were to help TCP recover from packet loss
  – Packet loss is how TCP detects congestion/link capacity
  – Difficult to "see" these algorithms with tcpdump, need simulation or Web100
- With no packet loss, SNDBUF/RCVBUF and bottleneck link speed control TCP performance
- There are more flavors of TCP that we will investigate later
  – Vegas, HS TCP, BI TCP, scalable TCP, Westwood, …

TCP evolution

- Split IP/TCP
- ’81 RFC 793 TCP
- ’82 RFC 813 delayed ACKs, silly window,
- ’84 nagle, source quench
- ACK and buffer out of order (BSD UNIX)
- ‘88 slow-start, expo. backoff, cwnd/ssthresh , fast retransmit (Tahoe)
- ’90 Reno fast recovery
- ’90 header compression, path MTU discovery
- SACK, window scale, timestamp (’90 RFC 1323)
- ’94 Vegas (congestion avoidance, delay-based) FAST ’04
- ’96 SACK RFC 2018 FACK
- ’99 New Reno (partial ACK) RFC 2582
- ’00 D-SACK RFC 28883
- ECN/AQM
- Congestion relief: HS TCP, BI TCP, Scalable TCP, Binomial TCP, TCP Westwood,…

Concept Collection

- ACK/NAK cumulative ACK
- ACK clocking
- AIMD
- Bandwidth-delay product
- Best effort
- Bit error rate
- Checksums
- Client/server/concurrent/iterative
- Congestion control/avoid
- Conservation of packets
- CIDR
- CSMA/CD
- cwnd/sstrhesh
- Datagram vs reliable stream
- Dup threshold
- Exponential backoff
- Flow control
- Forward ACK
- fragmentation
- Layers/encapsulation
- Maximum segment lifetime(MSL)
- MTU MSS/MTU discovery
- Network mask
- Packet switching vs circuit-based
- Partial ACK
- promiscuous
- Routing
- RTT and RTT estimation
- Selective ACK (SACK)
- Self-clocking
- Sliding window
- Slow-start
- Subnets/supernets
- Switch vs hub
- TTL

Next time …

- Network programming in Java, Perl, Windows
- review