Dear Instructor:

This *Instructors' Manual* contains solutions to almost all of the exercises in the second edition of Peterson and Davie's *Computer Networks: A Systems Approach*. The goal of the exercise program for the second edition has been to provide a wide range of exercises supporting the text that are both accessible and interesting. When applicable, exercises were chosen that attempt to illuminate why things are done the way they are, or how they might be done differently. It is hoped that these exercises will prove useful to:

- support mastery of basic concepts through straightforward examples
- provide a source of classroom examples and discussion topics
- provide a study guide and source of exam problems
- introduce occasional supplemental topics
- support an exercise-intensive approach to the teaching of networks.

Exercises are sorted (roughly) by section, not difficulty. While some exercises are more difficult than others, none are intended to be fiendishly tricky. A few exercises (notably, though not exclusively, the ones that involve calculating simple probabilities) require a modest amount of mathematical background; most do not. There is a sidebar summarizing much of the applicable basic probability theory in Chapter 2.

An occasional exercise (*eg* 4.21) is awkwardly or ambiguously worded in the text. This manual sometimes suggests better versions; see also the errata at the web site, below.

Where appropriate, relevant supplemental files for these solutions (*eg* programs) have been placed on the textbook web site, `www.mkp.com/pd2e`. Useful other material can also be found there, such as errata, sample programming assignments, PowerPoint lecture slides, EPS figures, and the x-kernel code and tutorial. If you have any questions about these support materials, please contact your Morgan Kaufmann sales representative. If you would like to contribute your own teaching materials to this site, please contact Karyn Johnson, Morgan Kaufmann Editorial Department, kjohnson@mkp.com.

We welcome bug reports and suggestions as to improvements for both the exercises and the solutions; these may be sent to netbugs@mkp.com.

Peter Lars Dordal
`pld@cs.luc.edu`
Loyola University of Chicago
September, 1999

Problems worthy
    of attack
prove their worth
    by hitting back.

Piet Hein

Solutions for Chapter 1

3. Success here depends largely on the ability of ones search tool to separate out the chaff. I thought a naive search for Ethernet would be hardest, but I now think it's MPEG.

Mbone	www.mbone.com
ATM	www.atmforum.com
MPEG	try searching for "mpeg format", or (1999) drogo.cselt.stet.it/mpeg
IPv6	playground.sun.com/ipng, www.ipv6.com
Ethernet	good luck.

5. We will count the transfer as completed when the last data bit arrives at its destination. An alternative interpretation would be to count until the last ACK arrives back at the sender, in which case the time would be half an RTT (50 ms) longer.

   (a) 2 initial RTT's (200ms) + 1000KB/1.5Mbps (transmit) + RTT/2 (propagation) $\approx$ 0.25 + 8Mbit/1.5Mbps = $0.25 + 5.33$ sec $= 5.58$ sec. If we pay more careful attention to when a mega is 10^6 versus 2^{20}, we get 8,192,000 bits/1,500,000 bits/sec = 5.46 sec, for a total delay of 5.71 sec.

   (b) To the above we add the time for 999 RTTs (the number of RTTs between when packet 1 arrives and packet 1000 arrives), for a total of $5.71 + 99.9 = 105.61$.

   (c) This is 49.5 RTTs, plus the initial 2, for 5.15 seconds.

   (d) Right after the handshaking is done we send one packet. One RTT after the handshaking we send two packets. At n RTTs past the initial handshaking we have sent $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$ packets. At $n = 9$ we have thus been able to send all 1,000 packets; the last batch arrives 0.5 RTT later. Total time is 2+9.5 RTTs, or 1.15 sec.

6. Propagation delay is 2×10^3 m/(2×10^8 m/sec) = 1×10^{-5} sec = 10 μs. 100 bytes/10 μs is 10 bytes/μs, or 10 MB/sec, or 80 Mbit/sec. For 512-byte packets, this rises to 409.6 Mbit/sec.

7. Postal addresses are strongly hierarchical (with a geographical hierarchy, which network addressing may or may not use). Addresses also provide embedded "routing information". Unlike typical network addresses, postal addresses are long and of variable length and contain a certain amount of redundant information. This last attribute makes them more tolerant of minor errors and inconsistencies.

   Telephone numbers are more similar to network addresses (although phone numbers are nowadays apparently more like network host names than addresses): they are (geographically) hierarchical, fixed-length, administratively assigned, and in more-or-less one-to-one correspondence with nodes.

8. One might want addresses to serve as *locators*, providing hints as to how data should be routed. One approach for this is to make addresses *hierarchical*.

   Another property might be *administratively assigned*, versus, say, the *factory-assigned* addresses used by Ethernet. Other address attributes that might be relevant are *fixed-length* v. *variable-length*, and *absolute* v. *relative* (like file names).

   If you phone a toll-free number for a large retailer, any of dozens of phones may answer. Arguably, then, all these phones have the same non-unique "address". A more traditional

application for non-unique addresses might be for reaching any of several equivalent servers (or routers).

9. Video or audio teleconference transmissions among a reasonably large number of widely spread sites would be an excellent candidate: unicast would require a separate connection between each pair of sites, while broadcast would send far too much traffic to sites not interested in receiving it.

   Trying to reach any of several equivalent servers or routers might be another use for multicast, although broadcast tends to work acceptably well for things on this scale.

10. STDM and FDM both work best for channels with constant and uniform bandwidth requirements. For both mechanisms bandwidth that goes unused by one channel is simply wasted, not available to other channels. Computer communications are bursty and have long idle periods; such usage patterns would magnify this waste.

    FDM and STDM also require that channels be allocated (and, for FDM, be assigned bandwidth) well in advance. Again, the connection requirements for computing tend to be too dynamic for this; at the very least, this would pretty much preclude using one channel per connection.

    FDM was preferred historically for tv/radio because it is very simple to build receivers; it also supports different channel sizes. STDM was preferred for voice because it makes somewhat more efficient use of the underlying bandwidth of the medium, and because channels with different capacities was not originally an issue.

11. 1 Gbps = 10^9 bps, meaning each bit is 10^{-9} sec (1 ns) wide. The length in the wire of such a bit is 1 ns $\times$ 2.3×10^8 m/sec = 0.23 m

12. x KB is $8 \times 1024 \times x$ bits. y Mbps is $y \times 10^6$ bps; the transmission time would be $8 \times 1024 \times x/y \times 10^6$ sec = $8.192x/y$ ms.

13. (a) The minimum RTT is $2 \times 385,000,000$ m / 3×10^8 m/sec = 2.57 sec.
    (b) The delay$\times$bandwidth product is 2.57 sec$\times$100 Mb/sec = 257Mb = 32 MB.
    (c) This represents the amount of data the sender can send before it would be possible to receive a response.
    (d) We require at least one RTT before the picture could begin arriving at the ground (TCP would take two RTTs). Assuming bandwidth delay only, it would then take 25MB/100Mbps = 200Mb/100Mbps = 2.0 sec to finish sending, for a total time of $2.0 + 2.57 = 4.57$ sec until the last picture bit arrives on earth.

14. (a) Delay-sensitive; the messages exchanged are short.
    (b) Bandwidth-sensitive, particularly for large files. (Technically this does presume that the underlying protocol uses a large message size or window size; stop-and-wait transmission (as in Section 2.5 of the text) with a small message size would be delay-sensitive.)
    (c) Delay-sensitive; directories are typically of modest size.
    (d) Delay-sensitive; a file's attributes are typically much smaller than the file itself (even on NT filesystems).

15. (a) One packet consists of 5000 bits, and so is delayed due to bandwidth 500 μs along each link. The packet is also delayed 10 μs on each of the two links due to propagation delay, for a total of 1020 μs.

    (b) With three switches and four links, the delay is

    $$4 \times 500\mu\text{s} + 4 \times 10\mu\text{s} = 2.04\text{ms}$$

    (c) With cutthrough, the switch delays the packet by 200 bits = 20 μs. There is still one 500 μs delay waiting for the last bit, and 20 μs of propagation delay, so the total is 540 μs. To put it another way, the last bit still arrives 500 μs after the first bit; the first bit now faces two link delays and one switch delay but never has to wait for the last bit along the way. With three cut-through switches, the total delay would be:

    $$500 + 3 \times 20 + 4 \times 10 = 600\,\mu s$$

16. (a) The effective bandwidth is 10 Mbps; the sender can send data steadily at this rate and the switches simply stream it along the pipeline. We are assuming here that no ACKs are sent, and that the switches can keep up and can buffer at least one packet.

    (b) The data packet takes 2.04 ms as in 15(b) above to be delivered; the 400 bit ACKs take 40 μs/link for a total of $4 \times 40\,\mu$s+$4 \times 10\,\mu$s = 200 μsec = 0.20 ms, for a total RTT of 2.24 ms. 5000 bits in 2.24 ms is about 2.2 Mbps, or 280 KB/sec.

    (c) $100 \times 6.5 \times 10^8$ bytes / 12 hours = 6.5×10^{10} bytes/(12×3600 sec) $\approx$ 1.5 MByte/sec = 12 Mbit/sec

17. (a) 1×10^7bits/sec $\times\ 10^{-6}$ sec = 100 bits = 12.5 bytes

    (b) The first-bit delay is 520 μs through the store-and-forward switch, as in 15(a). 10^7bits/sec $\times$ 520$\times10^{-6}$ sec = 5200 bits. Alternatively, each link can hold 100 bits and the switch can hold 5000 bits.

    (c) 1.5×10^6 bits/sec $\times\ 50 \times 10^{-3}$ sec = 75,000 bits = 9375 bytes

    (d) This was intended to be *through* a satellite, *ie* between two ground stations, not *to* a satellite; this ground-to-ground interpretation makes the total one-way travel distance 2×35,900,000 meters. With a propagation speed of $c = 3\times10^8$ meters/sec, the one-way propagation delay is thus 2×35,900,000/c = 0.24 sec. Bandwidth×delay is thus 1.5×10^6 bits/sec × 0.24 sec = 360,000 bits $\approx$ 45 KBytes

18. (a) Per-link transmit delay is 10^4 bits / 10^7 bits/sec = 1000 μs. Total transmission time = $2 \times 1000 + 2 \times 20 + 35 = 2075\ \mu$s.

    (b) When sending as two packets, here is a table of times for various events:

T=0	start
T=500	A finishes sending packet 1, starts packet 2
T=520	packet 1 finishes arriving at S
T=555	packet 1 departs for B
T=1000	A finishes sending packet 2
T=1055	packet 2 departs for B
T=1075	bit 1 of packet 2 arrives at B
T=1575	last bit of packet 2 arrives at B

Expressed algebraically, we now have a total of one switch delay and two link delays; transmit delay is now 500 μs:

$$3 \times 500 + 2 \times 20 + 1 \times 35 = 1575\,\mu\text{s}.$$

Smaller is faster, here.

19. (a) Without compression the total time is 1 MB/*bandwidth*. When we compress the file, the total time is

$$compression\_time + compressed\_size/bandwidth$$

Equating these and rearranging, we get

$$bandwidth = compression\_size\_reduction/compression\_time$$

= 0.5 MB/1 sec = 0.5 MB/sec for the first case,
= 0.6 MB/2 sec = 0.3 MB/sec for the second case.

(b) Latency doesn't affect the answer because it would affect the compressed and uncompressed transmission equally.

20. The number of packets needed, N, is $\lceil 10^6/D \rceil$, where D is the packet data size. Given that overhead = $100 \times N$ and loss = D (we have already counted the lost packet's header in the overhead), we have overhead+loss = $100 \times \lceil 10^6/D \rceil + D$.

D	overhead+loss
1000	101000
5000	25000
10000	20000
20000	25000

21. The time to send one 2000-bit packet is 2000 bits/100 Mbps = 20 μs. The length of cable needed to exactly contain such a packet is 20 μs $\times$ 2×10^8 m/sec = 4,000 meters.

250 bytes in 4000 meters is 2000 bits in 4000 meters, or 50 bits per 100 m. With an extra 10 bits/100 m, we have a total of 60 bits/100 m. A 2000-bit packet now fills 2000/(.6 bits/m) = 3333 meters.

22. For music we would need considerably more bandwidth, but we could tolerate high (but bounded) delays. We could *not* necessarily tolerate higher jitter, though; see Section 6.5.1.

We might accept an audible error in voice traffic every few seconds; we might reasonably want the error rate during music transmission to be a hundredfold smaller. Audible errors would come either from outright packet loss, or from jitter (a packet's not arriving on time).

Latency requirements for music, however, might be much lower; a several-second delay would be inconsequential. Voice traffic has at least a tenfold faster requirement here.

23. (a) $640 \times 480 \times 3 \times 30$ bytes/sec = 26.4 MB/sec

(b) $160 \times 120 \times 1 \times 5$ = 96,000 bytes/sec = 94KB/sec

(c) 650MB/75 min = 8.7 MB/min = 148 KB/sec

(d) $8 \times 10 \times 72 \times 72$ pixels = 414,720 bits = 51,840 bytes. At 14,400 bits/sec, this would take 28.8 seconds (ignoring overhead for framing and acknowledgements).

24. (a) A file server needs lots of peak bandwidth. Latency is relevant only if it dominates bandwidth; jitter and average bandwidth are inconsequential. No lost data is acceptable, but without realtime requirements we can simply retransmit lost data.

(b) A print server needs less bandwidth than a file server (unless images are extremely large). We may be willing to accept higher latency than (a), also.

(c) A file server *is* a digital library of a sort, but in general the world wide web gets along reasonably well with much less peak bandwidth than most file servers provide.

(d) For instrument monitoring we don't care about latency or jitter. If data were continually generated, rather than bursty, we might be concerned mostly with average bandwidth rather than peak, and if the data really were routine we might just accept a certain fraction of loss.

(e) For voice we need guaranteed average bandwidth and bounds on latency and jitter. Some lost data might be acceptable; eg resulting in minor dropouts many seconds apart.

(f) For video we are primarily concerned with average bandwidth. For the simple monitoring application here, relatively modest video of Exercise 23(b) might suffice; we could even go to monochrome (1 bit/pixel), at which point $160 \times 120 \times 5$ frames/sec requires 12KB/sec. We could tolerate multi-second latency delays; the primary restriction is that if the monitoring revealed a need for intervention then we still have time to act. Considerable loss, even of entire frames, would be acceptable.

(g) Full-scale television requires massive bandwidth. Latency, however, could be hours. Jitter would be limited only by our capacity absorb the arrival-time variations by buffering. Some loss would be acceptable, but large losses would be visually annoying.

25. In STDM the offered timeslices are always the same length, and are wasted if they are unused by the assigned station. The round-robin access mechanism would generally give each station only as much time as it needed to transmit, or none if the station had nothing to send, and so network utilization would be expected to be much higher.

26. (a) In the absence of any packet losses or duplications, when we are expecting the Nth packet we *get* the Nth packet, and so we can keep track of N locally at the receiver.

(b) The scheme outlined here is the stop-and-wait algorithm of Section 2.5; as is indicated there, a header with at least one bit of sequence number is needed (to distinguish between receiving a new packet and a duplication of the previous packet).

(c) With out-of-order delivery allowed, packets up to 1 minute apart must be distinguishable via sequence number. Otherwise a very old packet might arrive and be accepted as current. Sequence numbers would have to count as high as

$$bandwidth \times 1\,\text{minute} / packet\_size$$

27. In each case we assume the local clock starts at 1000.

(a) Latency: 100. Bandwidth: high enough to read the clock every 1 unit.

1000	1100	
1001	1101	
1002	1102	
1003	1104	tiny bit of jitter: latency = 101
1004	1104	

(b) Latency=100; bandwidth: only enough to read the clock every 10 units. Arrival times fluctuate due to jitter.

1000	1100	
1020	1110	latency = 90
1040	1145	
1060	1180	latency = 120
1080	1184	

(c) Latency = 5; zero jitter here:

1000	1005	
1001	1006	
1003	1008	we lost 1002
1004	1009	
1005	1010	

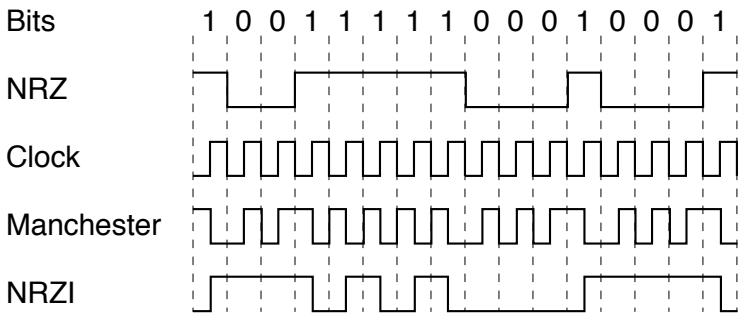
28. Generally, with MAX_PENDING =1, one or two connections will be accepted and queued; that is, the data won't be delivered to the server. The others will be ignored; eventually they will time out.

    When the first client exits, any queued connections are processed.

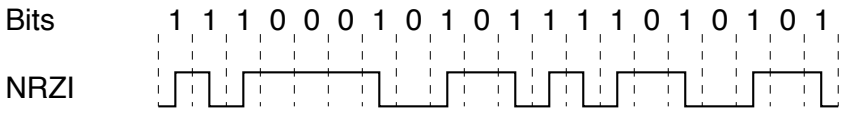
30. Note that UDP accepts a packet of data from any source at any time; TCP requires an advance connection. Thus, two clients can now talk simultaneously; their messages will be interleaved on the server.

Solutions for Chapter 2

1.



2.



3. One can list all 5-bit sequences and count, but here is another approach: there are 2^3 sequences that start with 00, and 2^3 that end with 00. There are two sequences, 00000 and 00100, that do both. Thus, the number that do either is $8 + 8 - 2 = 14$, and finally the number that do neither is $32 - 14 = 18$. Thus there would have been enough 5-bit codes meeting the stronger requirement; however, additional codes are needed for control sequences.

4. The stuffed bits (zeros) are in bold:
1101 0111 11**0**0 1011 111**0** 1010 1111 1**0**11 0

5. The $\wedge$ marks each position where a stuffed 0 bit was removed. There were no stuffing errors detectable by the receiver; the only such error the receiver could identify would be seven 1's in a row.
1101 0111 11$\wedge$10 1111 1$\wedge$010 1111 1$\wedge$110

6. ..., DLE, DLE, DLE, ETX, ETX

7. (a) X DLE Y, where X can be anything besides DLE and Y can be anything except DLE or ETX. In other words, each DLE must be followed by either DLE or ETX.

   (b) 0111 1111.

8. (a) After 48×8=384 bits we can be off by no more than ±1/2 bit, which is about 1 part in 800.

   (b) One frame is 810 bytes; at STS-1 51.8 Mbps speed we are sending $51.8 \times 10^6/(8 \times 810)$ = about 8000 frames/sec, or about 480,000 frames/minute. Thus, if station B's clock ran faster than station A's by one part in 480,000, A would accumulate about one extra frame per minute.

9. Suppose an undetectable three-bit error occurs. The three bad bits must be spread among one, two, or three rows. If these bits occupy two or three rows, then some row must have exactly one bad bit, which would be detected by the parity bit for that row. But if the three bits are all

in one row, then that row must again have a parity error (as must each of the three columns containing the bad bits).

10. If we flip the bits corresponding to the corners of a rectangle in the 2-D layout of the data, then all parity bits will still be correct. Furthermore, if four bits change and no error is detected, then the bad bits must form a rectangle: in order for the error to go undetected, each row and column must have no errors or exactly two errors.

11. If we know only one bit is bad, then 2-D parity tells us which row and column it is in, and we can then flip it. If, however, two bits are bad in the same row, then the row parity remains correct, and all we can identify is the columns in which the bad bits occur.

12. We need to show that the 1's-complement sum of two non-0x0000 numbers is non-0x0000. If no unsigned overflow occurs, then the sum is just the 2's-complement sum and can't be 0000 without overflow; in the absence of overflow, addition is monotonic. If overflow occurs, then the result is at least 0x0000 plus the addition of a carry bit, ie $\geq$0x0001.

13. Consider only the 1's complement sum of the 16-bit words. If we decrement a low-order byte in the data, we decrement the sum by 1, and can incrementally revise the old checksum by decrementing it by 1 as well. If we decrement a high-order byte, we must decrement the old checksum by 256.

14. Here is a rather combinatorial approach. Let a, b, c, d be 16-bit words. Let $[a, b]$ denote the 32-bit concatenation of a and b, and let $carry(a, b)$ denote the carry bit (1 or 0) from the 2's-complement sum $a + b$ (denoted here $a +_2 b$). It suffices to show that if we take the 32-bit 1's complement sum of $[a, b]$ and $[c, d]$, and then add upper and lower 16 bits, we get the 16-bit 1's-complement sum of a, b, c, and d. We note $a +_1 b = a +_2 b +_2 carry(a, b)$.

    The basic case is supposed to work something like this. First,

    $$[a, b] +_2 [c, d] = [a +_2 c +_2 carry(b, d), b +_2 d]$$

    Adding in the carry bit, we get

    $$[a, b] +_1 [c, d] = [a +_2 c +_2 carry(b, d), b +_2 d +_2 carry(a, c)] \tag{1}$$

    Now we take the 1's complement sum of the halves,

    $$a +_2 c +_2 carry(b, d) +_2 b +_2 d +_2 carry(a, c) + (carry(wholething))$$

    and regroup:

    $$= a +_2 c +_2 carry(a, c) +_2 b +_2 d +_2 carry(b, d) + (carry(wholething))$$

    $$= (a +_1 c) +_2 (b +_1 d) + carry(a +_1 c, b +_1 d)$$

    $$= (a +_1 c) +_1 (b +_1 d)$$

    which by associativity and commutativity is what we want.

    There are a couple annoying special cases, however, in the preceding, where a sum is 0xFFFF and so adding in a carry bit triggers an additional overflow. Specifically, the $carry(a, c)$ in

(1) is actually $carry(a, c, carry(b, d))$, and secondly adding it to $b +_2 d$ may cause the lower half to overflow, and no provision has been made to carry over into the upper half. However, as long as $a +_2 c$ and $b +_2 d$ are not equal to 0xFFFF, adding 1 won't affect the overflow bit and so the above argument works. We handle the 0xFFFF cases separately.

Suppose that $b +_2 d = \text{0xFFFF} =_2 0$. Then $a +_1 b +_1 c +_1 d = a +_1 c$. On the other hand, $[a, b] +_1 [c, d] = [a +_2 b, \text{0xFFFF}] + carry(a, b)$. If $carry(a, b) = 0$, then adding upper and lower halves together gives $a +_2 b = a +_1 b$. If $carry(a, b) = 1$, we get $[a, b] +_1 [c, d] = [a +_2 b +_2 1, 0]$ and adding halves again leads to $a +_1 b$.

Now suppose $a +_2 c = \text{0xFFFF}$. If $carry(b, d) = 1$ then $b +_2 d \neq \text{0xFFFF}$ and we have $[a, b] +_1 [c, d] = [0, b +_2 d +_2 1]$ and folding gives $b +_1 d$. The $carry(b, d) = 0$ case is similar.

Alternatively, we may adopt a more algebraic approach. We may treat a buffer consisting of n-bit blocks as a large number written in base 2^n. The numeric value of this buffer is congruent $\bmod (2^n - 1)$ to the (exact) sum of the "digits", that is to the exact sum of the blocks. If this latter sum has more than n bits, we can repeat the process. We end up with the n-bit 1's-complement sum, which is thus the remainder upon dividing the original number by $2^n - 1$.

Let b be the value of the original buffer. The 32-bit checksum is thus $b \bmod 2^{32} - 1$. If we fold the upper and lower halves, we get $(b \bmod (2^{32} - 1)) \bmod (2^{16} - 1)$, and, because $2^{32} - 1$ is divisible by $2^{16} - 1$, this is $b \bmod (2^{16} - 1)$, the 16-bit checksum.

15. (a) We take the message 11001001, append 000 to it, and divide by 1001. The remainder is 011; what we transmit is the original message with this remainder appended, or 1100 1001 011.

    (b) Inverting the first bit gives 0100 1001 011; dividing by 1001 ($x^3 + 1$) gives a quotient of 0100 0001 and a remainder of 10.

16. (b)

p	q	C×q
000	000	000 000
001	001	001 101
010	011	010 111
011	010	011 010
100	111	100 011
101	**110**	**101 110**
110	100	110 100
111	101	111 001

    (c) The bold entries 101 (in the dividend), 110 (in the quotient), and 101 110 in the body of the long division here correspond to the bold row of the preceding table.

```
                 110   101   011   100
        ---------------------------------
1101  | 101   001   011   001   100
        101   110
        -----------
              111   011
              111   001
              -------------
                    010   001
                    010   111
                    -------------
                          110   100
                          110   100
                          -------------
                                  0
```

17. (a) M has eight elements; there are only four values for e, so there must be m_1 and m_2 in M with $e(m_1) = e(m_2)$. Now if m_1 is transmuted into m_2 by a two-bit error, then the error-code e cannot detect this.

    (b) For a crude estimate, let M be the set of N-bit messages with four 1's, and all the rest zeros. The size of M is $(N \text{ choose } 4) = N!/(4!(N-4)!)$. Any element of M can be transmuted into any other by an 8-bit error. If we take N large enough that the size of M is bigger than 2^{32}, then as in part (a) there must for any 32-bit error code function $e(m)$ be elements m_1 and m_2 of M with $e(m_1) = e(m_2)$. To find a sufficiently large N, we note $N!/(4!(N-4)!) > (N-3)^4/24$; it thus suffices to find N so $(N-3)^4 > 24 \times 2^{32} \approx 10^{11}$. $N \approx 600$ works. Considerably smaller estimates are possible.

18. Assume a NAK is sent only when an out-of-order packet arrives. The receiver must now maintain a RESEND_NAK timer in case the NAK, or the packed it NAK'ed, is lost.

    Unfortunately, if the sender sends a packet and is then idle for a while, and this packet is lost, the receiver has no way of noticing the loss. Either the sender must maintain a timeout anyway, requiring ACKs, or else some zero-data filler packets must be sent during idle times. Both are burdensome.

    Finally, at the end of the transmission a strict NAK-only strategy would leave the sender unsure about whether *any* packets got through. A final out-of-order filler packet, however, might solve this.

19. (a) Propagation delay = $20 \times 10^3\,\text{m}/(2 \times 10^8\,\text{m/sec}) = 100\,\mu\text{s}$.

    (b) The roundtrip time would be about $200\,\mu$s. A plausible timeout time would be twice this, or 0.4 ms. Smaller values (but larger than 0.2 ms!) might be reasonable, depending on the amount of variation in actual RTTs. See Section 5.2.5 of the text.

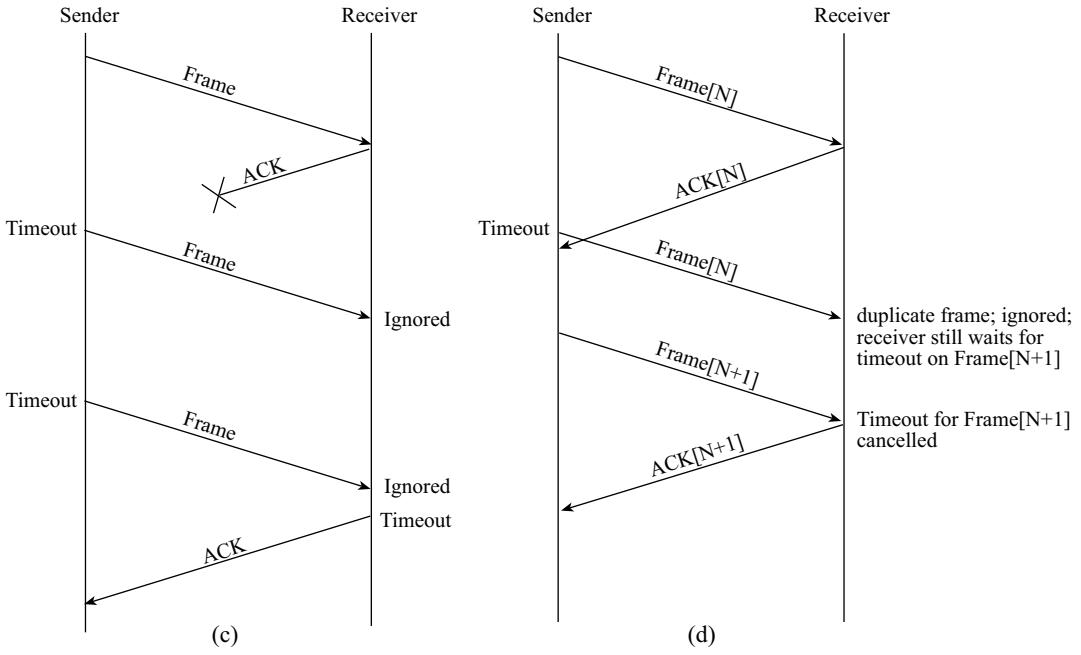
    (c) The propagation-delay calculation does not consider processing delays that may be introduced by the remote node; it may not be able to answer immediately.

20. Bandwidth×(roundtrip)delay is about 125KB/sec × 2.5 sec, or 312 packets. The window size should be this large; the sequence number space must cover twice this range, or up to 624. 10 bits are needed.

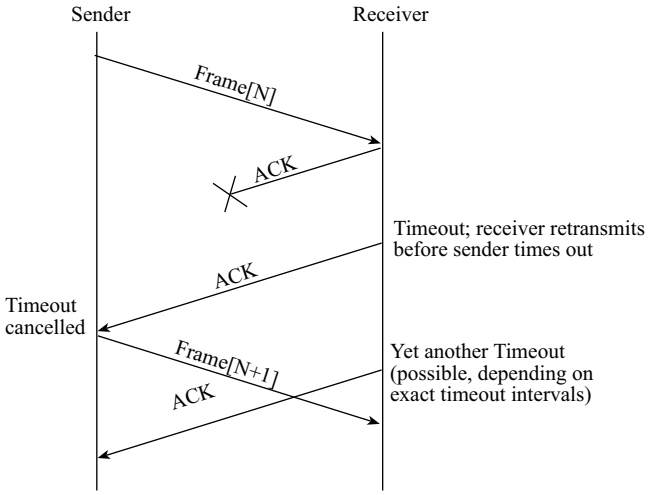
21. If the receiver delays sending an ACK until buffer space is available, it risks delaying so long that the sender times out unnecessarily and retransmits the frame.

22. For Fig 3.15(b) (lost frame), there are no changes from the diagram in the text.

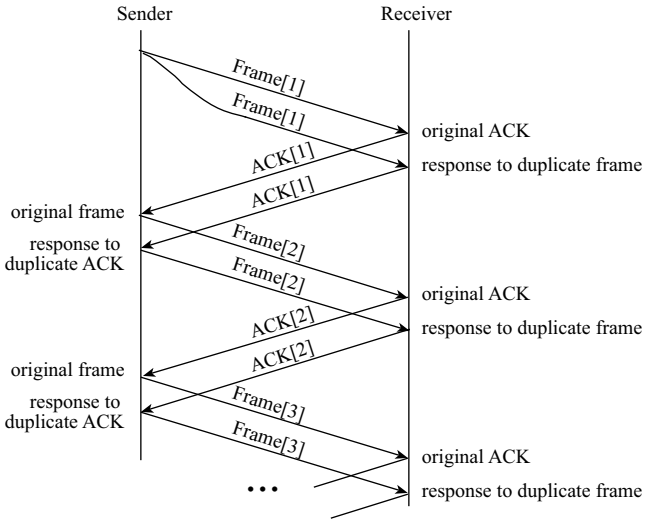
The next two figures correspond to the text's Fig 3.15(c) and (d); (c) shows a lost ACK and (d) shows an early timeout. For (c), the receiver timeout is shown slightly greater than (for definiteness) twice the sender timeout.



Here is the version of Fig 3.15(c) (lost ACK), showing a receiver timeout of approximately half the sender timeout.



23. (a) The duplications below continue until the end of the transmission.



(b) To trigger the sorcerer's apprentice phenomenon, a duplicate data frame must cross somewhere in the network with the previous ACK for that frame. If both sender and receiver adopt a resend-on-timeout strategy, *with the same timeout interval*, and an ACK is lost, then both sender and receiver will indeed retransmit at about the same time. Whether these retransmissions are synchronized enough that they cross in the network depends on other factors; it helps to have some modest latency delay or else slow hosts. With the right conditions, however, the sorcerer's apprentice phenomenon can be reliably reproduced.

24. The following is based on what TCP actually does: every ACK might (optionally or not) contain a value the sender is to use as a maximum for SWS. If this value is zero, the sender stops. A later ACK would then be sent with a nonzero SWS, when a receive buffer becomes available. Some mechanism would need to be provided to ensure that this later ACK is not lost, lest the sender wait forever. It is best if each new ACK reduces SWS by no more than 1, so that the sender's LFS never decreases.

Assuming the protocol above, we might have something like this:

T=0 Sender sends Frame1-Frame4. In short order, ACK1...ACK4 are sent setting SWS to 3, 2, 1, and 0 respectively. The Sender now waits for SWS>0.

T=1 Receiver frees first buffer; sends ACK4/SWS=1.
Sender slides window forward and sends Frame5.
Receiver sends ACK5/SWS=0.

T=2 Receiver frees second buffer; sends ACK5/SWS=1.
Sender sends Frame6; receiver sends ACK6/SWS=0.

T=3 Receiver frees third buffer; sends ACK6/SWS=1.
Sender sends Frame7; receiver sends ACK7/SWS=0.

T=4 Receiver frees fourth buffer; sends ACK7/SWS=1.
Sender sends Frame8; receiver sends ACK8/SWS=0.

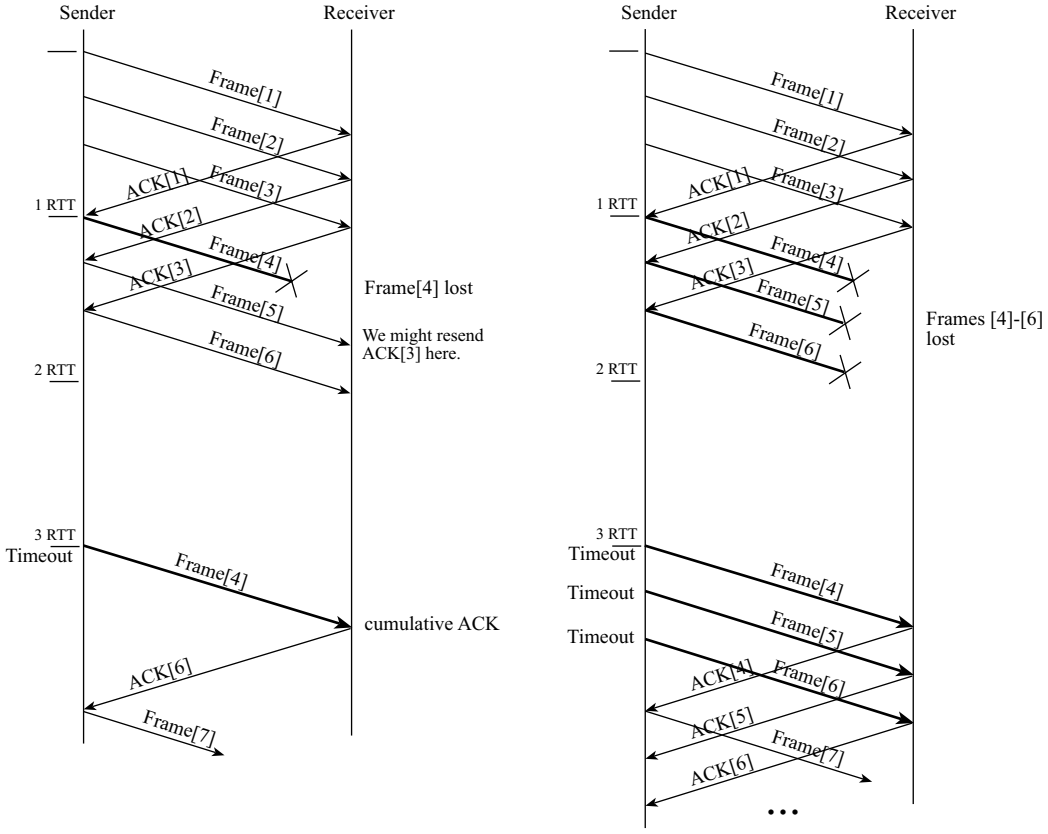
25. Here is one approach; variations are possible.

    If frame[N] arrives, the receiver sends ACK[N] if NFE=N; otherwise if N was in the receive window the receiver sends SACK[N].

    The sender keeps a bucket of values of N>LAR for which SACK[N] was received; note that whenever LAR slides forward this bucket will have to be purged of all N≤LAR.

    If the bucket contains one or two values, these could be attributed to out-of-order delivery. However, the sender might reasonably assume that whenever there was an N>LAR with frame[N] unacknowledged but with three, say, later SACKs in the bucket, then frame[N] was lost. (The number three here is taken from TCP with fast retransmit, which uses duplicate ACKs instead of SACKs.) Retransmission of such frames might then be in order. (TCP's fast-retransmit strategy would only retransmit frame[LAR+1].)

26. The right diagram, for part (b), shows each of frames 4-6 timing out after a 2×RTT timeout interval; a more realistic implementation (*eg* TCP) would probably revert to SWS=1 after losing packets, to address both congestion control and the lack of ACK clocking.



27. In the following, ACK[N] means that all packets with sequence number *less* than N have been received.

    1. The sender sends DATA[0], DATA[1], DATA[2]. All arrive.

2. The receiver sends ACK[3] in response, but this is slow. The receive window is now DATA[3]..DATA[5].

3. The sender times out and resends DATA[0], DATA[1], DATA[2]. For convenience, assume DATA[1] and DATA[2] are lost. The receiver accepts DATA[0] as DATA[5], because they have the same transmitted sequence number.

4. The sender finally receives ACK[3], and now sends DATA[3]-DATA[5]. The receiver, however, believes DATA[5] has already been received, when DATA[0] arrived, above, and throws DATA[5] away as a "duplicate". The protocol now continues to proceed normally, with one bad block in the received stream.

28. We first note that data below the sending window (that is, <LAR) is never sent again, and hence – because out-of-order arrival is disallowed – if DATA[N] arrives at the receiver then nothing at or before DATA[N-3] can arrive later. Similarly, for ACKs, if ACK[N] arrives then (because ACKs are cumulative) no ACK before ACK[N] can arrive later. As before, we let ACK[N] denote the acknowledgement of all data packets less than N.

    (a) If DATA[6] is in the receive window, then the earliest that window can be is DATA[4]-DATA[6]. This in turn implies ACK[4] was sent, and thus that DATA[1]-DATA[3] were received, and thus that DATA[0], by our initial remark, can no longer arrive.

    (b) If ACK[6] may be sent, then the lowest the sending window can be is DATA[3]..DATA[5]. This means that ACK[3] must have been received. Once an ACK is received, no smaller ACK can ever be received later.

29. (a) The smallest working value for MaxSeqNum is 8. It suffices to show that if DATA[8] is in the receive window, then DATA[0] can no longer arrive at the receiver. We have that DATA[8] in receive window
    $\Rightarrow$ the earliest possible receive window is DATA[6]..DATA[8]
    $\Rightarrow$ ACK[6] has been received
    $\Rightarrow$ DATA[5] was delivered.
    But because SWS=5, all DATA[0]'s sent were sent before DATA[5]
    $\Rightarrow$ by the no-out-of-order arrival hypothesis, DATA[0] can no longer arrive.

    (b) We show that if MaxSeqNum=7, then the receiver can be expecting DATA[7] and an old DATA[0] can still arrive. Because 7 and 0 are indistinguishable mod MaxSeqNum, the receiver cannot tell which actually arrived. We follow the strategy of Exercise 27.

    1. Sender sends DATA[0]...DATA[4]. All arrive.

    2. Receiver sends ACK[5] in response, but it is slow. The receive window is now DATA[5]..DATA[7].

    3. Sender times out and retransmits DATA[0]. The receiver accepts it as DATA[7].

    (c) MaxSeqNum $\geq$ SWS + RWS.

30. (a) Note that this is the canonical SWS = bandwidth$\times$delay case, with RTT = 4 sec. In the following we list the progress of one particular packet. At any given instant, there are four packets outstanding in various states.

T=N Data[N] leaves A
T=N+1 Data[N] arrives at R
T=N+2 Data[N] arrives at B; ACK[N] leaves
T=N+3 ACK[N] arrives at R
T=N+4 ACK[N] arrives at A; DATA[N+4] leaves.

Here is a specific timeline showing all packets in progress:

T=0 Data[0]...Data[3] ready; Data[0] sent
T=1 Data[0] arrives at R; Data[1] sent
T=2 Data[0] arrives at B; ACK[0] starts back; Data[2] sent
T=3 ACK[0] arrives at R; Data[3] sent
T=4 ACK[0] arrives at A; Data[4] sent
T=5 ACK[1] arrives at A; Data[5] sent ...

(b) T=0 Data[0]...Data[3] sent
T=1 Data[0]...Data[3] arrive at R
T=2 Data arrive at B; ACK[0]...ACK[3] start back
T=3 ACKs arrive at R
T=4 ACKs arrive at A; Data[4]...Data[7] sent
T=5 Data arrive at R

31. T=0 A sends frames 1-4. Frame[1] starts across the R–B link.
Frames 2,3,4 are in R's queue.

T=1 Frame[1] arrives at B; ACK[1] starts back; Frame[2] leaves R.
Frames 3,4 are in R's queue.

T=2 ACK[1] arrives at R and then A; A sends Frame[5] to R;
Frame[2] arrives at B; B sends ACK[2] to R.
R begins sending Frame[3]; frames 4,5 are in R's queue.

T=3 ACK[2] arrives at R and then A; A sends Frame[6] to R;
Frame[3] arrives at B; B sends ACK[3] to R;
R begins sending Frame[4]; frames 5,6 are in R's queue.

T=4 ACK[3] arrives at R and then A; A sends Frame[7] to R;
Frame[4] arrives at B; B sends ACK[4] to R.
R begins sending Frame[5]; frames 6,7 are in R's queue.

The steady-state queue size at R is two frames.

32. T=0 A sends frames 1-4. Frame[1] starts across the R–B link.
Frame[2] is in R's queue; *frames 3 & 4 are lost*.

T=1 Frame[1] arrives at B; ACK[1] starts back; Frame[2] leaves R.

T=2 ACK[1] arrives at R and then A; A sends Frame[5] to R.
R immediately begins forwarding it to B.
Frame[2] arrives at B; B sends ACK[2] to R.

T=3 ACK[2] arrives at R and then A; A sends Frame[6] to R.
R immediately begins forwarding it to B.
Frame[5] (not 3) arrives at B; B sends no ACK.

T=4 Frame[6] arrives at B; again, B sends no ACK.

T=5 A TIMES OUT, and retransmits frames 3 and 4.
R begins forwarding Frame[3] immediately, and enqueues 4.

T=6 Frame[3] arrives at B and ACK[3] begins its way back.
R begins forwarding Frame[4].

T=7 Frame[4] arrives at B and ACK[6] begins its way back.
ACK[3] reaches A and A then sends Frame[7].
R begins forwarding Frame[7].

33. Ethernet has a minimum frame size (64 bytes for 10Mbps; considerably larger for faster Ethernets); smaller packets are padded out to the minimum size. Protocols above Ethernet must be able to distinguish such padding from actual data.

34. One-way delays:

Coax:	1500m	6.49 μs
link:	1000m	5.13 μs
repeaters	two	1.20 μs
transceivers	six (two for each repeater, one for each station)	1.20 μs
drop cable	6×50m	1.54 μs
Total:		15.56 μs

The roundtrip delay is thus about 31.1 μs, or 311 bits. The "official" total is 464 bits, which when extended by 48 bits of jam signal exactly accounts for the 512-bit minimum packet size.

The 1982 Digital-Intel-Xerox specification presents a delay budget (page 62 of that document) that totals 463.8 bit-times, leaving 20 nanoseconds for unforeseen contingencies.

35. A station must not only detect a remote signal, but for collision detection it must detect a remote signal *while it itself is transmitting*. This requires much higher remote-signal intensity.

36. (a) Assuming 48 bits of jam signal was still used, the minimum packet size would be 4640+48 bits = 586 bytes.

    (b) This packet size is considerably larger than many higher-level packet sizes, resulting in considerable wasted bandwidth.

    (c) The minimum packet size could be smaller if maximum collision domain diameter were reduced, and if sundry other tolerances were tightened up.

37. (a) A can choose k_A=0 or 1; B can choose k_B=0,1,2,3. A wins outright if (k_A, k_B) is among (0,1), (0,2), (0,3), (1,2), (1,3); there is a 5/8 chance of this.

    (b) Now we have k_B among 0...7. If k_A=0, there are 7 choices for k_B that have A win; if k_A=1 then there are 6 choices. All told the probability of A's winning outright is 13/16.

    (c) P(winning race 1) = 5/8>1/2 and P(winning race 2) = 13/16>3/4; generalizing, we assume the odds of A winning the ith race exceed $(1 - 1/2^{i-1})$. We now have that

P(A wins every race given that it wins races 1-3)
$\geq (1 - 1/8)(1 - 1/16)(1 - 1/32)(1 - 1/64)$....
$\approx 3/4$.

(d) B gives up on it, and starts over with B_2.

38. (a) If A succeeds in sending a packet, B will get the next chance. If A and B are the only hosts contending for the channel, then even a wait of a fraction of a slot time would be enough to ensure alternation.

(b) Let A and B and C be contending for a chance to transmit. We suppose the following: A wins the first race, and so for the second race it defers to B and C for two slot times. B and C collide initially; we suppose B wins the channel from C one slot time later (when A is still deferring). When B now finishes its transmission we have the third race for the channel. B defers for this race; let us suppose A wins. Similarly, A defers for the fourth race, but B wins.

At this point, the backoff range for C is quite high; A and B however are each quickly successful – typically on their second attempt – and so their backoff ranges remain bounded by one or two slot times. As each defers to the other for this amount of time after a successful transmission, there is a strong probability that if we get to this point they will continue to alternate until C finally gives up.

(c) We might increase the backoff range given a decaying average of A's recent success rate.

39. Here is one possible solution; many, of course, are possible. The probability of four collisions appears to be quite low. Events are listed in order of occurrence.

A attempts to transmit; discovers line is busy and waits.

B attempts to transmit; discovers line is busy and waits.

C attempts to transmit; discovers line is busy and waits.

D finishes; A, B, and C all detect this, and attempt to transmit, and collide. A chooses k_A=1, B chooses k_B=1, and C chooses k_C=1.

One slot time later A, B, and C all attempt to retransmit, and again collide. A chooses k_A=2, B chooses k_B=3, and C chooses k_C=1.

One slot time later C attempts to transmit, and succeeds. While it transmits, A and B both attempt to retransmit but discover the line is busy and wait.

C finishes; A and B attempt to retransmit and a third collision occurs. A and B back off and (since we require a fourth collision) once again happen to choose the same $k < 8$.

A and B collide for the fourth time; this time A chooses k_A=15 and B chooses k_B=14.

14 slot times later, B transmits. While B is transmitting, A attempts to transmit but sees the line is busy, and waits for B to finish.

40. Many variations are, of course, possible. The scenario below attempts to demonstrate several plausible combinations.

D finishes transmitting.

First slot afterwards: all three defer (P=8/27).

Second slot afterwards: A,B attempt to transmit (and collide); C defers.

Third slot: C transmits (A and B are presumably backing off, although no relationship between p-persistence and backoff strategy was described).

C finishes.

First slot afterwards: B attempts to transmits and A defers, so B succeeds.

B finishes.

First slot afterwards: A defers.

Second slot afterwards: A defers.

Third slot afterwards: A defers.

Fourth slot afterwards: A defers a fourth time (P=16/81 $\approx 20\%$).

Fifth slot afterwards: A transmits.

A finishes.

41. (a) The second address must be distinct from the first, the third from the first two, and so on; the probability that none of the address choices from the second to the one thousandth collides with an earlier choice is

$$(1 - 1/2^{48})(1 - 2/2^{48}) \cdots (1 - 999/2^{48})$$

$$\approx 1 - (1 + 2 + ... + 999)/2^{48} = 1 - 999,000/(2 \times 2^{48}).$$

Probability of a collision is thus $999,000/(2 \times 2^{48}) \approx 1.77 \times 10^{-9}$. The denominator should probably be 2^{46} rather than 2^{48}, since two bits in an Ethernet address are fixed.

(b) Probability of the above on $2^{20} \approx 1$ million tries is 1.77×10^{-3}.

(c) Using the method of (a) yields $(2^{30})^2/(2 \times 2^{48}) = 2^{11}$; we are clearly beyond the valid range of the approximation. A better approximation, using logs, is presented in Exercise 8.18. Suffice it to say that a collision is essentially certain.

42. (a) Here is a sample run. The bold backoff-time binary digits were chosen by coin toss, with heads=1 and tails=0. Backoff times are then converted to decimal.

T=0: hosts A,B,C,D,E all transmit and collide. Backoff times are chosen by a single coin flip; we happened to get k_A=**1**, k_B=**0**, k_C=**0**, k_D=**1**, k_E=**1**. At the end of this first collision, T is now 1. B and C retransmit at T=1; the others wait until T=2.

T=1: hosts B and C transmit, immediately after the end of the first collision, and collide again. This time two coin flips are needed for each backoff; we happened to get $k_B = \mathbf{00} = 0$, $k_C = \mathbf{11} = 3$. At this point T is now 2; B will thus attempt again at T=2+0=2; C will attempt again at T=2+3=5.

T=2: hosts A,B,D,E attempt. B chooses a three-bit backoff time as it is on its third collision, while the others choose two-bit times. We got $k_A = \mathbf{10} = 2$, $k_B = \mathbf{010} = 2$, $k_D = \mathbf{01} = 1$, $k_E = \mathbf{11} = 3$. We add each k to T=3 to get the respective retransmission-attempt times: T=5,5,4,6.

T=3: Nothing happens.

T=4: Station D is the only one to attempt transmission; it successfully seizes the channel.

T=5: Stations A, B, and C sense the channel before transmission, but find it busy. E joins them at T=6.

(b) Perhaps the most significant difference on a real Ethernet is that stations close to each other will detect collisions almost immediately; only stations at extreme opposite points will need a full slot time to detect a collision. Suppose stations A and B are close, and C is far away. All transmit at the same time T=0. Then A and B will effectively start their backoff at T≈0; C will on the other hand wait for T=1. If A, B, and C choose the same backoff time, A and B will be nearly a full slot ahead.

Interframe spacing is only one-fifth of a slot time and applies to all participants equally; it is not likely to matter here.

43. Here is a simple program (also available on the web site):

```
#define USAGE "ether N"
// Simulates N ethernet stations all trying to transmit at once;
// returns average # of slot times until one station succeeds.

#include <iostream.h>
#include <stdlib.h>
#include <assert.h>

#define MAX 1000 /* max # of stations */

class station {
public:
    void reset() { _NextAttempt = _CollisionCount = 0;}
    bool transmits(int T) {return _NextAttempt==T;}
    void collide() { // updates station after a collision
        _CollisionCount ++;
        _NextAttempt += 1 + backoff(_CollisionCount);
            //the 1 above is for the current slot
    }
private:
    int _NextAttempt;
    int _CollisionCount;
    static int backoff(int k) {
    //choose random number 0..2^k-1; ie choose k random bits
        unsigned short r = rand();
        unsigned short mask = 0xFFFF >> (16-k); // mask = 2^k-1
        return int (r & mask);
    }
};
```

```
station S[MAX];

// run does a single simulation
// it returns the time at which some entrant transmits
int run (int N) {
    int time = 0;
    int i;
    for (i=0;i<N;i++) { S[i].reset(); }
    while(1) {
        int count = 0;    // # of attempts at this time
        int j= -1; // count the # of attempts; save j as index of one of them
        for (i=0; i<N; i++) {
            if (S[i].transmits(time)) {j=i; ++count;}
        }
        if (count==1) // we are done
            return time;
        else if (count>1) { // collisions occurred
            for (i=0;i<N;i++) {
                if (S[i].transmits(time)) S[i].collide();
            }
        }
        ++time;
    }
}

int RUNCOUNT = 10000;

void main(int argc, char * argv[]) {
    int N, i, runsum=0;
    assert(argc == 2);
    N=atoi(argv[1]);
    assert(N<MAX);
    for (i=0;i<RUNCOUNT;i++) runsum += run(N);
    cout << "runsum = " << runsum
        << " RUNCOUNT= " << RUNCOUNT
        << " average:  " << ((double)runsum)/RUNCOUNT << endl;
    return;
}
```

Here is some data obtained from it:

# stations	slot times
5	3.9
10	6.6
20	11.2
40	18.8
100	37.7
200	68.6

44. We alternate N/2 slots of wasted bandwidth with 5 slots of useful bandwidth. The useful fraction is: 5/(N/2 + 5) = 10/(N+10)

45. (a) The program is below (and on the web site). It produced the following output:

λ	# slot times
1	6.39577
1.1	5.78198
1.2	5.36019
1.3	5.05141
1.4	4.84586
1.5	4.69534
1.6	4.58546
1.7	4.50339
1.8	4.45381
1.9	4.43297
2	4.41884

λ	# slot times
2	4.41884
2.1	4.46704
2.2	4.4593
2.3	4.47471
2.4	4.49953
2.5	4.57311
2.6	4.6123
2.7	4.64568
2.8	4.71836
2.9	4.75893
3	4.83325

The minimum occurs at about λ=2; the theoretical value of the minimum is $2e - 1 = 4.43656$.

(b) If the contention period has length C, then the useful fraction is $8/(C + 8)$, which is about 64% for $C = 2e - 1$.

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

const int RUNCOUNT = 100000;

// X = X(lambda) is our random variable
double X(double lambda) {
    double u;
    do {
        u= double(rand())/RAND_MAX;
    } while (u== 0);
    double val = - log(u)*lambda;
    return val;
}

double run(double lambda) {
```

```
    int i = 0;
    double time = 0;
    double prevtime = -1;
    double nexttime = 0;
    time = X(lambda);
    nexttime = time + X(lambda);
    // while collision:  adjacent times within +/- 1 slot
    while (time - prevtime < 1 || nexttime - time < 1) {
        prevtime = time;
        time = nexttime;
        nexttime += X(lambda);
    }
    return time;
}

void main(int argc, char * argv[]) {
    int i;
    double sum, lambda;
    for (lambda = 1.0; lambda <= 3.01; lambda += 0.1) {
        sum = 0;
        for (i=0; i<RUNCOUNT; i++) sum += run(lambda);
        cout << lambda << "   " << sum/RUNCOUNT << endl;
    }
}
```

46. The sender of a frame normally removes it as the frame comes around again. The sender might either have failed (an orphaned frame), or the frame's source address might be corrupted so the sender doesn't recognize it.

    A monitor station fixes this by setting the monitor bit on the first pass; frames with the bit set (ie the corrupted frame, now on its second pass) are removed. The source address doesn't matter at this point.

47. 230m/(2.3×10^8m/sec) = 1 μs; at 16Mbps this is 16 bits. If we assume that each station introduces a minimum of 1 bit of delay, the the five stations add another five bits. So the monitor must add $24-(16+5)=3$ additional bits of delay. At 4Mbps the monitor needs to add $24-(4+5)=15$ more bits.

48. (a) THT/(THT + RingLatency)

    (b) Infinity; we let the station transmit as long as it likes.

    (c) TRT $\leq$ N $\times$ THT + RingLatency

49. At 4 Mbps it takes 2 ms to send a packet. A single active host would transmit for 2000 μs and then be idle for 200 μs as the token went around; this yields an efficiency of 2000/(2000+200) = 91%. Note that, because the time needed to transmit a packet exceeds the ring latency, immediate and delayed release here are the same.

At 100Mbps it takes 82 μs to send a packet. A single host would send for 82 μs, then wait a total of 200 μs from the start time for the packet to come round, then release the token and wait another 200 μs for the token to come back. Efficiency is thus 82/400 = 20%. With many hosts, each station would transmit about 200 μs apart, due to the wait for the delayed token release, for an efficiency of 82/200 $\approx$ 40%.

50. It takes a host 82 μs to send a packet. With immediate release, it sends a token upon completion; the earliest it can then transmit again is 200 μs later, when the token has completed a circuit. The station can thus transmit at most 82/282 = 29% of the time, for an effective bandwidth of 29Mbps.

    With delayed release, the sender waits 200 μs after beginning the transmission for the beginning of the frame to come around again; at this point the sender sends the token. The token takes another 200 μs to travel around before the original station could transmit again (assuming no other stations transmit). This yields an efficiency of 82/400 = 20%.

51. (a) $350 - 30 = 320$ μs is available for frame transmission, or 32,000 bits, or 4 KBytes. Divided among 10 stations this is 400 bytes each. (FDDI in fact lets stations with synchronous traffic divide up the allotments unequally, according to need.)

    (b) Here is a timeline, in which the latency between A,B,C is ignored. B transmits at T=0; the timeline goes back to T=-300 to allow TRT measurement.

T=-300	Token passes A,B,C
T=0	Token passes A; B seizes it
T=200	B finishes and releases token; C sees it C's measured_TRT is 500, too big to transmit.
T=500	Token returns to A,B,C A's measured_TRT: 500 B's measured_TRT: 500 C's measured_TRT: 300

    C may transmit next as its measured_TRT $<$ TTRT = 350. If all stations need to send, then this right to transmit will propagate round-robin down the ring.

Solutions for Chapter 3

1. The following table is cumulative; at each part the VCI tables consist of the entries at that part and also all previous entries.

Exercise part	Switch	Input		Output	
		Port	VCI	Port	VCI
(a)	1	2	0	1	0
	2	3	0	0	0
	3	0	0	3	0
(b)	1	3	0	1	1
	2	3	1	1	0
	4	3	0	1	0
(c)	2	2	0	0	1
	3	0	1	2	0
(d)	1	0	0	1	2
	2	3	2	0	2
	3	0	2	3	1
(e)	2	1	1	0	3
	3	0	3	1	0
	4	2	0	3	1
(f)	1	1	3	2	1
	2	1	2	3	3
	4	0	0	3	2

2. Node A:

Destination	Next hop
B	C
C	C
D	C
E	C
F	C

Node B:

Destination	Next hop
A	E
C	E
D	E
E	E
F	E

Node C:

Destination	Next hop
A	A
B	E
D	E
E	E
F	F

Node D:

Destination	Next hop
A	E
B	E
C	E
E	E
F	E

Node E:

Destination	Next hop
A	C
B	B
C	C
D	D
F	C

Node F:

Destination	Next hop
A	C
B	C
C	C
D	C
E	C

3. S1:

destination	port
A	1
B	2
default	3

S2:

destination	port
A	1
B	1
C	3
D	3
default	2

S3:

destination	port
C	2
D	3
default	1

S4:

destination	port
D	2
default	1

4. In the following, Si[j] represents the jth entry (counting from 1 at the top) for switch Si.

   A connects to D via S1[1]—S2[1]—S3[1]
   A connects to B via S1[2]
   A connects to E via S1[3]—S2[2]—S3[2]

5. We provide space in the packet header for a second address list, in which we build the return address. Each time the packet traverses a switch, the switch must add the *inbound* port number to this return-address list, in addition to forwarding the packet out the outbound port listed in the "forward" address. For example, as the packet traverses Switch 1 in Figure 3.7, towards forward address "port 1", the switch writes "port 2" into the return address. Similarly, Switch 2 must write "port 3" in the next position of the return address. The return address is complete once the packet arrives at its destination.

   Another possible solution is to assign each switch a locally unique name; that is, a name not shared by any of its directly connected neighbors. Forwarding switches (or the originating host) would then fill in the sequence of these names. When a packet was sent in reverse, switches would use these names to look up the previous hop. We might reject locally unique names, however, on the grounds that if interconnections can be added later it is hard to see how to permanently allocate such names without requiring global uniqueness.

   Note that switches cannot figure out the reverse route from the far end, given just the original forward address. The problem is that multiple senders might use the same forward address to reach a given destination; no reversal mechanism could determine to which sender the response was to be delivered. As an example, suppose Host A connects to port 0 of Switch 1, Host B connects to port 0 of Switch 2, and Host C connects to port 0 of Switch 3. Furthermore, suppose port 1 of Switch 1 connects to port 2 of Switch 3, and port 1 of Switch 2 connects to port 3 of Switch 3. The source-routing path from A to C *and* from B to C is (0,1); the reverse path from C to A is (0,2) and from C to B is (0,3).

6. Here is a proposal that entails separate actions for (a) the switch that lost state, (b) its immediate neighbors, and (c) everyone else. We will assume connections are bidirectional in that if a packet comes in on $\langle port_1, VCI_1 \rangle$ bound for $\langle port_2, VCI_2 \rangle$, then a packet coming in on the latter is forwarded to the former. Otherwise a reverse-lookup mechanism would need to be introduced.

   (a). A switch that has lost its state might send an *I am lost* message on its outbound links.

   (b). Immediate neighbors who receive this would identify the port through which the lost switch is reached, and then search their tables for any connection entries that use this port. A *connection broken* message would be sent out the *other* port of the connection entry, containing that port's corresponding VCI.

   (c). The remaining switches would then forward these *connection broken* messages back to the sender, forwarding them the usual way and updating the VCI on each link.

   A switch might not be aware that it has lost some or all of its state; one clue is that it receives a packet for which it was clearly expected to have state, but doesn't. Such a situation could, of course, also result from a neighbor's error.

7. If a switch loses its tables, it could notify its neighbors, but we have no means of identifying what hosts down the line might use that switch.

   So, the best we can do is notify senders by sending them an *unable to forward* message whenever a packet comes in to the affected switch.

8. We now need to keep a network address along with each outbound port (or with every port, if connections are bidirectional).

9. (a) The packet would be sent S1→S2→S3, the known route towards B. S3 would then send the packet back to S1 along the new connection, thinking it had forwarded it to B. The packet would continue to circulate.

   (b) This time it is the setup message itself that circulates forever.

10. Let us assume in Figure 3.37 that hosts H and J are removed, and that port 0 of Switch 4 is connected to port 1 of Switch 3. Here are the ⟨port,VCI⟩ entries for a path from Host E to host F that traverses the Switch2—Switch4 link twice; the VCI is 0 wherever possible.

    Switch 2: ⟨2,0⟩ to ⟨1,0⟩

    Switch 4: ⟨3,0⟩ to ⟨0,0⟩ (recall Switch 4 port 0 now connects to Switch 3)

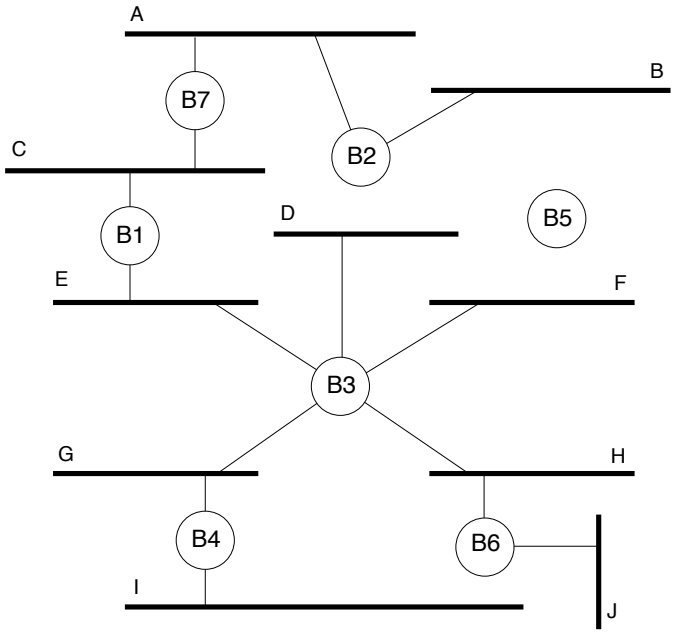
    Switch 3: ⟨1,0⟩ to ⟨0,0⟩

    Switch 2: ⟨0,0⟩ to ⟨1,1⟩

    Switch 4: ⟨3,1⟩ to ⟨2,0⟩

11. There is no guarantee that data sent along the circuit won't catch up to and pass the process establishing the connections, so, yes, data should not be sent until the path is complete.

12.



13. When A sends to C, all bridges see the packet and learn where A is. However, when C then sends to A, the packet is routed directly to A and B4 does not learn where C is. Similarly, when D sends to C, the packet is routed by B2 towards B1 only, and B1 does not learn where D is.

B1:	A-interface:	A	B2-interface:	C (not D)		
B2:	B1-interface:	A	B3-interface:	C	B4-interface:	D
B3:	B2-interface:	A,D	C-interface:	C		
B4:	B2-interface:	A (not C)	D-interface:	D		

14. (a) When X sends to Z the packet is forwarded on all links; all bridges learn where X is. Y's network interface would see this packet.

    (b) When Z sends to X, all bridges already know where X is, so each bridge forwards the packet only on the link towards X, that is, B3→B2→B1→X. Since the packet traverses all bridges, all bridges learn where Z is. Y's network interface would not see the packet as B2 would only forward it on the B1 link.

    (c) When Y sends to X, B2 would forward the packet to B1, which in turn forwards it to X. Bridges B2 and B1 thus learn where Y is. B3 and Z never see the packet.

    (d) When Z sends to Y, B3 does not know where Y is, and so retransmits on all links; W's network interface would thus see the packet. When the packet arrives at B2, though, it is retransmitted only to Y (and not to B1) as B2 does know where Y is from step (c). All bridges already knew where Z was, from step (b).

15. B1 will be the root; B2 and B3 each have two equal length paths (along their upward link and along their downward link) to B1. They will each, independently, select one of these vertical links to use (perhaps preferring the interface by which they first heard from B1), and disable the other. There are thus four possible solutions.

16. (a) The packet will circle endlessly, in both the M→B2→L→B1 and M→B1→L→B2 directions.

    (b) Initially we (potentially) have four packets: one from M clockwise, one from M counterclockwise, and a similar pair from L.

    Suppose a packet from L arrives at an interface to a bridge Bi, followed immediately via the same interface by a packet from M. As the first packet arrives, the bridge adds ⟨L,arrival-interface⟩ to the table (or, more likely, updates an existing entry for L). When the second packet arrives, addressed to L, the bridge then decides not to forward it, because it arrived from the interface recorded in the table as pointing towards the destination, and so it dies.

    Because of this, we expect that in the long run only one of the pair of packets traveling in the same direction will survive. We may end up with two from M, two from L, or one from M and one from L. A specific scenario for the latter is as follows, where the bridges' interfaces are denoted "top" and "bottom":

    1. L sends to B1 and B2; both place ⟨L,top⟩ in their table. B1 already has the packet from M in the queue for the top interface; B2 this packet in the queue for the bottom.

    2. B1 sends the packet from M to B2 via the top interface. Since the destination is L and ⟨L,top⟩ is in B2's table, it is dropped.

    3. B2 sends the packet from M to B1 via the bottom interface, so B1 updates its table entry for M to ⟨M,bottom⟩

    4. B2 sends the packet from L to B1 via the bottom interface, causing it to be dropped.

    The packet from M now circulates counterclockwise, while the packet from L circulates clockwise.

17. (a) In this case the packet would never be forwarded; as it arrived from a given interface the bridge would first record ⟨M,interface⟩ in its table and then conclude the packet destined for M did not have to be forwarded out the other interface.

    (b) Initially we would have a copy of the packet circling clockwise (CW) and a copy circling counterclockwise (CCW). This would continue as long as they traveled in perfect symmetry, with each bridge seeing alternating arrivals of the packet through the top and bottom interfaces. Eventually, however, something like the following is likely to happen:

    0. Initially, B1 and B2 are ready to send to each other via the top interface; both believe M is in the direction of the bottom interface.

    1. B1 starts to send to B2 via the top interface (CW); the packet is somehow delayed in the outbound queue.

    2. B2 does send to B1 via the top interface (CCW).

    3. B1 receives the CCW packet from step 2, and immediately forwards it over the bottom interface back to B2. The CW packet has not yet been delivered to B2.

    4. B2 receives the packet from step 3, via the bottom interface. Because B2 currently believes that the destination, M, lies on the bottom interface, B2 drops the packet.

    The clockwise packet would then be dropped on its next circuit, leaving the loop idle.

18. (a) If the bridge forwards all spanning-tree messages, then the remaining bridges would see networks D,E,F,G,H as a single network. The tree produced would have B2 as root, and would disable the following links:

    from B5 to A (the D side of B5 has a direct connection to B2)
    from B7 to B
    from B6 to either side

    (b) If B1 simply drops the messages, then as far as the spanning-tree algorithm is concerned the five networks D-H have no direct connection, and in fact the entire extended LAN is partitioned into two disjoint pieces A-F and G-H. Neither piece has any redundancy alone, so the separate spanning trees that would be created would leave all links active. Since bridge B1 still presumably *is* forwarding other messages, all the original loops would still exist.

19. (a) Whenever any host transmits, the packet collides with itself.

    (b) It is difficult or impossible to send status packets, since they too would self-collide as in (a). Repeaters do not look at a packet before forwarding, so they wouldn't be in a position to recognize status packets as such.

    (c) A hub might notice a loop because collisions *always* occur, whenever any host transmits. Having noticed this, the hub might send a specific signal out one interface, during the rare idle moment, and see if that signal arrives back via another. The hub might, for example, attempt to verify that whenever a signal went out port 1, then a signal always appeared immediately at, say, port 3.

    We now wait some random time, to avoid the situation where a neighboring hub has also noticed the loop and is also disabling ports, and if the situation still persists we disable one of the looping ports.

    Another approach altogether might be to introduce some distinctive signal that does not correspond to the start of any packet, and use this for hub-to-hub communication.

20. Once we determine that two ports *are* on the same LAN, we can choose the smaller-numbered port and shut off the other.

    A bridge will know it has two interfaces on the same LAN when it sends out its initial "I am root" configuration messages and receives its own messages back, without their being marked as having passed through another bridge.

21. A 53-byte ATM cell has 5 bytes of headers, for an overhead of about 9.4% for ATM headers alone.

    When a 512-byte packet is sent via AAL3/4, we first encapsulate it in a 520-byte CS-PDU. This is then segmented into eleven 44-byte pieces and one trailing 36-byte piece. These in turn are encapsulated into twelve ATM cells, each of which having 9 bytes of ATM+AAL3/4 headers. This comes to $9 \times 12 = 108$ bytes of header overhead, plus the 8 bytes added to the CS-PDU, plus $44 - 36 = 8$ bytes of padding for the last cell. The total overhead is 124 bytes, which we could also have arrived at as $12 \times 53 - 512$; as a percentage this is 124/(512+124) = 19.5%.

When the packet is sent via AAL5, we first form the CS-PDU by appending 8 AAL5 trailer bytes, preceded by another 8 bytes of padding. We then segment into *eleven* cells, for a total overhead of $8 + 8 + 11 \times 5 = 71$ bytes, or 71/(512+71) = 12.1%.

22. AAL3/4 has a 4-bit sequence in each cell; this number wraps around after 16 cells. AAL3/4 provides only a per-cell CRC, which wouldn't help with lost cells; there is no CRC over the entire CS-PDU.

23. The length of the AAL5 CS-PDU into which the ACK is encapsulated is exactly 48 bytes, and this fits into a single ATM cell. When AAL3/4 is used the CS-PDU is again 48 bytes, but now the per-cell payload is only 44 bytes and two cells are necessary.

24. For AAL5, the number of cells needed to transmit a packet of size x is $(x + 8)/48$ rounded up to the nearest integer, or $\lceil (x + 48)/48 \rceil$. This rounding is essentially what the padding does; it represents space that would be needed anyway when fitting the final segment of the CS-PDU into a whole ATM cells. For AAL3/4 it takes $\lceil (x + 8)/44 \rceil$ cells, again rounded up. The CS-PDU pad field is only to make the CS-PDU aligned on 32-bit boundaries; additional padding is still needed to fill out the final cell.

25. If x is the per-cell loss rate, and is very small, then the loss rate for 20 cells is about $20x$. The loss rate would thus have to be less than 1 in 20×10^6.

26. Let p be the probability one cell is lost. We want the probability of losing two (or more) cells. We apply the method of the probability sidebar in Section 2.4. Among 21 cells there are $21 \times 22/2$ = 231 pairs of cells; the probability of losing any specific pair is p^2. So, the probability of losing an arbitrary pair is about $231p^2$. Setting this equal to 10^{-6} and solving for p, we get $p \approx 1/15200$, or about 66 lost packets per million.

27. (a) The probability AAL3/4 fails to detect two errors, given that they occur, is about $1/2^{20}$. For three errors in three cells this is $1/2^{30}$. Both are less than the CRC-32 failure rate of $1/2^{32}$.

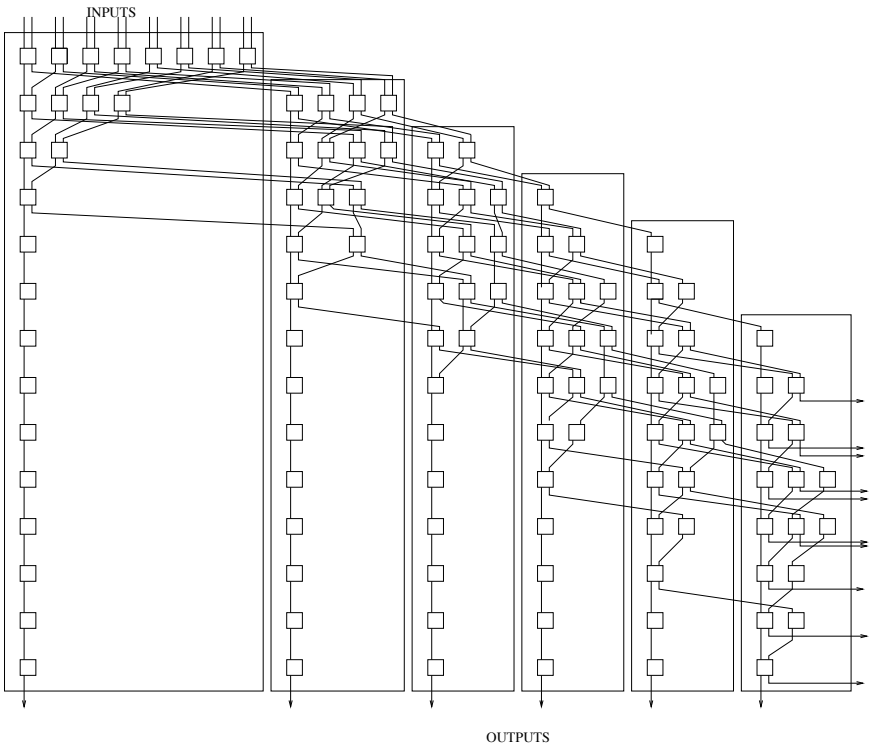
    (b) AAL3/4 would be more likely to detect errors if most errors affected four or more cells. This could be due to errors coming in large bursts.

28. The drawbacks to datagram routing for small cells are the larger addresses, which would now take up a considerable fraction of each cell, and the considerably higher per-cell processing costs in each router that are not proportional to cell size.

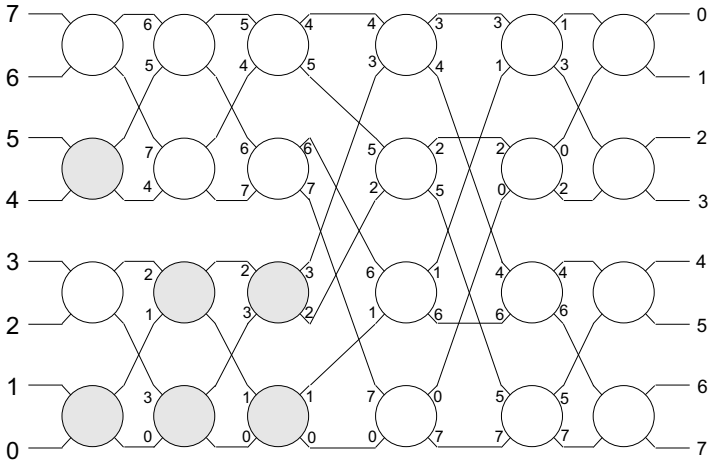
29. Since the I/O bus speed is less than the memory bandwidth, it is the bottleneck. Effective bandwidth that the I/O bus can provide is 800/2 Mbps because each packet crosses the I/O bus twice. Threfore, the number of interfaces is $\lfloor 400/45 \rfloor = 8$.

30. The workstation can handle 800/2 Mbps, as in the previous Exercise. Let the packet size be x bits; to support 100,000 packets/second we need a total capacity of $100000 \times x$ bps; equating $10^5 \times x = 400 \times 10^6$ bps, we get x = 4,000 bits/sec = 500 bytes/sec.

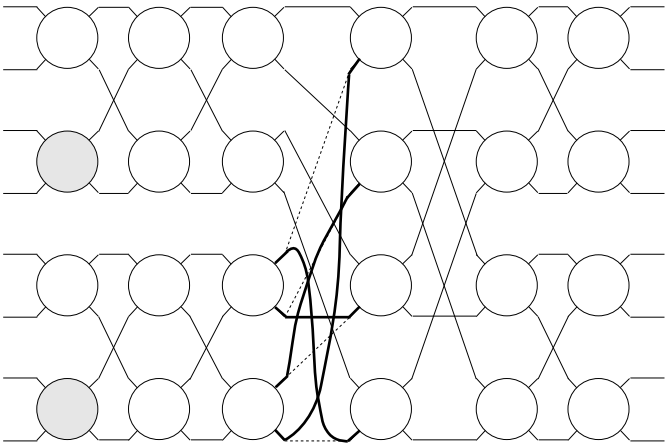
31.



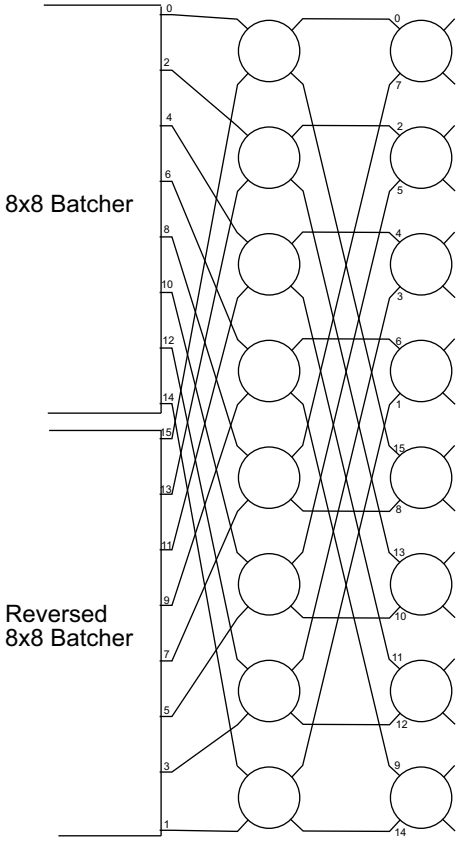
32.



33. We simply connect the output wires from the bottom-left 4×4 sub-Batcher sorter (the bottom-left 2×3 block of switching elements) in the reverse order of the way they were originally connected to the merger section. The unreversed sub-Batcher sorts its output in increasing order; by reversing the wires this output enters column 4 in decreasing order as desired.



34. The following diagram is labeled to show the merging of the inputs $\langle 0, 2, 4, 6, 8, 10, 12, 14\rangle$ from the upper-left sub-Batcher and $\langle 1, 3, 5, 7, 9, 11, 13, 15\rangle$ (reversed) from the lower-left sub-Batcher.



35. Each stage has $n/2$ switching elements. Since after each stage we eliminate half the network, *ie* half the rows in the $n \times n$ network, we need $\log_2 n$ stages. Therefore the number of switching elements needed is $(n/2)\log_2 n$. For $n = 8$, this is 12.

36. Let T_k be the number of switching elements required for an $n \times n$ Batcher network with

$n = 2^k$; we are assuming $T_1 = 1$. We have the following recurrence relation:

$$T_k = 2T_{k-1} + k2^{k-1}$$

that is, we have two recursive sorters of size $k-1 \times k-1$, and then k columns of 2^{k-1} rows for the merging portion. Reversing and expanding, we get

$$\begin{aligned}
T_k &= k2^{k-1} + 2((k-1)2^{k-2} + 2T_{k-2}) \\
&= k2^{k-1} + (k-1)2^{k-1} + 2^2T_{k-2} \\
&= k2^{k-1} + (k-1)2^{k-1} + (k-2)2^{k-1} + 2^3T_{k-3} \\
&= k2^{k-1} + (k-1)2^{k-1} + \cdots + 2 \times 2^{k-1} + 2^{k-1}T_1 \\
&= k2^{k-1} + (k-1)2^{k-1} + \cdots + 2 \times 2^{k-1} + 2^{k-1} \\
&= 2^{k-1}(k + (k-1) + \cdots + 1) \\
&= 2^{k-2}k(k+1)
\end{aligned}$$

We can then verify that this formula does in fact satisfy the recurrence relation.

For k=3, n=8, we get T_k=24.

37. (a) The probability that one random connection takes the link is about 1/2. So the probability that two each take the link is about $(1/2)^2 = 1/4$, and the probability that at most one takes the link is thus $1 - 1/4 = 3/4$.

    (b) P(no connection uses the link) = 1/8 and P(exactly one connection uses the link) = 3/8; these are equivalent to, if three coins are flipped, P(no heads) = 1/8 and P(exactly one head) = 3/8. The total probability that the link is not oversubscribed is 1/2.

38. (a) After the upgrade the server—switch link is the only congested link. For a busy Ethernet the contention interval is roughly proportional to the number of stations contending, and this has now been reduced to two. So performance should increase, but only slightly.

    (b) Both token ring and a switch allow nominal 100% utilization of the bandwidth, so differences should be negligible. The only systemic difference might be that with the switch we no longer have ring latency to worry about, but for rings that are enclosed in hubs, this should be infinitesimal.

    (c) A switch makes it impossible for a station to eavesdrop on traffic not addressed to it. On the other hand, switches tend to cost more than hubs, per port.

Solutions for Chapter 4

1. IP addresses include the network/subnet, so that interfaces on different networks must have different network portions of the address. Alternatively, addresses include location information and different interfaces are at different locations, topologically.

   Point-to-point interfaces can be assigned a duplicate address (or no address) because the other endpoint of the link doesn't use the address to reach the interface; it just sends. Such interfaces, however, cannot be addressed by any other host in the network. See also RFC1812, section 2.2.7, page 25, on "unnumbered point-to-point links".

2. The IPv4 header allocates only 13 bits to the Offset field, but a packet's length can be up to $2^{16}-1$. In order to support fragmentation of a maximum-sized packet, we must count offsets in multiples of $2^{16-13} = 2^3$ bytes.

   The only concerns with counting fragmentation offsets in 8-byte units are that we would waste space on a network with MTU $= 8n+7$ bytes, or that alignment on 8-byte boundaries would prove inconvenient. 8-byte chunks are small enough that neither of these is a significant concern.

3. Consider the first network. Packets have room for $1024 - 14 - 20 = 990$ bytes of IP-level data; because 990 is not a multiple of 8 each fragment can contain at most $8 \times \lfloor 990/8 \rfloor = 984$ bytes. We need to transfer $2048 + 20 = 2068$ bytes of such data. This would be fragmented into fragments of size 984, 984, and 100.

   Over the second network (which by the way has an illegally small MTU for IP), the 100-byte packet would be unfragmented but the 984-data-byte packet would be fragmented as follows. The network+IP headers total 28 bytes, leaving $512-28 = 484$ bytes for IP-level data. Again rounding down to the nearest multiple of 8, each fragment could contain 480 bytes of IP-level data. 984 bytes of such data would become fragments with data sizes 480, 480, and 24.

4. (a) The probability of losing both transmissions of the packet would be $0.1 \times 0.1 = 0.01$.

   (b) The probability of loss is now the probability that for some pair of identical fragments, both are lost. For any particular fragment the probability of losing both instances is $0.01 \times 0.01 = 10^{-4}$, and the probability that this happens at least once for the 10 different fragments is thus about 10 times this, or 0.001.

   (c) An implementation *might* (though generally most do not) use the same value for Ident when a packet had to be transmitted. If the retransmission timeout was less than the reassembly timeout, this might mean that case (b) applied and that a received packet might contain fragments from each transmission.

5.

M	offset	bytes data	source
1	0	360	1st original fragment
1	360	152	1st original fragment
1	512	360	2nd original fragment
1	872	152	2nd original fragment
1	1024	360	3rd original fragment
0	1384	16	3rd original fragment

If fragmentation had been done originally for this MTU, there would be four fragments. The first three would have 360 bytes each; the last would have 320 bytes.

6. The Ident field is 16 bits, so we can send 576×2^{16} bytes per 60 seconds, or about 5Mbps. If we send more than this, then fragments of one packet could conceivably have the same Ident value as fragments of another packet.

7. TCP/IP moves the error-detection field to the transport layer; the IP header checksum doesn't cover data and hence isn't really an analogue of CRC-10. Similarly, because ATM fragments must be received in sequence there is no need for an analogue of IP's Offset field.

Btag/Etag	Prevents frags of different PDUs from running together, like Ident
BAsize	No direct analogue; not applicable to IP
Len	Length of original PDU; no IP analogue
Type	corresponds more or less to IP Flags
SEQ	no analogue; IP accepts out-of-order delivery
MID	no analogue; MID permits multiple packets on one VC
Length	the size of this fragment, like IP Length field

8. IPv4 effectively requires that, if reassembly is to be done at the downstream router, then it be done at the link layer, and will be transparent to IPv4. IP-layer fragmentation is only done when such link-layer fragmentation isn't practical, in which case IP-layer reassembly might be expected to be even less practical, given how busy routers tend to be. See RFC791, page 23.

   IPv6 uses link-layer fragmentation exclusively; experience had by then established reasonable MTU values, and also illuminated the performance problems of IPv4-style fragmentation. (TCP path-MTU discovery is also mandatory, which means the sender always knows just how large TCP segments can be to avoid fragmentation.)

   Whether or not link-layer fragmentation is feasible appears to depend on the nature of the link; neither version of IP therefore requires it.

9. If the timeout value is too small, we clutter the network with unnecessary re-requests, and halt transmission until the re-request is answered.

   When a host's Ethernet address changes, *eg* because of a card replacement, then that host is unreachable to others that still have the old Ethernet address in their ARP cache. 10-15 minutes is a plausible minimal amount of time required to shut down a host, swap its Ethernet card, and reboot.

   While self-ARP (described in the following exercise) is arguably a better solution to the problem of a too-long ARP timeout, coupled with having other hosts update their caches whenever they see an ARP query from a host already in the cache, these features were not always universally implemented. A reasonable upper bound on the ARP cache timeout is thus necessary as a backup.

10. After B broadcasts any ARP query, all stations that had been sending to A's physical address will switch to sending to B's. A will see a sudden halt to all arriving traffic. (To guard against

this, A might monitor for ARP broadcasts purportedly coming from itself; A might even immediately follow such broadcasts with its own ARP broadcast in order to return its traffic to itself. It is not clear, however, how often this is done.)

If B uses self-ARP on startup, it will receive a reply indicating that its IP address is already in use, which is a clear indication that B should not continue on the network until the issue is resolved.

11. (a) If multiple packets after the first arrive at the IP layer for outbound delivery, but before the first ARP response comes back, then we send out multiple unnecessary ARP packets. Not only do these consume bandwidth, but, because they are broadcast, they interrupt every host and propagate across bridges.

    (b) We should maintain a list of currently outstanding ARP queries. Before sending a query, we first check this list. We also might now retransmit queries on the list after a suitable timeout.

    (c) This might, among other things, lead to frequent and excessive packet loss at the beginning of new connections.

12. (a)

Information Stored at Node	Distance to Reach Node					
	A	B	C	D	E	F
A	0	∞	3	8	∞	∞
B	∞	0	∞	∞	2	∞
C	3	∞	0	∞	1	6
D	8	∞	∞	0	2	∞
E	∞	2	1	2	0	∞
F	∞	∞	6	∞	∞	0

(b)

Information Stored at Node	Distance to Reach Node					
	A	B	C	D	E	F
A	0	∞	3	8	4	9
B	∞	0	3	4	2	∞
C	3	3	0	3	1	6
D	8	4	3	0	2	∞
E	4	2	1	2	0	7
F	9	∞	6	∞	7	0

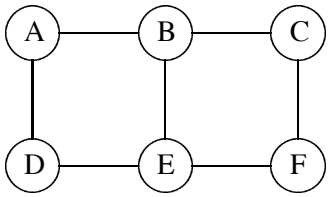
(c)

Information Stored at Node	Distance to Reach Node					
	A	B	C	D	E	F
A	0	6	3	6	4	9
B	6	0	3	4	2	9
C	3	3	0	3	1	6
D	6	4	3	0	2	9
E	4	2	1	2	0	7
F	9	9	6	9	7	0

13.

D	Confirmed	Tentative
1.	(D,0,-)	
2.	(D,0,-)	(A,8,A)
		(E,2,E)
3.	(D,0,-)	(A,8,A)
	(E,2,E)	(B,4,E)
		(C,3,E)
4.	(D,0,-)	(A,6,E)
	(E,2,E)	(B,4,E)
	(C,3,E)	(F,9,E)
5.	(D,0,-)	(A,6,E)
	(E,2,E)	(F,9,E)
	(C,3,E)	
	(B,4,E)	
6.	previous + (A,6,E)	
7.	previous + (F,9,E)	

14. The cost=1 links show A connects to B and D; F connects to C and E.
F reaches B through C at cost 2, so B and C must connect.
F reaches D through E at cost 2, so D and E must connect.
A reaches E at cost 2 through B, so B and E must connect.
These give:



As this network is consistent with the tables, it is the unique minimal solution.

15. (a)

A:

dest	cost	nexthop
B	∞	-
C	3	C
D	∞	-
E	∞	-
F	9	C

B:

dest	cost	nexthop
A	∞	-
C	∞	-
D	4	E
E	2	E
F	∞	-

D:

dest	cost	nexthop
A	∞	-
B	4	E
C	∞	-
E	2	E
F	∞	-

F:

dest	cost	nexthop
A	9	C
B	∞	-
C	6	C
D	∞	-
E	∞	-

(b) A:

dest	cost	nexthop
B	12	D
C	3	C
D	8	D
E	10	D
F	9	C

D:

dest	cost	nexthop
A	8	A
B	4	E
C	11	A
E	2	E
F	17	A

(c) C:

dest	cost	nexthop
A	3	A
B	15	A
D	11	A
E	13	A
F	6	F

16. Apply each subnet mask and if the corresponding subnet number matches the SubnetNumber column, then use the entry in Next-Hop. (In these tables there is always a unique match.)

    (a) Applying the subnet mask 255.255.255.128, we get 128.96.39.0. Use interface0 as the next hop.

    (b) Applying subnet mask 255.255.255.128, we get 128.96.40.0. Use R2 as the next hop.

    (c) All subnet masks give 128.96.40.128 as the subnet number. Since there is no match, use the default entry. Next hop is R4.

    (d) Next hop is R3.

    (e) None of the subnet number entries match, hence use default router R4.

17. (a) A necessary and sufficient condition for the routing loop to form is that B reports to A the networks B believes it can currently reach, after A discovers the problem with the A—E link, but before A has communicated to B that A no longer can reach E.

    (b) At the instant that A discovers the A—E failure, there is a 50% chance that the next report will be B's and a 50% chance that the next report will be A's. If it is A's, the loop will not form; if it is B's, it will.

    (c) At the instant A discovers the A—E failure, let t be the time until B's next broadcast. t is equally likely to occur anywhere in the interval $0 \leq t \leq 60$. The event of a loop forming is the same as the event that B broadcasts first, which is the event that $t < 1.0$ sec; the probability of this is 1/60.

18. Denote the act of A's sending an update to B about E by A⇒B. Any initial number of B⇒C or C⇒B updates don't change E entries. By split horizon, B⇒A and C⇒A are disallowed. Since we have assumed A reports to B before C, the first relevant report must be A⇒B. This makes C the sole believer in reachability of E; C's table entry for E remains (E,2,A).

    At this point legal and relevant updates are A⇒C, C⇒B, and B⇒C; A⇔B exchanges don't change E entries and C⇒A is disallowed by split horizon. If A⇒C or B⇒C the loop formation is halted, so we require C⇒B. Now C's table has (E,2,A) and B's has (E,3,C); we have two believers.

The relevant possibilities now are B⇒A, or A⇒C. If B⇒A, then A's table has (E,4,C) and the loop is complete. If A⇒C, then B becomes the sole believer. The only relevant update at that point not putting an end to belief in E is B⇒A, which then makes A a believer as well.

At this point, exchange A⇒C would then form the loop. On the other hand, C⇒B would leave A the sole believer. As things progress, we could either

(a) form a loop at some point,
(b) eliminate all belief in E at some point, or
(c) have sole-believer status migrate around the loop, C→B→A→C→ ⋯,
alternating with the dual-believer situation.

19. (b) Without poison reverse, A and B would send each other updates that simply didn't mention X; presumably (this does depend somewhat on implementation) this would mean that the false routes to X would sit there until they eventually aged out. With poison reverse, such a loop would go away on the first table update exchange.

(c) 1. B and A each send out announcements of their route to X via C to each other.
2. C announces to A and B that it can no longer reach X; the announcements of step 1 have not yet arrived.
3. B and A receive each others announcements from step 1, and adopt them.

20. We will implement hold-down as follows: when an update record arrives that indicates a destination is unreachable, all subsequent updates within some given time interval are ignored and discarded.

Given this, then in the EAB network A ignores B's reachability news for one time interval, during which time A presumably reaches B with the correct unreachability information.

Unfortunately, in the EABD case, this also means A ignores the valid B–D–E path. Suppose, in fact, that A reports its failure to B, D reports its valid path to B, and then B reports to A, all in rapid succession. This new route will be ignored.

One way to avoid delaying discovery of the B–D–E path is to keep the hold-down time interval as short as possible, relying on triggered updates to spread the unreachability news quickly.

Another approach to minimizing delay for new valid paths is to retain route information received during the hold-down period, but not to use it. At the expiration of the hold-down period, the sources of such information might be interrogated to determine whether it remains valid. Otherwise we might have to wait not only the hold-down interval but also wait until the next regular update in order to receive the new route news.

21. There is some confusion in the last paragraph of this exercise. OSPF routers send out *one* LSP, with one sequence number, that describes all the router's connections; however, the language "both ends of a link use the same sequence number in their LSP for that link" incorrectly suggests that routers send out a different LSP (or at least different LSA) for each link, each with its own sequence number. While it is certainly possible for link-state routing to take this approach, it is not how OSPF works and it is not what the text describes. We will use OSPF-style numbering here. We will also assume that each node increments its sequence number only when there is some change in the state of its local links, not for timer expirations ("no packets time out").

The central *point* of this exercise was intended to be an illustration of the "bringing-up-adjacencies" process: in restoring the connection between the left- and righthand networks, it is not sufficient simply to flood the information about the restored link. The two halves have evolved separately, and full information must be exchanged.

Given that each node increments its sequence number whenever it detects a change in its links to its neighbors, at the instant before the B—F link is restored the LSP data for each node is as follows:

node	seq#	connects to
A	2	B,C,D
B	2	A,C
C	2	A,B,D
D	2	A,C
F	2	G
G	2	F,H
H	1	G

When the B–F link is restored, OSPF has B and F exchange their full databases of all the LSPs they have seen with each other. Each then floods the other side's LSPs throughout its side of the now-rejoined network. These LSPs are as in the rows of the table above, except that B and F now each have sequence numbers of 3. (Had we assigned separate sequence numbers to each individual link, every sequence number would be 1 except for link B—F.)

The initial sequence number of an OSPF node is actually $-2^{31} + 1$.

22.

Step	confirmed	tentative
1	(A,0,-)	
2	(A,0,-)	(D,2,D) (B,5,B)
3	(A,0,-) (D,2,D)	(B,4,D) (E,7,D)
4	(A,0,-) (D,2,D) (B,4,D)	(E,6,D) (C,8,D)
5	(A,0,-) (D,2,D) (B,4,D) (E,6,D)	(C,7,D)
6	(A,0,-) (D,2,D) (B,4,D) (E,6,D) (C,7,D)	

23. (a) This could happen if the link changed state recently, and one of the two LSP's was old.

    (b) If flooding is working properly, and if A and B do in fact agree on the state of the link, then eventually (rather quickly) whichever of the two LSP's was old would be updated by the same sender's newer version, and reports from the two sides of C would again agree.

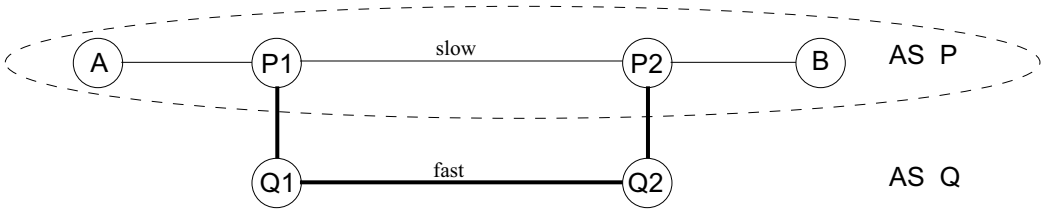
24. (a) Q will receive three routes to P, along links 1, 2, and 3.

    (b) A⟶B traffic will take link 1. B⟶A traffic will take link 2. Note that this strategy minimizes cost to the source of the traffic.

    (c) To have B⟶A traffic take link 1, Q could simply be configured to prefer link 1 in all cases. The only general solution, though, is for Q to accept into its routing tables some of the internal structure of P, so that Q for example knows where A is relative to links 1 and 2.

(d) If Q were configured to prefer AS paths through R, or to avoid AS paths involving links 1 and 2, then Q might route to P via R.

25. In the diagram below, autonomous system P contains A, P1, P2, and B; autonomous system Q contains Q1 and Q2. P and Q have long parallel links. P is provider for A and B, but Q's long link is much faster:



If we choose weights appropriately for the P1–P2 and Q1–Q2 links, we can have the optimum route be A–P1–Q1–Q2–P2–B; the AS_PATH would then be P–Q–P. To BGP, such an AS_PATH would appear as a loop, and be disallowed.

26. (a) The diameter D of a network organized as a binary tree, with root node as "backbone", would be of order $\log_2 A$. The diameter of a planar rectangular grid of connections would be of order $\sqrt{A}$.

(b) For each AS S, the BGP node needs to maintain a record of the AS_PATH to S, requiring 2×actual_path_length bytes. It also needs a list of all the networks within S, requiring 4×number_of_networks bytes. Summing these up for all autonomous systems, we get $2AD+4N$, or $2AC\log A+4N$ and $2AC\sqrt{A}+4N$ for the models from part (a), where C is a constant.

27. This exercise does not, alas, quite live up to its potential.

The central idea behind Ethernet bridges is that they learn new host locations by examining ordinary data packets, and do *not* receive new-host notices from other bridges. Thus the first part of the final sentence of the exercise effectively removes from consideration a genuine bridge-style approach for routers. While there are good reasons for this, outlined in the final paragraph below, a better way to phrase this might be to ask why IP routers do not work like learning bridges, or, even more basically, why bridges do not use vector-distance routing.

Furthermore, a consequence of the second half of the final sentence is that there is no real difference in the cases (a) and (b) with bridge-style learning. Proper configuration would prevent address-assignment inconsistencies in each, which apparently had been the original concern.

So we are left with a model of "bridge-style learning" in which routers learn about each other through messages each sends periodically to other routers. This is not terribly bridge-like. Moreover, it is not clear what it means for routers to learn of each other by this method; if they are sending each other messages then either they already know about each other or else some form of broadcast is used. And broadcast runs into serious problems if there is a possibility of loops. If routers are sending out messages that are just broadcast on directly connected subnets, listing all the subnets they know about, and these messages include distance information, then they are more-or-less doing vector-distance routing. One routing approach

that might qualify under the terms of the exercise is if routers send out link-state-style periodic messages identifying their directly connected networks, and that these are propagated by flooding.

The main reason that IP routers cannot easily learn new subnet locations by examination of *data* packets is that they would then have to fall back on network-wide broadcast for delivery to unknown subnets. IP does indeed support a notion of broadcast, but broadcast in the presence of loop topology (which IP must support) fails rather badly unless specific (shortest-path) routes to each individual subnet are already known by the routers. And even if some alternative mechanism were provided to get routing started, path-length information would not be present in data packets, so future broadcasting would remain loop-unsafe. We note too that subnet routing requires that the routers learn the subnet masks, which are also not present in data packets. Finally, bridges may prefer passive learning simply because it avoids bridge-to-bridge compatibility issues.

28. If an IP packet addressed to a specific host A were inadvertently broadcast, and all hosts on the subnet did forwarding, then A would be inundated with multiple copies of the packet.

    Other reasons for hosts' not doing routing include the risk that misconfigured hosts could interfere with routing, or might not have up-to-date tables, or might not even participate in the same routing protocol that the real routers were using.

31. Traceroute sends packets with limited TTL values. If we send to an unassigned network, then as long as the packets follow default routes, traceroute will get normal answers. When the packet reaches a default-free (backbone) router, however (or more precisely a router which recognizes that the destination doesn't exist), the process will abruptly stop. Packets will not be forwarded further.

    The router that finally realizes the error will send back "ICMP host unreachable" or "ICMP net unreachable", but this ICMP result may not in fact be listened for by traceroute (*is* not, in implementations with which I am familiar), in which case the traceroute session will end with timeouts either way.

32. A can reach B and D but not C. To reach B, A sends ARP requests directly to B; these are passed by RB as are the actual ethernet packets. To reach D, A uses ARP to send to R2. However, if A tries to ARP to C, the request will not pass R1. Even if A knows C's Ethernet address already, the packets will not pass R1.

33. (a) Giving each department a single subnet, the nominal subnet sizes are 2^7, 2^6, 2^5, 2^5 respectively; we obtain these by rounding up to the nearest power of 2. A possible arrangement of subnet numbers is as follows. Subnet numbers are in binary and represent an initial segment of the bits of the last byte of the IP address; anything to the right of the / represents host bits. The / thus represents the subnet mask. Any individual bit can, by symmetry, be flipped throughout; there are thus several possible bit assignments.

A	0/	one subnet bit, with value 0; seven host bits
B	10/	
C	110/	
D	111/	

The essential requirement is that any two distinct subnet numbers remain distinct when the longer one is truncated to the length of the shorter.

(b) We have two choices: either assign multiple subnets to single departments, or abandon subnets and buy a bridge. Here is a solution giving A two subnets, of sizes 64 and 32; every other department gets a single subnet of size the next highest power of 2:

A	01/
	001/
B	10/
C	000/
D	11/

34. To solve this with routing, C has to be given its own subnet. Even if this is small, this reduces the available size of the original Ethernet to at most seven bits of subnet address. Here is a possible routing table for B; subnet numbers and masks are in binary. Note that many addresses match neither subnet.

net	subnet	mask	interface
200.0.0	0/000 0000	1000 0000	Ethernet
200.0.0	1000 00/00	1111 1100	direct link

Here C's subnet has been made as small as possible; only two host bits are available (a single host bit can't be used because all-zero-bits and all-ones-bits are reserved in the host portion of an address). C's address might now be 200.0.0.10000001, with the last octet again in binary.

35. (a) A would broadcast an ARP request "where is C?"
B would answer it; it would supply its own Ethernet address.
A would send C's packet to B's Ethernet address.
B would forward the packet to C.

(b) For the above to work, B must know to forward the packet *without* using subnet addressing; this is typically accomplished by having B's routing table contain a "host-specific route":

net/host	interface
C	direct link
200.0.0	Ethernet

Host-specific routes must be checked first for this to work.

36. Many arrangements are possible, although perhaps not likely. Here is an allocation scheme that mixes two levels of geography with providers; it works with 48-bit InterfaceIDs. The subdivisions become much more plausible with 64-bit InterfaceIDs.

Bytes 0-1:	3-bit prefix + country where site is located (5 bits is not enough to specify the country)
Bytes 2-3:	provider
Bytes 4-5:	Geographical region within provider

Bytes 6-8: Subscriber (large providers may have >64K subscribers)
Bytes 8-9: (Byte 8 is oversubscribed) Subnet
Bytes 10-15: InterfaceID

37. (a) DHCP will have considerable difficulty sorting out to which subnet various hosts belonged; subnet assignments would depend on which server answered first. The full DHCP deals with this by allowing servers to be manually configured to ignore address-assignment requests from certain physical addresses on the other subnet. Note that subnet assignment in this situation may matter for consistent naming, performance reasons, certain security access rules, and for broadcast protocols.

    (b) ARP will not be affected. Hosts will only broadcast ARP queries for other hosts on the same subnet; hosts on the other subnet will hear these but won't answer. A host on one subnet *would* answer an ARP query from the other subnet, if it were ever issued, but it wouldn't be.

38. (a): B (b): A (c): E (d): F (e): C (f): D
(For the last one, note that the first 14 bits of C4.6B and C4.68 match.)

39. (a) P's table:

address	nexthop
C2.0.0.0/8	Q
C3.0.0.0/8	R
C1.A3.0.0/16	PA
C1.B0.0.0/12	PB

Q's table:

address	nexthop
C1.0.0.0/8	P
C3.0.0.0/8	R
C2.0A.10.0/20	QA
C2.0B.0.0/16	QB

R's table:

address	nexthop
C1.0.0.0/8	P
C2.0.0.0/8	Q

(b) The same, except for the following changes of one entry each to P's and R's tables:
P: C3.0.0.0/8 Q // was R
R: C1.0.0.0/8 Q // was P

(c) Note the use of the longest-match rule to distinguish the entries for Q & QA in P's table, and for P & PA in Q's table.

P's table:

address	nexthop	
C2.0.0.0/8	Q	
C2.0A.10.0/20	P	// for QA
C1.A3.0.0/16	PA	
C1.B0.0.0/12	PB	

Q's table:

address	nexthop	
C1.0.0.0/8	P	
C1.A3.0.0/16	PA	// for PA
C2.0A.10.0/20	QA	
C2.0B.0.0/16	QB	

40. The longest-match rule is intended for this. Note that *all* providers now have to include entries for PA and QB, though.

P's table:

address	nexthop	
C2.0.0.0/8	Q	
C3.0.0.0/8	R	
C1.A3.0.0/16	Q	// entry for PA
C1.B0.0.0/12	PB	
C2.0B.0.0/16	R	// entry for QB

Q's table:

address	nexthop	
C1.0.0.0/8	P	
C3.0.0.0/8	R	
C1.A3.0.0/16	PA	// now Q's customer
C2.0A.10.0/20	QA	
C2.0B.0.0/16	R	// entry for QB

R's table:

address	nexthop	
C1.0.0.0/8	P	
C2.0.0.0/8	Q	
C1.A3.0.0/16	Q	// R also needs an entry for PA
C2.0B.0.0/16	QB	// QB is now R's customer

41. (a) Inbound traffic takes a single path to the organization's address block, which corresponds to the organization's "official" location. This means all traffic enters the organization at a single point even if much shorter alternative routes exist.

(b) For outbound traffic, the organization could enter into its own tables all the highest-level geographical blocks for the outside world, allowing the organization to route traffic to the exit geographically closest to the destination.

(c) For an approach such as the preceding to work for inbound traffic as well, the organization would have to be divided internally into geographically based subnets, and the outside world would then have to accept routing entries for each of these subnets. Consolidation of these subnets into a single external entry would be lost.

(d) We now need each internal router to have entries for internal routes to all the other internal IP networks; this suffices to ensure internal traffic never leaves.

42. Perhaps the primary problem with geographical addressing is what to do with geographically dispersed sites that have their own internal connections. Routing all traffic to a single point of entry seems inefficient.

    At the time when routing-table size became a critical issue, most providers were regional and thus provider-based addressing *was* more or less geographical.

43. (a) If Q does not advertise A to the world, then only traffic originating within Q will take the Q—A link. Other traffic will be routed first to P. If Q does advertise A, then traffic originating at an external site B will travel in via Q whenever the B–Q–A path is shorter than the B–P–A path.

    (b) Q must advertise A's reachability to the world, but it may put a very low "preference value" on this link.

    (c) The problem is that most outbound traffic will take the DEFAULT path, and nominally this is a single entry. Some mechanism for load-sharing must be put into place. Alternatively, A could enter into its internal routing tables some of its most common external IP destinations, and route to these via Q.

44. One approach might be to use a tree of all netmasks. We decide on the left versus right subtree at level i based on the ith bit of the address. A network with an n-bit mask is marked in the tree at level n. Given an address, we use the bits to proceed down the tree until we reach a dead end. At that point we use the last-encountered network; this ensures the longest match was found. Such a tree is sometimes called a trie.

    This strategy is linear in the address size. Performance might be enhanced by handling 4 or even 8 address bits at each level of the tree, although this would lead to some increased space used.

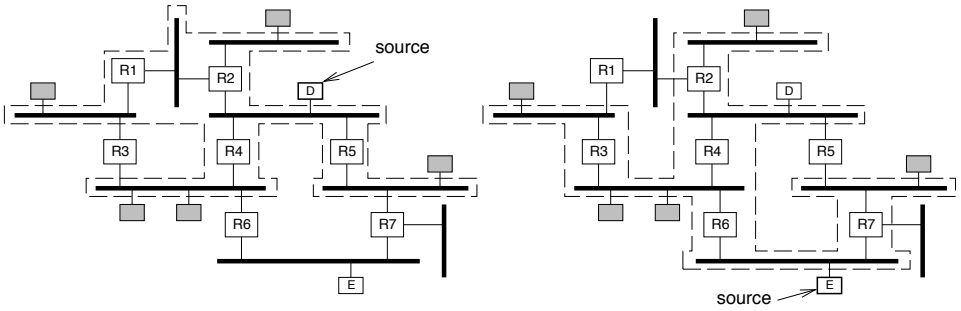
    Another approach might be to maintain a separate dictionary for each n, $1 \leq n \leq 24$, of all masks of length n. We start with the longest mask length and work backwards, at stage n searching for the first n bits of the address in the length-n dictionary. If dictionary lookup were sufficiently fast this might also be roughly linear in address length.

45. (a) R1 should be configured to forward traffic destined for outside of A to the new ISP. R2 should route traffic for A to A as before. Note that if a host on N sends its outbound traffic to R2 by mistake, R2 will send it via the old link. R2 should continue to advertise N to the rest of A. N's outbound traffic would then take the new link, but inbound traffic would still travel via A. Subnets are not announced into the backbone routing tables, so R1 would not announce N to the world.

    (b) If N has its own IP network number, then R1 does announce its route to N to the world. R1 would not necessarily announce its route to A, however. R2 would not change: it would still announce N into A. Assuming A does not announce its route to N into its provider, all external traffic to and from N now goes through the new link, and N-A traffic goes through R2.

    (c) If A wants to use N's R1 link as a backup, then R1 needs to announce to the backbone that it has a route to A, but give this route a cost higher than that of A's original link (techniques for doing this via BGP include route preference indications and "padding" the report with extra ASs.)

46.



47. (a) One multicast transmission involves all $k + k^2 + ... + k^{N-1} = (k^N - k)/(k - 1)$ links.

    (b) One unicast retransmission involves N links; sending to everyone would require $N \times k^N$ links.

    (c) Unicast transmission to x fraction of the recipients uses $x \times N \times k^N$ links. Equating this to the answer in (a), we get

$$x = (k^N - k)/((k - 1) \times N \times k^N) \approx 1/(k - 1) \times N$$

48. The simplest way to determine if one is connected to the MBone (and has no one to ask) is to install an "SDP session browser" such as sdr or multikit (www.mbone.com) and see if announcements show up.

Solutions for Chapter 5

1. (a) An application such as TFTP, when sending initial connection requests, might want to know the server isn't accepting connections.

   (b) On typical Unix systems, one needs to open a socket with attribute IP_RAW (traditionally requiring special privileges) and receive all ICMP traffic.

   (c) A receiving application would have no way to identify ICMP messages as such, or to distinguish between these messages and protocol-specific data.

2. (a) In the following, the client receives file "foo" when it thinks it has requested "bar".

   1. The client sends a request for file "foo", and immediately aborts locally. The request, however, arrives at the server.
   2. The client sends a new request, for file "bar". It is lost.
   3. The server responds with first data packet of "foo", answering the only request it has actually seen.

   (b) Requiring the client to use a new port number for each separate request would solve the problem. To do this, however, the client would have to trust the underlying operating system to assign a new port number each time a new socket was opened. Having the client attach a timestamp or random number to the file request, to be echoed back in each data packet from the server, would be another approach fully under the application's control.

3. The TFTP protocol is a reasonable model although with some idiosyncrasies that address other issues; see RFC 1350. TFTP's first packet, called Read Request, RRQ, simply names a file. Upon receipt, the server creates a new ephemeral port from which to answer, and begins sending data from that new port. The client assumes that the first well-formed packet it receives is this server data, and records the data's source port. Any subsequent packets from a different port are discarded and an error response is sent.

   The basic stop-and-wait transfer is standard, although one must decide if sequence numbers are allowed to wrap around or not. Here are approaches, TFTP's and otherwise, for (a)-(c):

   (a) The most basic approach here is to require the server to keep track of connections, as long as they are active. The problem with this is that the client is likely to be simply an application, and can exit at any time. It may exit and retransmit a request for a different file, or a new request for the same file, before the server knows there was a problem or status change.

   A more robust mechanism for this situation might be a CONNECT_NUM field, either chosen randomly or clock-driven or incremented via some central file for each client connection attempt. Such a field would correspond roughly with TCP's ISN.

   In TFTP, if the RRQ is duplicated then the server might well create two processes and two ports from which to answer. (A server that attempted to do otherwise would have to maintain considerable state about past RRQ's.) Whichever process contacted the client first would win out, though, while the other would receive an error response from the client. In one sense, then, duplicate TFTP RRQ's do duplicate the connection, but only one of the duplicates survives.

(b) The TFTP approach here is to have the client enter a "dallying" period after the final data was received, so that the process is still around (perhaps moved to the background) to receive and reacknowledge any retransmissions of the final data. This period roughly corresponds to TIMEWAIT.

(c) The dallying approach of (b) also ties up the client socket for that period, preventing another incarnation of the connection. (However, TFTP has no requirement that dallying persist for a time interval approaching the MSL.)

TFTP also specifies that *both* sides are to choose "random" port numbers for each connection (although "random" is generally interpreted as "assigned by the operating system"). If either side chooses a new port number, then late-arriving packets don't interfere even if the other side reuses its previous port number. A CONNECT_NUM field would also be effective here.

4. Host A has sent a FIN segment to host B, and has moved from ESTABLISHED to FIN_WAIT_1. Host A then receives a segment from B that contains both the ACK of this FIN, and also B's own FIN segment. This could happen if the application on host B closed its end of the connection immediately when the host A's FIN segment arrived, and was thus able to send its own FIN along with the ACK.

   Normally, because the host B application must be scheduled to run before it can close the connection and thus have the FIN sent, the ACK is sent before the FIN. While "delayed ACKs" are a standard part of TCP, traditionally only ACKs of DATA, not FIN, are delayed. See RFC 813 for further details.

5. The two-segment-lifetime timeout results from the need to purge old late duplicates, and uncertainty of the sender of the last ACK as to whether it was received. For the first issue we only need one connection endpoint in TIMEWAIT; for the second issue, a host in the LAST_ACK state expects to receive the last ACK, rather than send it.

6. The receiver includes the advertised window in the ACKs to the sender. The sender probes the receiver to know when the advertised window becomes greater than 0; if the receiver's ACK advertising a larger window is lost, then a later sender probe will elicit a duplicate of that ACK.

   If responsibility for the lost window-size-change ACK is shifted from the sender to the receiver, then the receiver would need a timer for managing retransmission of this ACK until the receiver were able to verify it had been received.

   A more serious problem is that the receiver only gets confirmation that the sender has received the ACK when new data arrives, so if the connection happens to fall idle the receiver may be wasting its time.

8. The sequence number doesn't always begin at 0 for a transfer, but is randomly or clock generated.

9. (a) The advertised window should be large enough to keep the pipe full; delay (RTT) $\times$ bandwidth here is 100 ms $\times$ 100 Mbps = 10 Mb = 1.25 MB of data. This requires 21 bits ($2^{21} = 2,097,152$) for the AdvertisedWindow field. The sequence number field

must not wrap around in the maximum segment lifetime. In 60 seconds, 750 MB can be transmitted. 30 bits allows a sequence space of 1024 MB, and so will not wrap in 60 seconds. (If the maximum segment lifetime were not an issue, the sequence number field would still need to be large enough to support twice the maximum window size; see "Finite Sequence Numbers and Sliding Window" in Section 2.5.)

(b) The bandwidth is straightforward from the hardware; the RTT is also a precise measurement but will be affected by any future change in the size of the network. The MSL is perhaps the least certain value, depending as it does on such things as the size and complexity of the network, and on how long it takes routing loops to be resolved.

10. The problem is that there is no way to determine whether a packet arrived on the first attempt or whether it was lost and retransmitted.

Having the receiver echo back immediately and measuring the elapsed times would help; many Berkeley-derived implementations measure timeouts with a 0.5 sec granularity and round-trip times for a single link without loss would generally be one to two orders of magnitude smaller. But verifying that one had such an implementation is itself rather difficult.

11. (a) This is 125MB/sec; the sequence numbers wrap around when we send 2^{32} B = 4 GB. This would take 4GB/(125MB/sec) = 32 seconds.

(b) Incrementing every 32 ms, it would take about $32 \times 4 \times 10^9$ ms, or about four years, for the timestamp field to wrap.

12. (a) If a SYN packet is simply a duplicate, its ISN value will be the same as the initial ISN. If the SYN is not a duplicate, and ISN values are clock-generated, then the second SYN's ISN will be different.

(b) We will assume the receiver is single-homed; that is, has a unique IP address. Let $\langle raddr, rport \rangle$ be the remote sender, and $lport$ be the local port. We suppose the existence of a table T indexed by $\langle lport, raddr, rport \rangle$ and containing (among other things) data fields lISN and rISN for the local and remote ISNs.

```
if (connections to lport are not being accepted)
    send RST
else if (there is no entry in T for ⟨lport, raddr, rport⟩) // new SYN
    Put ⟨lport, raddr, rport⟩ into a table,
    Set rISN to be the received packet's ISN,
    Set lISN to be our own ISN,
    Send the reply ACK
    Record the connection as being in state SYN_RECD
else if (T[⟨lport, raddr, rport⟩] already exists)
    if (ISN in incoming packet matches rISN from the table)
        // SYN is a duplicate; ignore it
    else
        send RST to ⟨raddr, rport⟩)
```

13. $x =< y$ if and only if $(y - x) \geq 0$, where the expression $y - x$ is taken to be signed even though x and y are not.

14. (a) A would send an ACK to B for the new data. When this arrived at B, however, it would lie outside the range of "acceptable ACKs" and so B would respond with its own current ACK. B's ACK would be acceptable to A, and so the exchanges would stop.

    If B later sent less than 100 bytes of data, then this exchange would be repeated.

    (b) Each end would send an ACK for the new, forged data. However, when received both these ACKs would lie outside the range of "acceptable ACKs" at the other end, and so each of A and B would in turn generate their current ACK in response. These would again be the ACKs for the forged data, and these ACKs would again be out of range, and again the receivers would generate the current ACKs in response. These exchanges would continue indefinitely, until one of the ACKs was lost.

    If A later sent 200 bytes of data to B, B would discard the first 100 bytes as duplicate, and deliver to the application the second 100 bytes. It would acknowledge the entire 200 bytes. This would be a valid ACK for A.

    For more examples of this type of scenario, see Joncheray, L; A Simple Active Attack Against TCP; *Proceedings of the Fifth USENIX UNIX Security Symposium*, June, 1995.

15. Let H be the host to which A had been connected; we assumed B is able to guess H. As we are also assuming telnet connections, B can restrict probes to H's telnet port (port 23).

    First, B needs to find a port A had been using. For various likely ephemeral port numbers N, B sends an ACK packet from port N to ⟨H,telnet⟩. For many implementations, ephemeral ports start at some fixed value (eg N=1024) and increase sequentially; for an unshared machine it is unlikely that very many ports had been used. If A had had no connection from port N, H will reply to B with a RST packet. But if H *had* had an outstanding connection to ⟨A,N⟩, then H will reply with either nothing (if B's forged ACK happened to be Acceptable, *ie* in the current window at the point when A was cut off), or the most recent Acceptable ACK (otherwise). Zero-byte data packets can with most implementations also be used as probes here.

    Once B finds a successful port number, B then needs to find the sequence number H is expecting; once B has this it can begin sending data on the connection as if it were still A. To find the sequence number, B again takes advantage of the TCP requirement that H reply with the current ACK if B sends an ACK or DATA inconsistent with H's current receive window [that is, an "unacceptable ACK"]. In the worst case B's first probe lies in H's window, in which case B needs to send a second probe.

16. We keep a table T, indexed by ⟨address,port⟩ pairs, and containing an integer field for the ISN and a string field for the connection's DATA.

    We will use =< for sequence number comparison as in Exercise 13.

    ```
    if (SYN flag is set in P.TCPHEAD.Flags)
        Create the entry T[⟨P.IPHEAD.SourceAddr,P.TCPHEAD.SrcPort⟩]
        T[...].ISN = P.TCPHEAD.SequenceNum
        T[...].DATA = ⟨empty string⟩
    else
        See if DATA bit in P.TCPHEAD.Flags is set; if not, ignore
        Look up T[⟨P.IPHEAD.SourceAddr,P.TCPHEAD.SrcPort⟩]
          (if not found, ignore the packet)
    ```

See if P.TCPHEAD.SequenceNum =< T[...].ISN+100.
If so, append the appropriate portion of the packet's data to T[...].DATA

17. (a) 1. C connects to A, and gets A's current clock-based ISN_{A1}.

2. C sends a SYN packet to A, purportedly from B. A sends SYN+ACK, with ISN_{A2} to B, which we are assuming is ignored.

3. C makes a guess at ISN_{A2}, eg ISN_{A1} plus some suitable increment, and sends the appropriate ACK to A, along with some data that has some possibly malign effect on A. As in Step 2, this packet too has a forged source address of B.

4. C does nothing further, and the connection either remains half-open indefinitely or else is reset, but the damage is done.

(b) In one 40 ms period there are 40 ms/4μsec = 10,000 possible ISN_As; we would expect to need about 10,000 tries.

Further details can be found in Morris, RT; A Weakness in the 4.2BSD UNIX TCP/IP Software; *Computing Science Technical Report No. 117,* AT&T Bell Laboratories, Murray Hill, NJ, 1985.

18. (a)
T=0.0	'a' sent
T=1.0	'b' collected in buffer
T=2.0	'c' collected in buffer
T=3.0	'd' collected in buffer
T=4.0	'e' collected in buffer
T=4.1	ACK of 'a' arrives, "bcde" sent
T=5.0	'f' collected in buffer
T=6.0	'g' collected in buffer
T=7.0	'h' collected in buffer
T=8.0	'i' collected in buffer
T=8.1	ACK arrives; "fghi" sent

(b) The user would type ahead blindly at times. Characters would be echoed between 4 and 8 seconds late, and echoing would come in chunks of four or so. Such behavior is quite common over telnet connections, even those with much more modest RTTs, but the extent to which this is due to the Nagle algorithm is unclear.

(c) With the Nagle algorithm, the mouse would appear to skip from one spot to another. Without the Nagle algorithm the mouse cursor would move smoothly, but it would display some inertia: it would keep moving for one RTT after the physical mouse were stopped.

19. To trigger the Silly Window Syndrome we require that the receiver is doing reads in small chunks, after the sender has sent everything in its window and thus shrunk the window size to zero. After each read, the receiver frees a small chunk of buffer space and sends back its ACK with a window advertisement for this space. The sender then sends a chunk of data to fill this space, shrinking the window back to zero.

The receiver could unilaterally prevent this by not advertising less than a full-sized segment's worth of space. The sender could by not sending less than a full segment (unless PUSHed), and instead waiting for further window size increases.

Further details can be found in RFC 813.

20. (a) We have 4096 ports; we eventually run out if the connection rate averages more than 4096/60 = 70 per sec. (The range used here for ephemeral ports, while small, is typical of older TCP implementations.)

    (b) In the following we let A be the host that initiated the close (and that is in TIMEWAIT); the other host is B. A is nominally the client; B the server.

    If B fails to receive an ACK of its final FIN, it will eventually retransmit that FIN. So long as A remains in TIMEWAIT it is supposed to reply again with the corresponding ACK. If the sequence number of the FIN were incorrect, A would send RST.

    If we allow reopening before TIMEWAIT expires, then a given very-late-arriving FIN might have been part of any one of a number of previous connections. For strict compliance, host A would have to maintain a list of prior connections, and if an old FIN arrived (as is theoretically possible, given that we are still within the TIMEWAIT period for the old connection), host A would consult this list to determine whether the FIN had an appropriate sequence number and hence whether an ACK or RST should be sent.

    Simply responding with an ACK to all FINs with sequence numbers before the ISN of the current connection would seem reasonable, though. The old connection, after all, no longer exists at B's end to be reset, and A knows this. A knows, in fact, that a prior final ACK or RST that it sent in response to B's FIN *was* received by B, since B allowed the connection to be reopened, and so it might justifiably not send anything.

21. Whichever endpoint remains in TIMEWAIT must retain a record of the connection for the duration of TIMEWAIT; as the server typically is involved in many more connections than clients, the server's recordkeeping requirements would be much more onerous.

    Note also that some implementations of TIMEWAIT simply disallow *all* new connections to the port in question for the duration, not only those from the particular remote connection that initiated the TIMEWAIT. Since a server cannot choose a new port, this might mean it could process at most one connection per TIMEWAIT interval.

    In situations where the client requests some variable-length stream (*eg* a file), the server might plausibly initiate the active close to indicate the end of the data.

22. Timeouts indicates that the network is congested and that one should send fewer packets rather than more. Exponential backoff immediately gives the network twice as long to deliver packets (though a single linear backoff would give the same); it also rapidly adjusts to even longer delays, thus it in theory readily accommodating sharp increases in RTT without further loading the already overtaxed routers. If the RTT suddenly jumps to 15 times the old TimeOut, exponential increase retransmits at T=1, 3, 7, and 15; linear increase would retransmit at T=1, 3, 6, 10, and 15. The difference here is not large. Exponential backoff makes the most difference when the RTT has increased by a very large amount, either due to congestion or network reconfiguration, or when "polling" the network to find the initial RTT.

23. The probability that a Normally distributed random variable is more than π standard deviations above the mean is about 0.0816%.

24. If every other packet is lost, we transmit each packet twice.

(a) Let $E \geq 1$ be the value for EstimatedRTT, and $T = 2 \times E$ be the value for TimeOut. We lose the first packet and back off TimeOut to $2 \times T$. Then, when the packet arrives, we resume with EstimatedRTT = E, TimeOut = T. In other words, TimeOut doesn't change.

(b) Let T be the value for TimeOut. When when we transmit the packet the first time, it will be lost and we will wait time T. At this point we back off and retransmit using TimeOut = $2 \times T$. The retransmission succeeds with an RTT of 1 sec, but we use the backed-off value of $2 \times T$ for the next TimeOut. In other words, TimeOut doubles with each received packet. This is Not Good.

25. Here is a spreadsheet run with initial Deviation =1.0; it took 20 iterations for TimeOut to fall below 4.0. With an initial Deviation of 0.1, it took 19 iterations; with an initial Deviation of 2 it took 21.

row #	SampleRTT	EstRTT	Dev	diff	TimeOut
		4.00	1.00		
1	1.00	3.63	1.25	-3.00	8.63
2	1.00	3.30	1.42	-2.63	8.98
3	1.00	3.01	1.53	-2.30	9.13
4	1.00	2.76	1.59	-2.01	9.12
5	1.00	2.54	1.61	-1.76	8.99
6	1.00	2.35	1.60	-1.54	8.76
7	1.00	2.18	1.57	-1.35	8.46
8	1.00	2.03	1.52	-1.18	8.12
9	1.00	1.90	1.46	-1.03	7.74
10	1.00	1.79	1.39	-0.90	7.35
11	1.00	1.69	1.32	-0.79	6.95
12	1.00	1.60	1.24	-0.69	6.55
13	1.00	1.53	1.16	-0.60	6.16
14	1.00	1.46	1.08	-0.53	5.78
15	1.00	1.40	1.00	-0.46	5.41
16	1.00	1.35	0.93	-0.40	5.06
17	1.00	1.31	0.86	-0.35	4.73
18	1.00	1.27	0.79	-0.31	4.42
19	1.00	1.24	0.72	-0.27	4.13
20	1.00	1.21	0.66	-0.24	3.86

26. One approach to this, shown below, is to continue the table above, except that whenever TimeOut would fall below 4.0 we replace the SampleRTT of that row with 4.0.

We could also create a table starting from scratch, using an initial EstimatedRTT of 1.0 and seeding the first few rows with a couple instances of SampleRTT = 4.0 to get TimeOut $\geq$ 4.0 in the first place.

Either way, N is between 6 and 7 here.

row #	SampleRTT	EstRTT	Dev	diff	TimeOut
19	1.00	1.24	0.72	-0.27	4.13
20	**4.00**	1.58	0.98	2.76	5.50
21	1.00	1.51	0.93	-0.58	5.22
22	1.00	1.45	0.88	-0.51	4.95
23	1.00	1.39	0.82	-0.45	4.68
24	1.00	1.34	0.77	-0.39	4.42
25	1.00	1.30	0.72	-0.34	4.16
26	**4.00**	1.64	0.96	2.70	5.49
27	1.00	1.56	0.92	-0.64	5.25
28	1.00	1.49	0.88	-0.56	4.99
29	1.00	1.43	0.83	-0.49	4.74
30	1.00	1.37	0.78	-0.43	4.48
31	1.00	1.33	0.73	-0.37	4.24
32	**4.00**	1.66	0.97	2.67	5.54

27. Here is the table of the updates to the EstRTT, etc statistics. Packet loss is ignored; the SampleRTTs given may be assumed to be from successive singly transmitted segments. Note that the first column, therefore, is simply a row number, *not* a packet number, as packets are sent without updating the statistics when the measurements are ambiguous. Note also that both algorithms calculate the same values for EstimatedRTT; only the TimeOut calculations vary.

					new TimeOut	old TimeOut
	SampleRTT	EstRTT	Dev	diff	EstRTT+4×Dev	2×EstRTT
		1.00	0.10		1.40	2.00
1	5.00	1.50	0.59	4.00	3.85	3.00
2	5.00	1.94	0.95	3.50	5.74	3.88
3	5.00	2.32	1.22	3.06	7.18	4.64
4	5.00	2.66	1.40	2.68	8.25	5.32

New algorithm (TimeOut = EstimatedRTT+ 4×Deviation):

There are a total of three retransmissions, two for packet 1 and one for packet 3.

The first packet after the change times out at T=1.40, the value of TimeOut at that moment. It is retransmitted, with TimeOut backed off to 2.8. It times out again 4.2 sec after the first transmission, and TimeOut is backed off to 5.6.

At T=5.0 the first ACK arrives and the second packet is sent, using the backed-off TimeOut value of 5.6. This second packet does not time out, so this constitutes an unambiguous RTT measurement, and so timing statistics are updated to those of row 1 above.

When the third packet is sent, with TimeOut=3.85, it times out and is retransmitted. When its ACK arrives the fourth packet is sent, with the backed-off TimeOut value, 2×3.85 = 7.70; the resulting RTT measurement is unambiguous so timing statistics are updated to row 2. When the fifth packet is sent, TimeOut=5.74 and no further timeouts occur.

If we continue the above table to row 9, we get the maximum value for TimeOut, of 10.1, at which point TimeOut decreases toward 5.0.

Original algorithm (TimeOut = 2×EstimatedRTT):

There are five retransmissions: for packets 1, 2, 4, 6, 8.

The first packet times out at T=2.0, and is retransmitted. The ACK arrives before the second timeout, which would have been at T=6.0.

When the second packet is sent, the backed-off TimeOut of 4.0 is used and we time out again. TimeOut is now backed off to 8.0. When the third packet is sent, it thus does not time out; statistics are updated to those of row 1.

The fourth packet is sent with TimeOut=3.0. We time out once, and then transmit the fifth packet without timeout. Statistics are then updated to row 2.

This pattern continues. The sixth packet is sent with TimeOut = 3.88; we again time out once, send the seventh packet without loss, and update to row 3. The eighth packet is sent with TimeOut=4.64; we time out, back off, send packet 9, and update to row 4. Finally the tenth packet does not time out, as TimeOut=2×2.66=5.32 is larger than 5.0.

TimeOut continues to increase monotonically towards 10.0, as EstimatedRTT converges on 5.0.

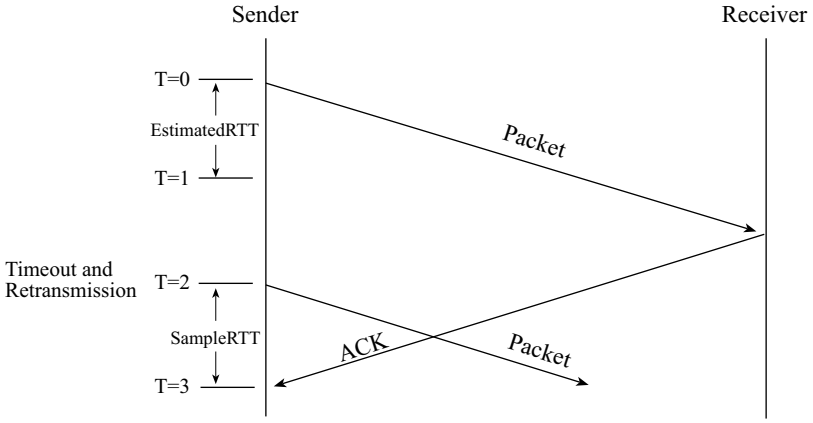
28. Let the real RTT (for successful transmissions) be 1.0 units. By hypothesis, every packet times out once and then the retransmission is acknowledged after 1.0 units; this means that each SampleRTT measurement is TimeOut+1 = EstimatedRTT+1. We then have

$$\begin{aligned}\text{EstimatedRTT} &= \alpha\times \text{EstimatedRTT} + \beta\times \text{SampleRTT} \\ &= \text{EstimatedRTT} + \beta\times(\text{SampleRTT} - \text{EstimatedRTT}). \\ &\geq \text{EstimatedRTT} + \beta\end{aligned}$$

Thus it follows that the Nth EstimatedRTT is greater than or equal to $N\beta$.

Without the assumption TimeOut = EstimatedRTT we still have SampleRTT − EstimatedRTT ≥ 1 and so the above argument still applies.

29. For the steady state, assume the true RTT is 3 and EstimatedRTT is 1. At T=0 we send a data packet. Since TimeOut is twice EstimatedRTT=1, at T=2 the packet is retransmitted. At T=3 the ACK of the original packet returns (because the true RTT is 3); measured SampleRTT is thus $3 - 2 = 1$; this equals EstimatedRTT and so there is no change. This is illustrated by the following diagram:



To get to such a steady state, assume that originally RTT = EstimatedRTT = 1.45, say, and RTT then jumps to 3.0 as above. The first packet sent under the new rules will time out and be retransmitted at T=2.9; when the ACK arrives at T=3.0 we record SampleRTT = 0.1. This causes EstimatedRTT to decrease. It will continue to grow smaller, monotonically (at least if β is not too large), converging on the value 1.0 as above.

30. A FIN or RST must lie in the current receive window. A RST outside this window is ignored; TCP responds to an out-of-window FIN with the current ACK:

> If an incoming segment is not acceptable, an acknowledgment should be sent in reply (unless the RST bit is set, if so drop the segment and return) [RFC793]

Note that a RST can lie anywhere within the current window; its sequence number need not be the next one in sequence.

If a FIN lies in the current window, then TCP waits for any remaining data and closes the connection. If a RST lies in the current window then the connection is immediately closed:

> If the RST bit is set then, any outstanding RECEIVEs and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

31. (a) The first incarnation of the connection must have closed successfully and the second must have opened; this implies the exchange of FIN and SYN packets and associated ACKs. The delayed data must also have been successfully retransmitted.

(b) One plausible hypothesis is that two routes were available from one host to the other. Traffic flowed on one link, which suddenly developed severe congestion or routing-loop delays at which point subsequent traffic was switched to the other, now-faster, link. It doesn't matter whether the two routes were both used initially on an alternating basis, or if the second route was used only after the first route failed.

32. We suppose A is connected to B and wishes to hand off the connection to C. There is more than one approach; in the following we assume that A and C do most of the work, and that A decides on a point in the sequence number stream after which B is to send to C. It could be argued that B is better positioned to make that determination. C also continues here with the sequence numbers started by A.

New function call event: handoff(); allowed in ESTABLISHED state only.
New packet types:

```
HANDOFF_REQ     // request from A to C
HANDOFF_DO      // request from A to B
```

New states:

```
HANDOFF_CALLED  // for A
H_REQ_SENT      // for A, HANDOFF_REQ sent to C
```

```
H_REQ_ACK       // for A; C has acknowledged HANDOFF_REQ
H_REQ_RECD      // for C
H_START_WAIT    // for C
H_TIMEWAIT      // for A
```

Here is a chronology of events.

1. handoff() called. A moves to state HANDOFF_CALLED, and identifies a sequence number H_SEQ (for B) after which data is to be sent to C. A waits for B to send up to this sequence number, blocking further transmissions by shrinking the upper edge of the receive window to H_SEQ. Whether or not A buffers data following H_SEQ, and forwards it to C, is optional.

2. A sends HANDOFF_REQ to C, with sequence number H_SEQ−1 (from B) and A's own current sequence number. C moves to state H_REQ_RECD. A moves to state H_REQ_SENT. If A has been buffering data past H_SEQ, it might send it to C at this point.

3. C sends an ACK to A to accept the handoff (or RST to reject it). If the former, A moves to state H_REQ_ACK. C moves to H_START_WAIT, and waits to hear from B.

4. A sends HANDOFF_DO to B, with H_SEQ. B remains ESTABLISHED, and sends an ACK to A, which moves to H_TIMEWAIT. B also sends an ACK to C, which moves to ESTABLISHED.

Any data with sequence number before H_SEQ that arrives at A during the H_TIMEWAIT period is now forwarded by A to C.

33. (a) In order to disallow simultaneous open, an endpoint in state SYN_SENT should not accept SYN packets from the other end. This means that the edge in the state diagram from SYN_SENT to SYN_RECD should be removed. Instead, the response to SYN in SYN_SENT would be something like RST.

(b) As long as either side is allowed to close, no. The timings of the close() calls are an application-level issue. If both sides happen to request at approximately the same instant that a connection be closed, it hardly seems appropriate to hold the connection open while the requests are serialized.

Looked at another way, disallowing simultaneous opens in effect simply requires that both sides adhere to the established roles of "client" and "server". At the point of closing, however, there simply is no established role for whether client or server is to close the connection. It would indeed be possible to require that the client, for example, had to initiate the close, but that would leave the server somewhat at the mercy of the client.

(c) The minimum additional header information for this new interpretation is a bit indicating whether the sender of the packet was the client for that connection or the server. This would allow the receiving host to figure out to which connection the arriving packet belongs.

With this bit in place, we can label the nodes and edges at or above the ESTABLISHED state with "client" or "server" roles. The edge from LISTEN to SYN_SENT is the exception, traversed only if a server (LISTEN state) takes on a client role (SYN_SENT). We replace this edge with a notion of creating the second connection; the original endpoint remains in the server-role LISTEN state and a new endpoint (with same port number), on in effect a new diagram, is created in the client-role SYN_SENT state.

The edge from SYN_SENT to SYN_RCVD would be eliminated; during a simultaneous open the arriving SYN would be delivered to the server-role endpoint, still in the LISTEN state, rather than to the client-role endpoint in state SYN_SENT.

34. (a) One would now need some sort of connection number assigned by the client side to play the role of the port number in demultiplexing traffic; with this in place, headers might not change much at all. Client and server sockets will now be fundamentally different objects. Server sockets would be required to bind() themselves to a port number (perhaps at creation time); clients would be forbidden to do this.

    (b) We still need to make sure that a client connection number is not reused within the 2×MSL period, at least not with the same server port. However, this is now a TCP-layer responsibility, not an application concern. Assuming the client connection number were assigned at the time of connection, clients would not need to be aware of TIMEWAIT at all: they would be freed of the requirement they close one socket and reopen a new one to get a fresh port number.

    Since client connection numbers are now not visible to the client, simply placing a connection number out of service entirely during the TIMEWAIT interval, for connections to any server, would be a tolerable approach.

    (c) The rlogin/rsh protocol authenticates clients by seeing that they are using a "reserved" port on the sending host (normally, a port only available to system-level processes). This would no longer be possible.

    However, the following variation would still be possible: when an rsh server host S receives a client request from host C, with connection number N, then S could authenticate the request with C by initiating a second connection to a reserved port on C, whereupon some sort of authentication application on C would verify that connection number N was indeed being used by an authorized rsh client on C. Note that this scheme implies that connection numbers are at least visible to the applications involved.

35. (a) A program that connect()s, and then sends whatever is necessary to get the server to close its end of the connection (eg the string "QUIT"), and then sits there, idle but not disconnecting, will suffice. Note that the server has to be willing to initiate the active close based on some client action.

    (b) Alas, most telnet clients do *not* work here. Although many can connect to an arbitrary port, and issue a command such as QUIT to make the server initiate the close, they generally do close immediately in response to receiving the server's FIN.

    However, the sock program, written by W. Richard Stevens, can be used instead. In the (default) client mode, it behaves like a command-line telnet. The option -Q 100 makes sock wait 100 seconds after receiving the server FIN before it closes its end of the connection. Thus the command

        sock -Q 100 *hostname* 25

    can be used to demonstrate FIN_WAIT_2 with an SMTP (email) server (port 25) on *hostname*, using the QUIT command.

    sock is available from ftp.uu.net in published/books/stevens.tcpipiv1.tar.Z.

36. (a) Let A be the closing host and B the other endpoint. A sends message1, pauses, sends message2, and then closes its end of the connection for reading. B gets message1 and

sends a reply, which arrives after A has performed the half-close. B doesn't read message2 immediately; it remains in the TCP layer's buffers. B's reply arrives at A after the latter has half-closed, and so A responds with RST as per the quoted passage from RFC 1122. This RST then arrives at B, which aborts the connection and the remaining buffer contents (ie message2) are lost.

Note that if A had performed a full-duplex close, the same scenario can occur. However, it now depends on B's reply crossing A's FIN in the network. The half-close-for-reading referred to in this exercise is actually purely a local state change; a connection that performs a half-close closing its end for *writing* may however send a FIN segment to indicate this state to the other endpoint.

(b) This is perhaps misleading; essentially everyone's close initiates a full-duplex close. On typical Unix systems, a half-duplex close is achieved through use of the shutdown call. The socket option SO_LINGER pertains to closing also, determining whether a process with queued but undelivered TCP data will block on close until the data is delivered.

37. Incrementing the Ack number for a FIN is essential, so that the sender of the FIN can determine that the FIN was received and not just the preceding data.

For a SYN, any ACK of subsequent data would increment the acknowledgement number, and any such ACK would implicitly acknowledge the SYN as well (data cannot be ACKed until the connection is established). Thus, the incrementing of the sequence number here is a matter of convention and consistency rather than design necessity.

38. (b) One method would be to invent an option to specify that the first n bytes of the TCP data should be interpreted as options.

(c) A tcp endpoint receiving an unknown option might

- *close/abort the connection*. This makes sense if the connection cannot meaningfully continue when the option isn't understood.
- *ignore the option but keep the TCP data*. This is the current RFC 1122 requirement.
- *send back "I don't understand"*. This is simply an explicit form of the previous response. A refinement might be to send back some kind of list of options the host *does* understand.
- *discard the accompanying the TCP data*. One possible use might be if the data segment were encrypted, or in a format specified by the option. Some understanding would be necessary regarding sequence numbers for this to make sense; if the entire TCP data segment was an extended option block then the sequence numbers shouldn't increase at all.
- *discard the first n bytes of the TCP data*. This is an extension of the previous strategy to handle the case where the first n bytes of the TCP data was to be interpreted as an expanded options block; it is not clear though when the receiver might understand n but not the option itself.

39. TCP faces two separate crash-and-reboot scenarios: a crash can occur in the middle of a connection, or between two consecutive incarnations of a connection.

The first leads to a "half-open" connection where one endpoint has lost all state regarding the connection; if either the stateless side sends a new SYN or the stateful side sends new data, the other side will respond with RST and the half-open connection will be dissolved bilaterally. CHAN, by comparison, does not require any connection setup before sending data.

If one host crashes and reboots between two consecutive connection incarnations, the only way the first incarnation could affect the second is if a late-arriving segment from the first happens to fit into the receive window of the second. TCP establishes a quasi-random initial sequence number during its three-way handshake at connection open time. A 64KB window, the maximum allowed by the original TCP, spans less than 0.0015% of the sequence number space. Therefore, there is very little chance that data from a previous incarnation of the connection will happen to fall in the current window; any data outside the window is discarded. (TCP also is supposed to implement "quiet time on startup", an initial $1\times$MSL delay for all connections after bootup.)

CHAN doesn't have a three-way handshake, and so can't decide on a random sequence number.

40. (a) Nonexclusive open, reading block N, writing block N, and seeking to block N all are idempotent, *ie* have the same effect whether executed once or twice.

    (b) create() is idempotent if it means "create if nonexistent, or open if it exists already". mkdir() is idempotent if the semantics are "create the given directory if it does not exist; otherwise do nothing". delete() (for either file or directory) works this way if its meaning is "delete if the object is there; otherwise, ignore."

    Operations fundamentally incompatible with at-least-once semantics include exclusive open (and any other form of file locking), and exclusive create.

    (c) The directory-removing program would first check if the directory exists. If it does not, it would report its absence. If it does exist, it invokes the system call rmdir().

41. (a) The problem is that reads aren't serviced in FIFO order; disk controllers typically use the "elevator" or SCAN algorithm to schedule writes, in which the pool of currently outstanding writes is sorted by disk track number and the writes are then executed in order of increasing track number. Using a single CHAN channel would force writes to be executed serially even when such a sequence required lots of otherwise-unnecessary disk head motion.

    If a pool of N CHAN-like channels were used, the disk controller would at any time have about N writes to schedule in the order it saw fit.

    (b) Suppose a client process writes some data to the server, and then the client system shuts down "gracefully", flushing its buffers (or avails itself of some other mechanism to flush the buffer cache). At this point data on a local disk would be safe; however, a *server* crash would now cause the loss of client data remaining in the server's buffers. The client might *never* be able to verify that the data was safely written out.

    (c) One approach would be to modify CHAN to support multiple independent outstanding requests on a single logical channel, and to support replies in an arbitrary order, not necessarily that in which the corresponding requests were received. Such a mechanism

would allow the server to respond to multiple I/O requests in whatever order was most convenient.

A subsequent request could now no longer serve as an ACK of a previous reply; ACKs would have to be explicit and noncumulative. There would be changes in retransmission management as well: the client would have to maintain a list of the requests that hadn't yet been answered and the server would have to maintain a list of replies that had been sent but not acknowledged. Some bound on the size of these lists (corresponding to window size) would be necessary.

42. (a) BLAST increments MID for each message; CHAN increments MID for each message with the same CID. Thus, the two can be synchronized only if there is only a single active channel.

    (b) CHAN's MID must be sequential in order for implicit acknowledgements to work. BLAST's MID is simply a unique identifier; it could be chosen at random or on a clock-driven basis.

43. (a) The probability that the last fragment is lost (and thus that the LAST_FRAG timer expires) is the same as the probability any other particular fragment is lost: 10%

    (b) The probability that none of the first five fragments was lost is 0.9^5, or .59; this leaves a 41% chance one or more was lost. Multiplying by 0.9 we get 36% as the probability one of the first five fragments was lost and the sixth was not.

    (c) The probability that all fragments are lost is $0.1^6 = 10^{-6}$.

44. (a) The client sends the request. The server executes it (and successfully commits any resulting changes to disk), but then crashes just before sending its reply. The client times out and resends the request, which is executed a second time by the server as it restarts.

    (b) The tipoff to the client that this *might* have happened is that the server's BID field incremented over that from the previous request (which would always cause the RPC call to fail). While a server reboot would always be indicated by an incremented BID, it would not necessarily be the case that any particular request was actually executed twice.

45. We will use the log blocks to maintain a "transaction log", a simplified version of the strategy used by database applications. In this particular example the actual update is atomic; if two data blocks had to be updated together we would have additional complications.

    Upon receipt of the request, the RPC server does the following:

    - reads in block N from the disk.
    - records in the log block the CID and MID values, the values of X and N, and an indication that the transaction is in progress.
    - performs the actual update write.
    - replaces the log entry with one that contains CID and MID and an indication that the operation was successful, and sends the reply stating this.

This last logfile record is retained until the client ACKs the reply.

On restart the server looks in the log block. If this indicates nothing about the transaction, then either the transaction was never started or else the final ACK was received; either way, the RPC server has no further immediate responsibilities. If the log block indicates that the transaction completed successfully, we reload its status as completed but unacknowledged. The server doesn't know if the reply has been sent, but this doesn't matter as it will be retransmitted if necessary when the appropriate timeout occurs. If such a retransmission was unnecessary, then the client will infer this from the expired MID.

Finally, if the restarting server finds the in-progress indication in the log, then it reads data block N and determines, by comparing X there with the X in the log, whether the write operation completed. If so, the log is updated as in the fourth step above; if not, the server resumes the sequence above at the third step, the point of performing the actual write.

46. (a) If a client has only sent the request once, and has received a reply, and if the underlying network never duplicates packets, then the client can be sure its request was only executed once.

    (b) To ensure at-most-once semantics a server would have to buffer a reply with a given transaction XID until it had received an acknowledgement from the client that the reply had been received properly. This would entail adding such ACKs to the protocol, and also adding the appropriate buffering mechanism to the implementation.

47. One TCP connection can manage multiple outstanding requests, and so is capable of supporting multiple logical channels; we will assume that this is the case. The alternative, of one TCP connection per channel, is similar.

    The overlying RPC protocol would need to provide a demultiplexing field corresponding to CID. (In the one-TCP-connection-per-channel setting, the TCP socketpair defining the connection represents the CID.) The MID would correspond to the sequence number; the primary purpose of the MID is to keep track of acknowledgements. BID is dealt with by the stateful nature of TCP; if either end rebooted and the other end eventually sent a message, the RST response would be an indication of that reboot.

    The RPC REQ and REP packets would now become RPC headers that divide the TCP byte stream into discrete messages. There would be no guarantee, of course, that these headers were transmitted in the same segment as the associated data. The RPC ACK would be replaced by the TCP ACK. PROBEs would still have to be generated by the client, if they were desired; although TCP does support KeepAlive messages they are for a vastly different ($\sim$2-hour) time scale and they do not address the issue of whether the server process is alive.

    If the TCP layer delayed sending ACKs for, say, 100ms (such "Delayed ACKs" are standard practice), then in many instances the reply might be available in time for it to be sent with the TCP ACK. This would achieve the effect of implicit ACKs in having only one packet handle both ACK and reply.

48. (a) IP and BLAST headers together account for 40 bytes, leaving 1460 bytes per packet for data. This would require 22 full packets and one 648-byte packet. (A real example would probably also include one CHAN and one SELECT header; these would increase the size of the last packet by the size of the additional headers.)

(b) i. 32KB is approximately 260 Kbits, and takes about 26 ms to send at 10 Mbps Ethernet speed.

ii. Fragmented by BLAST, the pieces would be delayed one fragment's worth, or 1.5ms, in addition to the time in (i), for a total of 27.5 ms.

Solutions for Chapter 6

1. (a) From the application's perspective, it is better to define flows as process-to-process. If a flow is host-to-host, then an application running on a multi-user machine may be penalized (by having its packets dropped) if another application is heavily using the same flow. However, it is much easier to keep track of host-to-host flows; routers need only look at the IP addresses to identify the flow. If flows are process-to-process (i.e. end-to-end), routers must also extract the TCP or UDP ports that identify the endpoints. In effect, routers have to do the same demultiplexing that is done on the receiver to match messages with their flows.

   (b) If flows are defined on a host-to-host basis, then FlowLabel would be a hash of the host-specific information; that is, the IP addresses. If flows are process-to-process, then the port numbers should be included in the hash input.

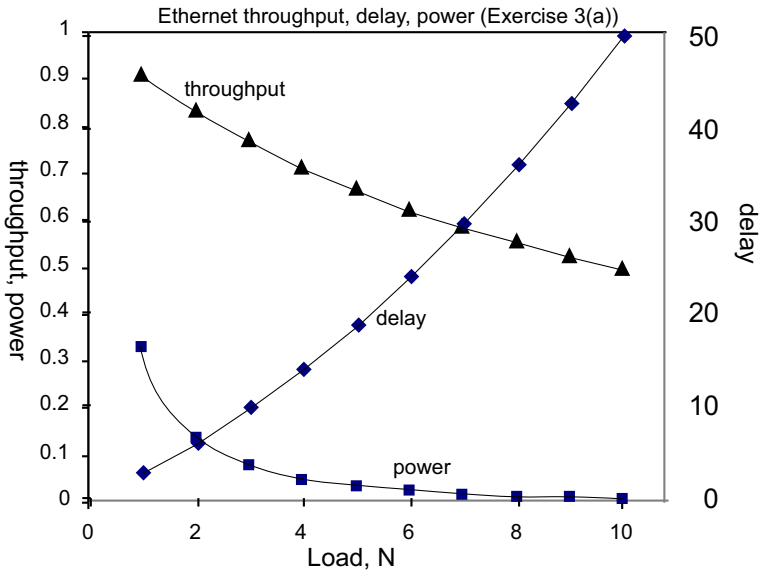
2. (a) In a rate-based TCP the receiver would advertise a rate at which it could receive data; the sender would then limit itself to this rate, perhaps making use of a token bucket filter with small bucket depth. Congestion-control mechanisms would also be converted to terms of throttling back the rate rather than the window size. Note that a window-based model sending one windowful per RTT automatically adjusts its rate inversely proportional to the RTT; a rate-based model might not. Note also that if an ACK arrives for a large amount of data, a window-based mechanism may immediately send a burst of a corresponding large amount of new data; a rate-based mechanism would likely smooth this out.

   (b) A router-centric TCP would send as before, but would receive (presumably a steady stream of) feedback packets from the routers. All routers would have to participate, perhaps through a connection-oriented packet-delivery model. TCP's mechanisms for inferring congestion from changes in RTT would all go away.

   TCP might still receive some feedback from the receiver about its rate, but the receiver would only do so as a "router" of data to an application; this is where flow control would take place.

3. (a) For Ethernet, throughput with N stations is $5/(N/2 + 5) = 10/(N + 10)$; to send one useful packet we require $N/2$ slots to acquire the channel and 5 slots to transmit. On average, a waiting station has to wait for about half the others to transmit first, so with N stations the delay is the time it takes for $N/2$ to transmit; combining this with a transmission time of $N/2{+}5$ this gives a total delay of $N/2{\times}(N/2{+}5) = N(N{+}10)/4$. Finally, power is throughput/delay = $40/N(N + 10)^2$. Graphs are below.

   (b) Because at least one station is always ready to send, token-ring throughput is constant, which we take to be 1 packet per unit time. Delay with N stations is again the time it takes for $N/2$ to send, which with the normalization above is $N/2$ time units. Power is thus $2/N$. The graph is omitted.



Both power curves have their maximum at $N = 1$, the minimum N, which is somewhat artificial and is an artifact of the unnatural way we are measuring load.

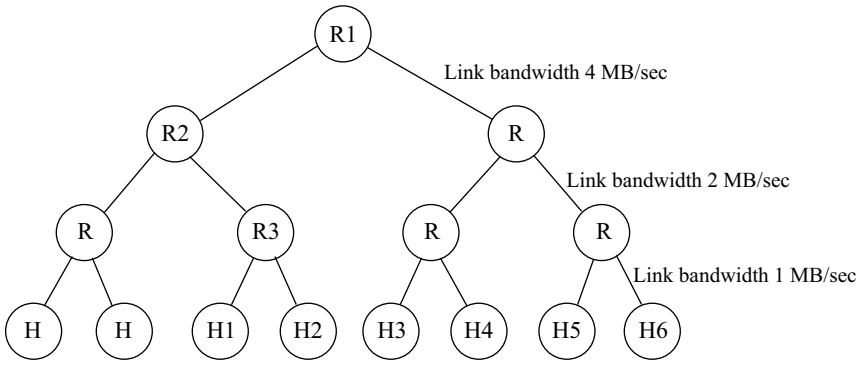
4. Throughput here is $\min(x, 1)$, where x is the load. For $x \le 1$ the delay is 1 second, constantly. We cannot sustain $x > 1$ at all; the delay approaches infinity. The power curve thus looks like $y = x$ for $x \le 1$ and is undefined beyond that.

   Another way to measure load might be in terms of the percentage of time the peak rate exceeds 1, assuming that the average rate remains less than 1.

5. Yes, particularly if the immediate first link is high-bandwidth, the first router has a large buffer capacity, and the delay in the connection is downstream. CongestionWindow can grow arbitrarily; the excess packets will simply pile up at the first router.

6. R1 cannot become congested because traffic arriving at one side is all sent out the other, and the bandwidths on each side are the same.

   We now show how to congest only the router R2 that is R1's immediate left child; other R's are similar.



We arrange for H3 and H4 to send 1MB/sec to H1, and H5 and H6 to send 1MB/sec to H2. Each of the links to the right of R1 reaches its maximum capacity, as does the R1—R2 link,

but none of these routers becomes congested. However, R2 now wants to send 4MB/sec to R3, which it cannot.

R3 is not congested as it receives at 2MB/sec from R2 and this traffic is evenly divided between H1 and H2.

7. (a) The fairness index is 0.926; $x_1 + \cdots + x_5 = 515$ and $x_1^2 + \cdots + x_5^2 = 57225$.

   (b) The index falls to 0.361.

8. F_i still represents a timestamp, but now when computing F_i as a packet arrives we run the clock slow by the sum of the weights w of the active flows, rather than by the number of active flows.

   Consider two flows with weights 1 and 2. If the the packet size of the packet in the queue for flow 2 is twice that of the packet in flow 1, then both packets should look equally attractive to transmit. Hence, the effective packet size of the second packet should be $P_i/2$. In general, if the flow has a weight w then the effective packet size is P_i/w. Hence the final time-stamps are calculated as

$$F_i = \max(F_{i-1}, A_i) + P_i/w$$

9. If we are in the process of transmitting a large sized packet and a small packet arrives just after the start of the transmission, then due to non-preemption the small packet gets transmitted after the large. However, in perfect bit-by-bit round robin the small packet would have finished being transmitted before the large packet gets completely transmitted.

10. (a) First we calculate the finishing times F_i. We don't need to worry about clock speed here since we may take $A_i = 0$ for all the packets. F_i thus becomes just the cumulative per-flow size, *ie* $F_i = F_{i-1} + P_i$.

Packet	size	flow	F_i
1	100	1	100
2	100	1	200
3	100	1	300
4	100	1	400
5	190	2	190
6	200	2	390
7	110	3	110
8	50	3	170

   We now send in increasing order of F_i:
   Packet 1, Packet 7, Packet 8, Packet 5, Packet 2, Packet 3, Packet 6, Packet 4.

   (b) To give flow 2 a weight of 2 we divide each of its F_i by 2, *ie* $F_i = F_{i-1} + P_i/2$; again we are using the fact that there is no waiting.

Packet	size	flow	weighted F_i
1	100	1	100
2	100	1	200
3	100	1	300
4	100	1	400
5	190	2	95
6	200	2	195
7	110	3	110
8	50	3	170

Transmitting in increasing order of the weighted F_i we send as follows:
Packet 5, Packet 1, Packet 7, Packet 8, Packet 6, Packet 2, Packet 3, Packet 4.

11. (a) The advantage would be that the dropped packets are the resource hogs, in terms of buffer space consumed over time. One drawback is the need to recompute *cost* whenever the queue advances.

    (b) Suppose the queue contains three packets. The first has size 5, the second has size 15, and the third has size 5. Using the sum of the sizes of the earlier packets as the measure of time remaining, the cost of the third packet is $5 \times 20 = 100$, and the cost of the (larger) second is $15 \times 5 = 75$. (We have avoided the issue here of whether the first packet should always have cost 0, which might be mathematically correct but is arguably a misleading interpretation.)

    (c) We again measure cost in terms of size; *ie* we assume it takes 1 time unit to transmit 1 size unit. A packet of size 3 arrives at T=0, with the queue such that the packet will be sent at T=5. A packet of size 1 arrives right after.
    At T=0 the costs are $3 \times 5 = 15$ and $1 \times 8 = 8$.
    At T=3 the costs are $3 \times 2 = 6$ and $1 \times 5 = 5$.
    At T=4 the costs are $3 \times 1 = 3$ and $1 \times 4 = 4$; *cost* ranks have now reversed.
    At T=5 the costs are 0 and 3.

12. (a) With round-robin service, we will alternate one telnet packet with each ftp packet, causing telnet to have dismal throughput.

    (b) With FQ, we send roughly equal volumes of data for each flow. There are about $552/41 \approx 13.5$ telnet packets per ftp packet, so we now send 13.5 telnet packets per ftp packet. This is better.

    (c) We now send 512 telnet packets per ftp packet. This excessively penalizes ftp.

    Note that with the standard Nagle algorithm a backed-up telnet would not in fact send each character in its own packet.

13. In light of the complexity of the solution here, instructors may wish to consider limiting the exercise to those packets arriving before, say, T=6.

    (a) For the ith arriving packet on a given flow we calculate its estimated finishing time F_i by the formula $F_i = \max\{A_i, F_{i-1}\} + 1$, where the clock used to measure the arrival times A_i runs slow by a factor equal to the number of active queues. The A_i clock is global; the sequence of F_i's calculated as above is local to each flow. A helpful observation here is that packets arrive and are sent at integral wallclock times.

The following table lists all events by wallclock time. We identify packets by their flow and arrival time; thus, packet A4 is the packet that arrives on flow A at wallclock time 4, *ie* the third packet. The last three columns are the queues for each flow for the subsequent time interval, *including* the packet currently being transmitted. The number of such active queues determines the amount by which A_i is incremented on the subsequent line. Multiple packets appear on the same line if their F_i values are all the same; the F_i values are in italic when $F_i = F_{i-1} + 1$ (versus $F_i = A_i + 1$).

We decide ties in the order flow A, flow B, flow C. In fact, the only ties are between flows A and C; furthermore, *every* time we transmit an A packet we have a C packet tied with the same F_i.

Wallclock	A_i	arrivals	F_i	sent	A's queue	B's queue	C's queue
1	1.0	A1,C1	2.0	A1	A1		C1
2	1.5	B2	2.5	C1	A2	B2	C1,C2
		A2,C2	*3.0*				
3	1.833	C3	*4.0*	B2	A2	B2	C2,C3
4	2.166	A4	*4.0*	A2	A2,A4		C2,C3
5	2.666	C5	*5.0*	C2	A4		C2,C3,C5
6	3.166	A6	*5.0*	A4	A4,A6	B6	C3,C5,C6
		B6	4.166				
		C6	*6.0*				
7	3.5	A7	*6.0*	C3	A6,A7	B6	C3,C5,C6,C7
		C7	*7.0*				
8	3.833	B8	*5.166*	B6	A6,A7	B6,B8	C5,C6,C7,C8
		C8	*8.0*				
9	4.166	A9	*7.0*	A6	A6,A7,A9	B8	C5,C6,C7,C8
10	4.5	A10	*8.0*	C5	A7,A9,A10	B8	C5,C6,C7,C8
11	4.833	B11	*6.166*	B8	A7,A9,A10	B8,B11	C6,C7,C8
12	5.166	B12	*7.166*	A7	A7,A9,A10	B11,B12	C6,C7,C8
13	5.5			C6	A9,A10	B11,B12	C6,C7,C8
14	5.833			B11	A9,A10	B11,B12	C7,C8
15	6.166	B15	*8.166*	A9	A9,A10	B12,B15	C7,C8
16				C7	A10	B12,B15	C7,C8
17				B12	A10	B12,B15	C8
18				A10	A10	B15	C8
19				C8		B15	C8
20				B15		B15	

(b) For weighted fair queuing we have, for flow C,

$$F_i = max\{A_i, F_{i-1}\} + 0.5$$

For flows A and B, F_i is as before. Here is the table corresponding to the one above.

Wallclock	A_i	arrivals	F_i	sent	A's queue	B's queue	C's queue
1	1.0	A1	2.0	C1	A1		C1
		C1	1.5				
2	1.5	A2	*3.0*	A1	A1,A2	B2	C2
		B2	2.5				
		C2	2.0				
3	1.833	C3	*2.5*	C2	A2	B2	C2,C3
4	2.166	A4	*4.0*	B2	A2,A4	B2	C3
5	2.5	C5	3.0	C3	A2,A4		C3,C5
6	3.0	A6	*5.0*	A2	A2,A4,A6	B6	C5,C6
		B6	4.0				
		C6	3.5				
7	3.333	A7	*6.0*	C5	A4,A6,A7	B6	C5,C6,C7
		C7	*4.0*				
8	3.666	B8	*5.0*	C6	A4,A6,A7	B6,B8	C6,C7,C8
		C8	*4.5*				
9	4.0	A9	*7.0*	A4	A4,A6,A7,A9	B6,B8	C7,C8
10	4.333	A10	*8.0*	B6	A6,A7,A9,A10	B6,B8	C7,C8
11	4.666	B11	*6.0*	C7	A6,A7,A9,A10	B8,B11	C7,C8
12	5.0	B12	*7.0*	C8	A6,A7,A9,A10	B8,B11,B12	C8
13	5.333			A6	A6,A7,A9,A10	B8,B11,B12	
14	5.833			B8	A7,A9,A10	B8,B11,B12	
15	6.333	B15	*8.0*	A7	A7,A9,A10	B11,B12,B15	
16				B11	A9,A10	B11,B12,B15	
17				A9	A9,A10	B12,B15	
18				B12	A10	B12,B15	
19				A10	A10	B15	
20				B15		B15	

14. (a) In slow start, the size of the window doubles every RTT. At the end of the ith RTT, the window size is 2^i KB. It will take 10 RTTs before the send window has reached 2^{10} KB = 1 MB.

    (b) After 10 RTTs, 1023 KB = 1 MB − 1 KB has been transferred, and the window size is now 1 MB. Since we have not yet reached the maximum capacity of the network, slow start continues to double the window each RTT, so it takes 4 more RTTs to transfer the remaining 9MB (the amounts transferred during each of these last 4 RTTs are 1 MB, 2 MB, 4 MB, 1 MB; these are all well below the maximum capacity of the link in one RTT of 12.5 MB). Therefore, the file is transferred in 14 RTTs.

    (c) It takes 1.4 seconds (14 RTTs) to send the file. The effective throughput is (10MB / 1.4s) = 7.1MBps = 57.1Mbps. This is only 5.7% of the available link bandwidth.

15. Let the sender window size be 1 packet initially. The sender sends an entire windowful in one batch; for every ACK of such a windowful that the sender receives, it increases its effective window (which is counted in packets) by one. When there is a timeout, the effective window is cut into half the number of packets.

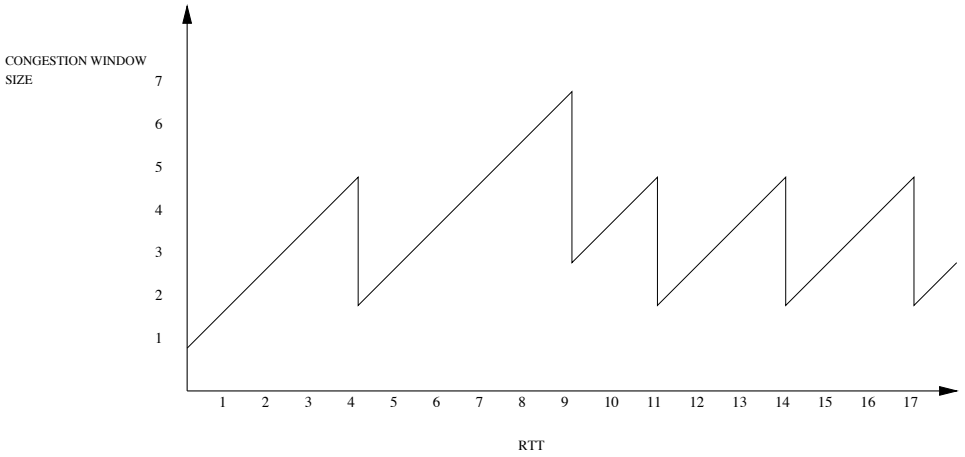
Now consider the situation when the indicated packets are lost. The window size is initially 1; when we get the first ACK it increases to 2. At the beginning of the second RTT we send packets 2 and 3. When we get their ACKs we increase the window size to 3 and send packets 4, 5 and 6. When these ACKs arrive the window size becomes 4.

Now, at the beginning of the fourth RTT, we send packets 7, 8, 9, and 10; by hypothesis packet 9 is lost. So, at the end of the fourth RTT we have a timeout and the window size is reduced to $4/2 = 2$.

Continuing, we have

RTT	5	6	7	8	9
Sent	9-10	11-13	14-17	18-22	23-28

Again the congestion window increases up until packet 25 is lost, when it is halved, to 3, at the end of the ninth RTT. The plot below shows the window size vs. RTT.



16. From the figure for the preceding exercise we see that it takes about 17 RTTs for 50 packets, including the necessary retransmissions. Hence the effective throughput is $50/17 \times 100 \times 10^{-3}$ KB/s = 29.4 KB/s.

17. The formula is accurate if each new ACK acknowledges one new MSS-sized segment. However, an ACK can acknowledge either small size packets (smaller than MSS) or cumulatively acknowledge many MSS's worth of data.

    Let $N = \mathsf{CongestionWindow}/\mathrm{MSS}$, the window size measured in segments. The goal of the original formula was so that after N segments arrived the net increment would be MSS, making the increment for one MSS-sized segment MSS/N. If instead we receive an ACK acknowledging an arbitrary AmountACKed, we should thus expand the window by

$$\begin{aligned} \text{Increment} &= \text{AmountACKed}/N \\ &= (\text{AmountACKed} \times \text{MSS})/\mathsf{CongestionWindow} \end{aligned}$$

18. We may still lose a batch of packets, or else the window size is small enough that three subsequent packets aren't sent before the timeout. Fast retransmit needs to receive three duplicate ACKs before it will retransmit a packet. If so many packets are lost (or the window size is so small) that not even three duplicate ACKs make it back to the sender, then the the mechanism cannot be activated, and a timeout will occur.

19. One would first need to identify an appropriate notion of a flow. One could do this at the CHAN level; a CHAN CID is analogous to a connection. However, such a CHAN "connection" maintains a fixed window size of one message, and, while that message can be rather large, the segmentation of the message into packets isn't done at the CHAN level. Furthermore, multiple CHAN connections between the same pair of hosts seems plausible. Thus the BLAST level might be more suitable.

    At the BLAST level a sender might maintain a count of the total number of packets outstanding between a pair of hosts, and manage this much like TCP manages CongestionWindow. This might mean sending only part of a BLAST message, and waiting for the acknowledging SRR before the rest (or next part) is sent. Alternatively, a rate-based approach might be to send all the fragments of a given message, but to increase the temporal spacing between the fragments when congestion is experienced.

    An alternative to BLAST-level congestion control might be a new abstract layer representing the union of all CHAN connections between a given pair of endpoints.

20. We will assume in this exercise and the following two that when TCP encounters a timeout it reverts to stop-and-wait as the outstanding lost packets in the existing window get retransmitted one at a time, and that the slow start phase begins only when the existing window is fully acknowledged. In particular, once one timeout and retransmission is pending, subsequent timeouts of later packets are suppressed or ignored until the earlier acknowledgement is received. Such timeouts are still shown in the tables below, but no action is taken.

    We will let Data N denote the Nth packet; Ack N here denotes the acknowledgement for data up through and *including* data N.

    (a) Here is the table of events with TimeOut = 2 sec. There is no idle time on the R–B link.

Time	A recvs	A sends	R sends	cwnd size
0		Data0	Data0	1
1	Ack0	Data1,2	Data1	2
2	Ack1	Data3,4 (4 dropped)	Data2	3
3	Ack2	Data5,6 (6 dropped)	Data3	4
4	Ack3/timeout4	Data 4	Data5	1
5	Ack3/timeout5&6		Data4	1
6	Ack5	Data 6	Data6	1
7	Ack 6	Data7,8 (slow start)	Data7	2

    (b) With TimeOut = 3 sec, we have the following. Again nothing is transmitted at T=6 because ack 4 has not yet been received.

Time	A recvs	A sends	R sends	cwnd size
0		Data0	Data0	1
1	Ack0	Data1,2	Data1	2
2	Ack1	Data3,4 (4 dropped)	Data2	3
3	Ack2	Data5,6 (6 dropped)	Data3	4
4	Ack3	Data7,8 (8 dropped)	Data5	5
5	Ack3/timeout4	Data4	Data7	1
6	Ack3/timeout5&6		Data4	1
7	Ack5/timeout7&8	Data6	Data6	1
8	Ack7	Data8	Data8	1
9	Ack8	Data9,10 (slow start)	Data9	2

21. We follow the conventions and notation of the preceding exercise. Although the first packet is lost at T=4, it wouldn't have been transmitted until T=8 and its loss isn't detected until T=10. During the final few seconds the outstanding losses in the existing window are made up, at which point slow start would be invoked.

	A recvs Ack #	cwnd size	A sends Data	R sending/R's queue	
T=0		1	1	1/	
T=1	1	2	2,3	2/3	
T=2	2	3	4,5	3/4,5	
T=3	3	4	6,7	4/5,6,7	
T=4	4	5	8,9	5/6,7,8	9 lost
T=5	5	6	10,11	6/7,8,10	11 lost
T=6	6	7	12,13	7/8,10,12	13 lost
T=7	7	8	14,15	8/10,12,14	15 lost
T=8	8	9	16,17	10/12,14,16	17 lost
T=9	8	9		12/14,16	
T=10	8	9	9	14/16,9	2nd duplicate Ack8
T=11	8			16/9	
T=12	8			9/	
T=13	10		11	11/	B gets 9
T=14	12		13	13/	
T=15	14		15	15/	
T=16	16		17	17/	
T=17	17	2	18,19	18/19	slow start

22. R's queue size is not the least bit irrelevant; this is what determines the point at which packet losses occur. We regret the confusion; we seem to have meant that R's queue size does not affect any RTT.

    We will assume that R can handle a burst of four packets when idle (one immediately transmitted and three stored).

    Notation and conventions are again as in #20 above.

	A recvs	cwnd	A sends data #	
T=0		1	1	
T=1	Ack1	2	2,3	
T=2	Ack3	4	4,5,6,7	
T=3	Ack7	8	8,9,10,11/12,13,14,15	12-15 lost
T=4	Ack11	12	16-19/20-23	window is [Data12..Data23]; 20-23 lost
T=5	Ack11		12	timeout
T=6	Ack12		13	
T=7	Ack13		14	
T=8	Ack14		15	
T=9	Ack19		20	

23. With a full queue of size N, it takes a idle period on the sender's part of N+1 seconds for R1's queue to empty and link idling to occur. If the connection is maintained for any length of time with CongestionWindow=N, no losses occur but EstimatedRTT converges to N. At this point, if a packet is lost the timeout of $2\times N$ means an idle stretch of length $2N - (N+1) = N-1$.

    With fast retransmit, this idling would not occur.

24. The router is able in principle to determine the actual number of bytes outstanding in the connection at any time, by examining sequence and acknowledgement numbers. This we can take to be the congestion window except for immediately after when the latter decreases.

    The host is complying with slow start at startup if only one more packet is outstanding than the number of ACKs received. This is straightforward to measure.

    Slow start after a coarse-grained timeout is trickier. The main problem is that the router has no way to know when such a timeout occurs; the TCP might have inferred a lost packet by some other means. We may, however, on occasion be able to rule out three duplicate ACKs, or even two, which means that a retransmission might be inferred to represent a timeout.

    After any packet is retransmitted, however, we should see the congestion window fall at least in half. This amounts to verifying multiplicative decrease, though, not slow start.

25. Slow start is active up to about 0.5 sec on startup. At that time a packet is sent that is lost; this loss results in a coarse-grained timeout at T=1.9.

    At that point slow start is again invoked, but this time TCP changes to the linear-increase phase of congestion avoidance before the congestion window gets large enough to trigger losses. The exact transition time is difficult to see in the diagram; it occurs sometime around T=2.4.

    At T=5.3 another packet is sent that is lost. This time the loss is detected at T=5.5 by fast retransmit; this TCP feature is the one not present in Figure 6.11 of the text, as all lost packets there result in timeouts. Because the congestion window size then drops to 1, we can infer that fast recovery was not in effect; instead, slow start opens the congestion window to half its previous value and then linear increase takes over. The transition between these two phases is shown more sharply here, at T=5.7.

26. We assume here that the phone link delay is due to bandwidth, not latency, and that the rest of the network path offers a bandwidth at least as high as the phone link's. During the first RTT we send one packet, due to slow start, and by the final assumption we thus transmit over the link for a third of the RTT, and thus use only a third of the total bandwidth, or 1 KB/sec. During the second RTT we send two packets; in the third and subsequent RTTs send three packets, saturating the phone link. The sequence of averages, however, climbs more slowly: at the end of the second RTT the fraction of bandwidth used is 3/6; at the end of the third RTT it is 6/9, then 9/12, at the end of the Nth RTT we have used 1-1/N of the bandwidth.

    Packet losses cause these averages to drop now and then, although since the averages are cumulative the drops are smaller and smaller as time goes on.

27. (a) Here is how a connection startup might progress:
        Send packet 1
        Get ack 1
        Send packets 2 & 3
        Get ack 2
        Send packet 4, which is lost due to link errors, so CongestionWindow=1.

    One way or another, we get lots of coarse-grained timeouts when the window is still too small for fast retransmit. We will never be able to get past the early stages of slow start.

    (b) Over the short term such link losses cannot be distinguished from congestion losses, unless some router feedback mechanism (eg ICMP Source Quench) were expanded and made more robust. (Over the long term, congestion might be expected to exhibit greater temporal variability, and careful statistical analysis might indicate when congestion was present.)

    (c) In the presence of explicit congestion indications, TCP might now be tuned to respond to ordinary timeout losses by simply retransmitting, without reducing the window size. Large windows could now behave normally.

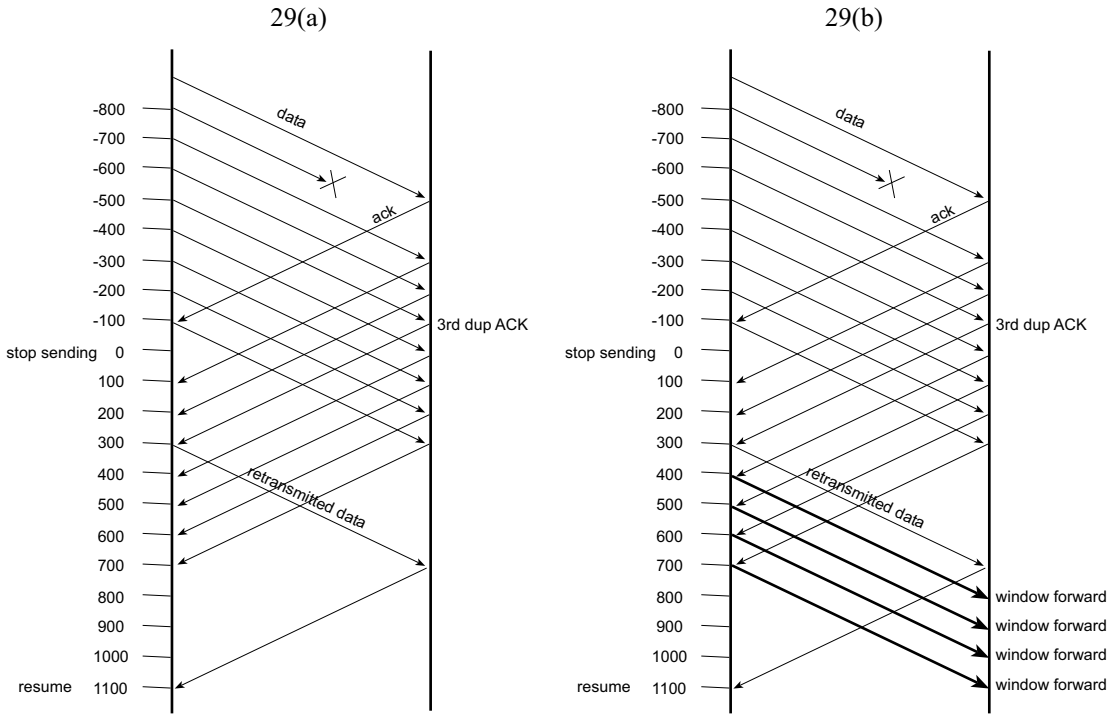
    We would, however, need some way for keeping the ACK clocking running; coarse-grained timeouts would still necessitate a return to CongestionWindow$= 1$ because ACKs would have drained. Either TCP's existing fast retransmit/fast recovery, or else some form of selective ACKs, might be appropriate. Either might need considerable tuning to handle a 25% loss rate.

28. Suppose the first two connections keep the queue full 95% of the time, alternating transmissions in lockstep and timed so that their packets always arrive just as a queue vacancy opens. Suppose also that the third connection's packets happen always to arrive when the queue is full. The third connection's packets will thus be lost, whether we use slow start or not. The first two connections will not be affected.

    Congestion avoidance by the first two connections means that they will eventually try a window size of 4, and fall back to 2, and give the third connection a real foot in the door. Slow start for the third connection would mean that if *a* packet got through, then the window would expand to 2 and the third sender would have about twice the probability of getting at least one packet through. However, since a loss is likely, the window size would soon revert to 1.

29. (a) We lose 1100 ms: we wait 300 ms initially to detect the third duplicate ACK, and then one full 800 ms RTT as the sender waits for the ACK of the retransmitted segment. If the lost packet is sent at T=-800, the lost ACK would have arrived at T=0. The duplicates arrive at T=100, 200, and 300. We retransmit at T=300, and the ACK finally arrives at T=1100.

    (b) We lose $1100 - 400 = 700$ ms. As shown in the diagram, the elapsed time before we resume is again 1100 ms but we have had four extra chances to transmit during that interval, for a savings of 400 ms.



30. We might alternate between congestion-free backoff and heavy congestion, moving from the former to the latter in as little as 1 RTT. Moving from congestion back to no congestion unfortunately tends not to be so rapid.

    TCP Reno also oscillates between congestion and noncongestion, but the periods of noncongestion are considerably longer.

31. Marking a packet allows the endpoints to adjust to congestion more efficiently—they may be able to avoid losses (and timeouts) altogether by slowing their sending rates. However, transport protocols must be modified to understand and account for the congestion bit. Dropping packets leads to timeouts, and therefore may be less efficient, but current protocols (such as TCP) need not be modified to use RED. Also, dropping is a way to rein in an ill-behaved sender.

32. (a) We have

$$\mathsf{TempP} = \mathsf{MaxP} \times \frac{\mathsf{AvgLen\_MinThreshold}}{\mathsf{MaxThreshold\_MinThreshold}}.$$

AvgLen is halfway between MinThreshold and MaxThreshold, which implies that the fraction here is $1/2$ and so TempP = MaxP/2 = 0.01.

We now have $\mathsf{P}_{\mathsf{count}} = \mathsf{TempP}/(1 - \mathsf{count} \times \mathsf{TempP})$ = 1/(100−count). For count=1 this is 1/99; for count=50 it is 1/50.

(b) Evaluating the product $(1 - \mathsf{P}_1) \times \cdots \times (1 - \mathsf{P}_{50})$ gives

$$\frac{98}{99} \times \frac{97}{98} \times \frac{96}{97} \times \cdots \times \frac{50}{51} \times \frac{49}{50}$$

which all telescopes down to 49/99, or 0.495.

33. The difference between MaxThreshold and MinThreshold should be large enough to accommodate the average increase in the queue length in one RTT; with TCP we expect the queue length to double in one RTT, at least during slow start, and hence want MaxThreshold to be at least twice MinThreshold. MinThreshold should also be set at a high enough value so that we extract maximum link utilization. If MaxThreshold is too large, however, we lose the advantages of maintaining a small queue size; excess packets will simply spend time waiting.

34. Only when the *average* queue length exceeds MaxThreshold are packets automatically dropped. If the average queue length is less than MaxThreshold, incoming packets may be queued even if the real queue length becomes larger than MaxThreshold. The router must be able to handle this possibility.

35. It is easier to allocate resources for an application that can precisely state its needs, than for an application whose needs vary over some range. Bursts consume resources, and are hard to plan for.

36. Between MinThreshold and MaxThreshold we are using the drop probability as a signaling mechanism; a small value here is sufficient for the purpose and a larger value simply leads to multiple packets dropped per TCP window, which tends to lead to unnecessarily small window sizes.

    Above MaxThreshold we are no longer signaling the sender. There is no logical continuity intended between these phases.

37. (a) Assume the TCP connection has run long enough for a full window to be outstanding (which may never happen if the first link is the slowest). We first note that each data packet triggers the sending of exactly one ACK, and each ACK (because the window size is constant) triggers the sending of exactly one data packet.

    We will show that two consecutive RTT-sized intervals contain the same number of transmissions. Consider one designated packet, P1, and let the first RTT interval be from just before P1 is sent to just before P1's ACK, A1, arrives. Let P2 be the data packet triggered by the arrival of A1, let A2 be the ACK for P2, and let the second interval be from just before the sending of P2 to just before the receipt of A2. Let N be the number of segments sent within the first interval, *ie* counting P1 but not P2. Then, because packets don't cross, this is the number of ACKs received during the second RTT interval, and these ACKs trigger the sending of exactly N segments during the second interval as well.

(b) The following shows a window size of four, but only two packets sent per RTT once the steady state is reached. It is based on an underlying topology A—R—B, where the A–R link has infinite bandwidth and the R–B link sends one packet per second each way. We thus have RTT=2 sec; in any 2-second interval beginning on or after T=2 we send only two packets.

T=0 send data[1] through data[4]
T=1 data[1] arrives at destination; ACK[1] starts back
T=2 receive ACK[1], send data[5]
T=3 receive ACK[2], send data[6]
T=4 receive ACK[3], send data[7]

The extra packets are, of course, piling up at the intermediate router.

38. The first time a timed packet takes the doubled RTT, TCP Vegas still sends one windowful and so measures an ActualRate = CongestionWindow/RTT_{new} of half of what it had been, and thus about half (or less) of ExpectedRate. We then have Diff = ExpectedRate−ActualRate $\approx (1/2)\times$ExpectedRate, which is relatively large (and, in particular, larger than β), so TCP Vegas starts reducing CongestionWindow linearly. This process stops when Diff is much closer to 0; that is, when CongestionWindow has shrunk by a factor close to two.

The ultimate effect is that we underestimate the usable congestion window by almost a factor of two.

39. (a) If we send 1 packet, then in either case we see a 1 sec RTT. If we send a burst of 10 packets, though, then in the first case ACKs are sent back at 1 sec intervals; the last packet has a measured RTT of 10 sec. The second case gives a 1 sec RTT for the first packet and a 2 sec RTT for the last.

The technique of packet-pairs, sending multiple instances of two consecutive packets right after one another and analyzing the minimum time difference between their ACKs, achieves the same effect; indeed, packet-pair is sometimes thought of as a technique to find the minimum path bandwidth. In the first case, the two ACKs of a pair will always be 1 second apart; in the second case, the two ACKs will sometimes be only 100 ms apart.

(b) In the first case, TCP Vegas will measure RTT = 3 as soon as there is a full window outstanding. This means ActualRate is down to 1 packet/sec. However, BaseRTT is 1 sec, and so ExpectedRate = CongestionWindow/BaseRTT is 3 packets/sec. Hence, Diff is 2 packets/sec, and CongestionWindow will be decreased.

In the second case, when a burst of three packets is sent the measured RTTs are 1.0, 1.1, 1.2. Further measurements are similar. This likely does not result in enough change in the measured RTT to decrease ActualRate sufficiently to trigger a decrease in CongestionWindow, and depending on the value of α may even trigger an increase. At any rate, ActualRate decreases much more slowly than in the first case.

40. An ATM network may be only one network, or one type of network, in an internet. Making service guarantees across such an ATM link does not in this setting guarantee anything on an end-to-end basis. In other words, congestion management is an end-to-end issue.

If IP operates exclusively over ATM, then congestion management at the ATM level may indeed address total congestion (although if partial packet discard is not implemented then

dropped cells do not correspond very well to dropped packets). In this setting, congestion control at the TCP level has the drawback that it doesn't address other protocols, and doesn't take into account the switches' knowledge of virtual circuits.

41. We will assume AAL5 here. To implement partial packet discard, once we drop a cell we continue to drop other cells of that connection until a cell arrives that has the "last cell of the PDU" bit set in its ATM header. The switch needs to remember the connection, but that is straightforward (and small).

    To implement early packet discard, we need to wait for the first cell of a PDU, which is marginally more complex. Among other things, we now must address from which connection a packet is dropped. More significantly, a useful EPD implementation would also require a mechanism to notice when the switch is getting busy; the point of EPD, after all, is early warning. A free-buffer threshold might be the simplest such mechanism.

42. (a) Robot control is naturally realtime-intolerant: the robot can not wait indefinitely for steering control if it is about to crash, and it can not afford to lose messages such as "halt", "set phasars on stun", or even "switch to blue paint". Such an application could be adaptive in a setting where we have the freedom to slow the robot down.

    (b) If an application tolerates a loss rate of $x, 0 < x < 1$, then it is only receiving fraction $1 - x$ of the original bandwidth and can tolerate a reduction to that bandwidth over a lossless link.

    (c) Suppose the data being transmitted are positioning coordinates for some kind of robotic device. The device must follow the positions plotted, though some deviation is permitted. We can tolerate *occasional* lost data, by interpolating the correct path (there is a continuity assumption here); this would qualify the application as loss-tolerant.

    We also want to be able to claim that the application is nonadaptive. So we will suppose that too much transmission delay means the robot cannot follow the path closely enough (or at least not with the required speed), making the application non-delay-adaptive. A significant rate reduction, similarly, might mean the device can't keep to within the required tolerance – perhaps it requires at least 80% of the coordinates – and so it would qualify as non-rate-adaptive.

43. (a) If we start with an empty bucket but allow the bucket volume to become negative (while still providing packets), we get the following table of bucket "indebtedness": At T=0, for example, we withdraw 5 packets and deposit 2.

Time, secs	0	1	2	3	4	5
Bucket volume	-3	-6	-5	-3	-7	-6

    We thus need an initial bucket depth of 7, so as not to run out at T=4. Because all the volumes above are negative, the bucket with depth 7 never overflows.

    (b) If we do the same thing as above we get

Time, secs	0	1	2	3	4	5
Bucket volume	-1	-2	1	5	3	6

    A bucket depth of 2 will thus accommodate T=1. If we start with an initially full bucket of depth 2, we get

Time, secs	0	1	2	3	4	5
Bucket volume	1	0	*2*	*2*	0	2

Entries in italic represent filling the bucket, and it turns out that due to this truncation we scrape the bottom at T=4 as well. However, the depth of 2 does suffice.

44. (a) If the router queue is empty and all three flows dump their buckets at the same time, the burst amounts to 15 packets for a maximum delay of 1.5 sec. Since the router can keep up with packets due to steady-state traffic alone, and can drain any earlier bucket dumps faster than the buckets get refilled, such a burst is in fact the maximum queue.

    (b) In 2.0 seconds the router can forward 20 packets. If flow1 sends an initial burst of 10 at T=0 and another single packet at T=1, and flow2 sends 4 at T=0 and 2 at T=1, that amounts to 17 packets in all. This leaves a minimum capacity of 3 packets for flow3. Over the long term, of course, flow3 is guaranteed an average of 8 packets per 2.0 seconds.

45. (a) If the router was initially combining both reserved and nonreserved traffic into a single FIFO queue, then reserved flows before the loss were not getting genuine service guarantees. After the loss the router is still handling all traffic via a single FIFO queue; the only difference is that all traffic is now considered nonreserved. The state loss should thus make no difference.

    (b) If the router used weighted fair queuing to segregate reserved traffic, then a state loss may lead to considerable degradation in service, because the reserved traffic now is forced to compete on an equal footing with *hoi polloi* traffic.

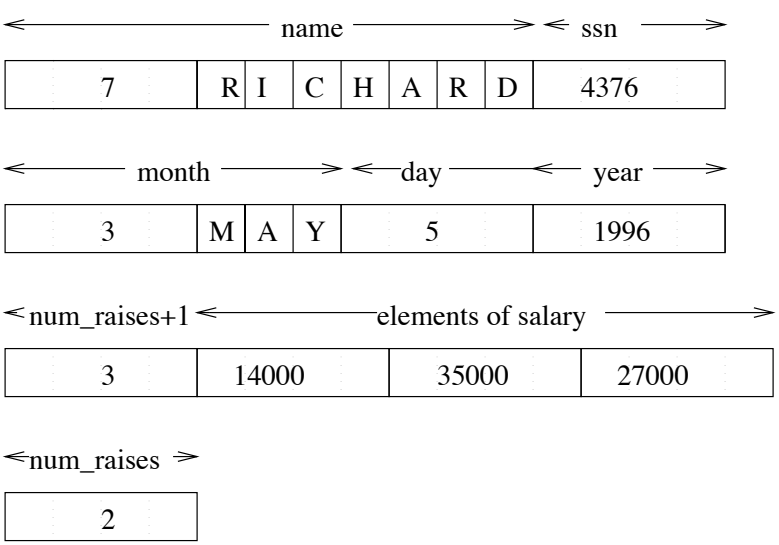
    (c) Suppose new reservations from some third parties reach the router before the periodic refresh requests are received to renew the original reservations; if these new reservations use up all the reservable capacity the router may be forced to turn down the renewals.

46. (a) S2 now faces the prospect of considerable buffering, as H1 now has license to send at a rate higher than S2 is allowed to forward towards H2.

    (b) This amounts to a single RM cell traversing the entire circuit, and eliminates the responsiveness we had sought to introduce by segmentation of control loops.

Solutions for Chapter 7

1.



4. Limited measurements suggest that, at least in one particular setting, use of htonl slows the array-converting loop down by about a factor of two.

5. The following measurements were made on a 300Mhz Intel system, compiling with Microsoft's Visual C++ 6.0 and optimizations turned off. We normalize to the case of a loop that repeatedly assigns the same integer variable to another:

   for (i=0;i<N;i++) {j=k}

   Replacing the loop body above with j=htonl(k) made the loop take about 2.9 times longer. The following homemade byte-swapping code took about 3.7 times longer:

```
char * p = (char *) & k;
char * q = (char *) & j;
q[0]=p[3];
q[1]=p[2];
q[2]=p[1];
q[3]=p[0];
```

   For comparison, replacing the loop body with an array copy A[i]=B[i] took about 2.8 times longer.

6. ASN.1 encodings are as follows:

INT	4	21645

INT	4	10120

INT	4	101

7. Here are the big-endian and little-endian representations for 21645, 10120 and 101.

1000 1111	0101 0100	0000 0000	0000 0000	little endian
0000 0000	0000 0000	0101 0100	1000 1101	big endian

1000 1000	0010 0111	0000 0000	0000 0000	little endian
0000 0000	0000 0000	0010 0111	10001000	big endian

0110 0101	0000 0000	0000 0000	0000 0000	little endian
0000 0000	0000 0000	0000 0000	0110 0101	big endian

For more on big-endian versus little-endian we quote Jonathan Swift, writing in *Gulliver's Travels*:

> ...Which two mighty powers have, as I was going to tell you, been engaged in a most obstinate war for six and thirty moons past. It began upon the following occasion. It is allowed on all hands, that the primitive way of breaking eggs before we eat them, was upon the larger end: but his present Majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the Emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. The people so highly resented this law, that our histories tell us there have been six rebellions raised on that account.... Many hundred large volumes have been published upon this controversy: but the books of the Big-Endians have been long forbidden, and the whole party rendered incapable by law of holding employments.

8. The problem is that we don't know whether the RPCVersion field is in big-endian or little-endian format until after we extract it, but we need this information to decide on which extraction to do.

   It would be possible to work around this problem provided that among all the version IDs assigned, the big-endian representation of one ID never happened to be identical to the little-endian representation of another. This would be the case if, for example, future versions of XDR continued to use big-endian format for the RPCVersion field, but not necessarily elsewhere.

9. It is often possible to do a better job of compressing the data if one knows something about the type of the data. This applies even to lossless compression; it is particularly true if lossy compression can be contemplated. Once encoded in a message and handed to the encoding layer, all the data looks alike, and only a generic, lossless compression algorithm can be applied.

10. [The DEC-20 was perhaps the best-known example of 36-bit architecture.]

    Incoming 32-bit integers are no problem; neither are outbound character strings. Outbound integers could either be sent as 64-bit integers, or else could lose the high-order bits (with or without notification to the sender). For inbound strings, one approach might be to strip them to 7 bits by default, make a flag available to indicate whether any of the eighth bits had been set, and, if so, make available a lossless mechanism (perhaps one byte per word) of re-reading the data.

11. Here is a C++ solution, in which we make netint⇒int an automatic conversion. To avoid potential ambiguity, we make use of the explicit keyword in the constructor converting int to netint, so that this does not also become an automatic conversion. (Note that the ambiguity would require additional code to realize.)

    To support assignment netint = int, we introduce an assignment operator.

```
class netint {
   public:
   operator int() {return ntohl(_netint);}
   netint() :  _netint(0)                    // default constructor
   explicit netint (int n) :  _netint(ntohl(n)) {}
   netint & operator=(int n) {
         _netint = htonl(n);
         return *this;
   }
   int raw() {return _netint;}               // for testing
private:
   int _netint;
};
```

    The above strategy doesn't help at all with pointers, and not much with structures and arrays. It doesn't address alignment problems, for example.

12. Transmission bit order is the province of the network adapter, which addresses this as it transmits or receives each byte. Generally all numeric formats on the same machine (different sizes of ints, floats, etc) use the same bit order; only if they didn't would the programmer have to make distinctions.

13. For big-endian network byte order the average number of conversions is $0 \times p^2 + 1 \times 2p(1-p) + 2 \times (1-p)^2$. For receiver-makes-right this is $0 \times p^2 + 1 \times 2p(1-p) + 0 \times (1-p)^2$; that is, if both sender and receiver are little-endian then no conversion is done. These are evaluated below:

	$p = 0.1$	$p = 0.5$	$p = 0.9$
big-endian network	1.80	1.00	0.20
receiver-makes-right	0.18	0.50	0.18

14. Try data files with lots of byte-string-level redundancy.

15. (a)

letter	encoding
a	1
b	01
c	001
d	000

    (b) $1 \times 0.5 + 2 \times 0.3 + 3 \times 0.1 + 3 \times 0.1 = 1.7$

    (c) The table is the same, although we could now give either **a** or **b** the 1-bit encoding. The average compression is now $1 \times 0.4 + 2 \times 0.4 + 3 \times 0.15 + 3 \times 0.05 = 1.8$

16. (a) This is a counting argument: there are 2^N strings of length N and only $2^0 + 2^1 + \cdots + 2^{N-1} = 2^N - 1$ strings of length $< N$. Some string, therefore, cannot get shorter.

    (c) We let

$$c'(s) = \begin{cases} 0^\frown c(s) & \text{if } \text{length}(c(s)) < \text{length}(s) \\ 1^\frown s & \text{otherwise} \end{cases}$$

    (where $0^\frown c(s)$ is $c(s)$ with a zero-bit prepended). The initial bit is a flag to indicate whether the remainder was compressed or not.

17. Bytes that occur with a run length of 1 we represent with themselves. If a byte occurs in a run of more than 1, we represent it with the three bytes

    [ESC] [count] [byte]

    The byte [ESC] can be any agreed-upon escape character; if it occurs alone it might be represented as [ESC][ESC].

18. A sample program appears on the web page; it generated the following data. The uncompressed size of RFC 791 is 94892 bytes. There are 11,243 words in all; the number of distinct words is 2255 and the dictionary size is 18226 bytes. The encoded size of the non-dictionary part with 12 bits per word is thus $(12 \times 11243)/8$ = 16865 bytes; together with the dictionary we get a compressed size of 35091 bytes, 37% of the original size. There are 132 words appearing at least 13 times, with a total frequency count of 6689. This means the 128 most common words appear a total of 6637 times; this gives a compressed size of $(8 \times 6637 + 13 \times (11243 - 6637))/8$ = 14122 bytes, plus the dictionary; the total is now

34% of the original size. Note that exact numbers are sensitive to the precise definition of a "word" used.

19. An Excel spreadsheet for this, dct.xls, is available at the web site.

    (a) For "symmetric" data such as this, coefficients $DCT(i)$ for $i = 1, 3, 5, 7$ (starting the indexing at $i = 0$) should be zero or near-zero.

    (b) If we keep six coefficients, the maximum error in $pixel(i)$ after applying the DCT and its inverse is about 0.7%, for $i = 1$ and $i = 2$. If we keep only four or five coefficients (note that both choices lead to the same values for the inverse DCT), then the maximum error is 6%, at $i = 0$; the error at $i = 1$ is 5.6%.

    (c) Here is a table listing, for each s_i, the percentage error in the ith place of the final result. The smallest error is for $i = 0$ and 7; the largest is for i = 1 and 6.

i	0	1	2	3	4	5	6	7
% error	12.3	53.1	39.6	45.0	45.0	39.6	53.1	12.3

20. The all-white image generates all zeros in the DCT phase. The quantization phase leaves the 8×8 grid of 0's unchanged; the encoding phase then compresses it to almost nothing.

21. Here is the first row of an 8×8 pixmap consisting of a black line (value 0) in the first column, and the rest all white:
    00 FF FF FF FF FF FF FF

    Here is the image after default-quality (cjpeg -quality 75) JPEG compression and decompression; there is some faint vertical fringing (the columns with FC, FD, and FE would appear as progressively fainter grey lines). All rows are identical; here is the first:
    01 FC FF FD FF FF FE FF
    With -quality 100, or even -quality 90, the fringing is gone; the image after compression and decompression is identical to the original.

22. We start with an example specifically generated for the 8×8 grid; note that the letters change gradually in both directions. Here is the original data:
    a b c d e f g h
    b c d e f g h i
    c d e f g h i j
    d e f g h i j k
    e f g h i j k l
    f g h i j k l m
    g h i j k l m n
    h i j k l m n o
    We get the following after default-quality (quality=75) jpeg compression and decompression; no letter is off by more than 1 ascii value.

b	b	c	d	e	g	h	h
b	c	d	e	f	g	h	h
c	d	d	f	g	h	i	i
d	e	f	g	h	i	j	j
f	f	g	h	i	j	k	l
g	g	h	i	j	l	l	m
h	h	i	j	k	l	m	n
h	h	i	k	l	m	n	n

At quality=100 the text is preserved exactly. However, this is the best case.

Here is the first line of Lincoln's Gettysburg Address,

Fourscore and seven years ago our fathers brought forth on this continent....,

compressed and decompressed. With spaces between words eliminated and everything made lowercase, at quality=75 we get:

hnruugdtdihjpkirmqicjlfgowpekoifappiosqrbnnjonkppqjioidjulafrnhq

At quality=100 we get:

fourscoreandsevenyeassagoovrfathersbroughtfnrthonthiscontinentan

The three errors are underlined. Leaving in the spaces, then at quality=100 we get:

fourscpre and seven years ago our fathers bsought eosth_on this

where the "_" character is the character with decimal value 31, versus 32 for a space character.

Lowercase letters are all within the ascii range 97-122 numerically. Space characters are numerically 32; these thus appear to the DCT as striking discontinuities.

23. Jpeg includes an encoding phase, but as this is lossless it doesn't affect the image. Jpeg's quantization phase, however, potentially rounds off all coefficients, to a greater or lesser degree; the strategy here however leaves un-zeroed coefficients alone.

24. Recalling that $C(0) = 1/\sqrt{2}$, we have

$$\begin{aligned} DCT(0,0) &= \frac{1}{2\sqrt{2N}} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \\ &= \frac{N\sqrt{N}}{2\sqrt{2}} \times \frac{1}{N^2} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \\ &= \frac{N\sqrt{N}}{2\sqrt{2}} \times (\text{average of the } pixel(x,y)\text{'s}) \end{aligned} \tag{2}$$

26. If you display I frames only, then the fast-forward speed is limited to the rate at which I-frames are included; note that this may be variable.

    The worst case for decoding an arbitrary frame is when the frame you want is a B frame. It depends on a previous P frame P and a future P or I frame Q. To decode the B frame you want, you will first need P and its I frame, and also Q. If Q is a P frame, then its I frame is the same as that of P. The total number of frames processed, including the one wanted, is four.

29. (a) For a while, the intervening B frames would show each macroblock containing a point as an appropriately translated macroblock from the original I frame. The δ for the frame

is zero. Once the two points were close enough that they were in the same macroblock, however, we would need a nonzero δ to represent the frame, perhaps translating a macroblock from the original I frame so as to show one point, and using a δ with one nonzero entry to show the second point.

(b) If the points were of a fixed color, the only difference from the above is that color macroblocks come at a different resolution. With points that are *changing* in color, modest deltas are needed from the beginning to indicate this.

Solutions for Chapter 8

2. The transformation is the same: $R_{i-1} = L_i$ and $L_{i-1} = R_i \oplus F(R_{i-1}, K_i)$.

3. We first note that the S box given implies that to get $R_i = F(R_{i-1}, K_i)$ we simply xor the middle 4 bits of each 6-bit chunk of K_i with the corresponding 4-bit chunk of R_{i-1}. In bits, K_i is

   1010 0101 1011 1101 1001 0110 1000 0110 0000 1000 0100 0001

   or with the middle four bits separated from each 6-bit piece, with deadbeef written below:

   1 0100 10 1101 11 1011 00 1011 01 0000 11 0000 01 0000 10 0000 1
   1101 __ 1110 __ 1010 __ 1101 __ 1011 __ 1110 __ 1110 __ 1111

   R_i is then the xor of the corresponding 4-bit chunks, or

   1001 0011 0001 0110 1011 1110 1110 1111

   or, in hex, 9516beef. Note that the second half of K_i didn't do much encrypting.

4. (a) We have $R_1 = L_0 \oplus F(R_0, K_1)$. Knowing R_1 and L_0 allows us to recover $F(R_0, K_1)$. Because of the simplified S box, this is just the xor of R_0 and K_1', where the latter is the concatenation of the middle 4 bits of every 6-bit chunk from K_1. We now have K_1', and hence 32 of the 48 bits of K_1.

   We now note this gives us 32 of the 56 bits of K. To get from the full key K to K_1, we first rotated each half of K, and then applied the DES compression permutation. We can easily track single bits through both of these operations.

   (b) Because the attacker can obtain arbitrarily many ⟨plaintext,ciphertext⟩ pairs by encrypting (with the public key) on their own.

5. (a) We must find d so that $3d \equiv 1 \bmod (p-1)(q-1)$. This means $3d = 1 + 11200k$ for some k. Since $k < 3$, we quickly find $k = 2$ works and $d = 7467$.

   (b) $m^3 \equiv m^2 m \equiv 11291 \times 9876 \equiv 4906 \bmod pq$.

6. We need to show that $m^{de} = m \bmod p$. But $de = 1 \bmod (p-1)(q-1)$, so $de = 1 \bmod p-1$, so $de = 1 + k(p-1)$ for some k, so $m^{de} = m^{1+k(p-1)} = m(m^{p-1})^k$. But $m^{p-1} \equiv 1 \bmod p$ for all m not divisible by p; if m is divisible by p then $m^{de} \equiv m \equiv 0 \bmod p$.

7. (a) We express $n-1$ in binary. Let l be the length of this representation in bits; we have $l = O(\log n)$. We calculate $b^{2^i} \bmod n$ for all $i < l$; this takes $l-1$ squarings. We then multiply together all the b^{2^i} for which the ith bit of l is 1; this yields $b^{n-1} \bmod n$.

   (b) Using the method of part (a), we have $2^{50620} \equiv 45062 \not\equiv 1 \bmod 50621$. Hence 50621 is composite; we remark that $50621 = 223 \times 227$. Here is a short program to evaluate $2^{50620} \bmod 50621$:

```
#include <iostream.h>
const int modulus = 50621;
int pow(int x, int n){
   int s = x;
```

```
    int prod = 1;
    while (n>0) {
    // Invariant:  prod * s^n is constant
       if (n % 2 == 1) prod = (prod * s) % modulus;
       s = s * s % modulus;
       n = n / 2;
    }  return prod;
}
void main() {
    int i;
    cout << pow(2,modulus-1) << endl;
}
```

(c) Here is a table of values of $b_n = 2^{2^n} \bmod 561$:

n	b_n
0	2
1	4
2	16
3	256
4	460
5	103
6	511
7	256
8	460
9	103

140 is 10001100 in binary, so $2^{140} = b_7 \times b_3 \times b_2 = 256 \times 256 \times 16 \equiv 67 \bmod 561$. We then have $2^{280} \equiv 67^2 \equiv 1 \bmod 561$. This means that we have found a nontrivial square root of 1, modulo 561; specifically (using the original notation) one of form $b^{(n-1)/2^i}$ for some i. Miller's test is said to fail if we find such a root; failure here definitively refutes the primality of n.

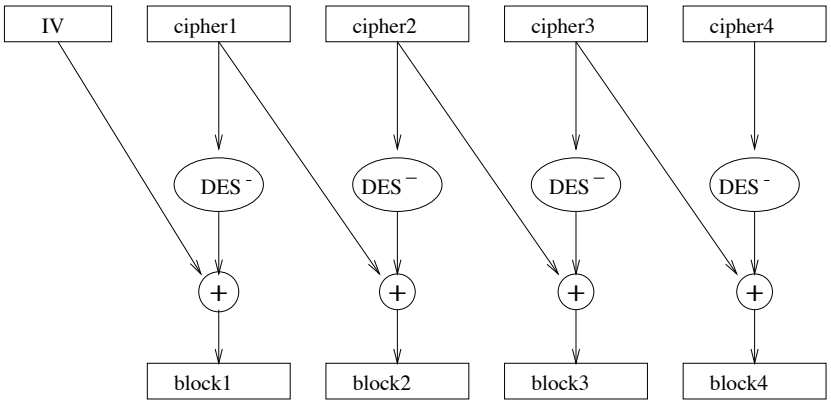
8. An eavesdropper may have intercepted the messages exchanged in a previous transaction, and may be replaying the client's earlier responses. Until the real client has successfully decrypted the server's new challenge and sent back evidence of this, the server cannot be sure such a replay attack isn't in progress.

9. (a) The client can now guess at y; it doesn't need to decrypt the server's message E(y,SHK). However, the client doesn't have CHK, and so cannot send E(y+1,CHK).

   (b) Eavesdropping doesn't help. We now know some past E(y+1,CHK) (and may even know the corresponding y), but we can't generate new values.

   (c) If we could eavesdrop and reset the clock, then we could eavesdrop on a session at server time t_1 and record the client's E(y+1,CHK). Later, at time t_2, we first reset the server's clock back to t_1. The server now will choose the same value for the clock-based y, and so will accept the replayed E(y+1,CHK) from before.

10.



11. For a two-way authentication, the one-way authentication given in the chapter is done in both directions. That is, the client sends random number x to the server encrypted with the server's public key, which the server decrypts with its private key and sends back unencrypted; at about the same time, the server encrypts random number y with the client's public key, which the client decrypts with its private key and sends back.

13. We have $password[N] = g(password[N-1])$; the essential property of g is that it be believed that knowing $g(x)$ does not provide any information that can be used to find x.

14. (a) let q be the first $N-1$ characters of the password p, of length N. The eavesdropper is in possession of q at the point indicated in the hint; we now assume that "sufficiently slowly" means that the eavesdropper can try all passwords $q^\frown ch$, for all characters ch, before the original user has typed the Nth character. If we assume passwords are restricted to 7-bit printable ascii, that's only 96 tries.

    (b) Other attacks include a compromised utility to calculate the one-time password $f(mp, N)$ from the master password mp, discovery of a way to invert the function g at least partially, eavesdropping on the initial selection of mp, and "hijacking" a connection after authentication has been completed. There are doubtless others, as well.

15. We have $c \equiv c_1 \equiv m^3 \bmod n_1$, $c \equiv c_2 \equiv m^3 \bmod n_2$, and $c \equiv c_3 \equiv m^3 \bmod n_3$. This means that $c - m^3$ is divisible by each of n_1, n_2, and n_3. Since the n_i are relatively prime, this means $c \equiv m^3 \bmod n_1 n_2 n_3$. We may take $c < n_1 n_2 n_3$, and, since $m < n_i$ for $i = 1$, 2, 3, we have $m^3 < n_1 n_2 n_3$. This now implies $c = m^3$, and hence that m is the ordinary integer-arithmetic cube root of c.

16. Because s is short, an exhaustive search conducted by generating all possible s and comparing the MD5 checksums with m would be straightforward. Sending MD5($s^\frown r$), for some random or time-dependent r, would suffice to defeat this search strategy, but note that now we would have to remember r and be able to present it later to show we knew s. Using RSA to encrypt $s^\frown r$ would be better in that sense, because we could decrypt it at any time and verify s without remembering r.

17. Each side chooses x_i privately. They exchange signatures of their respective choices as in the previous exercise, perhaps MD5($x_i^\frown r_i$) for random r_i. Then they exchange the actual x_i's

(and r_i's); because of the signatures, whoever reveals their x_i last is not able to change their choice based on knowing the other x_i. Then let $x = x_1 \oplus x_2$; as long as either party chooses their x_i randomly then x is random.

18. Let P_N be the probability that of N messages each checksum value is different from all the preceding. As in Chapter 2 Exercise 41 we have

$$P_N = \left(1 - \frac{1}{2^{128}}\right)\left(1 - \frac{2}{2^{128}}\right)\cdots\left(1 - \frac{N-1}{2^{128}}\right)$$

Taking logs and approximating we get

$$\begin{aligned} logP_N &= -(1/2^{128} + 2/2^{128} + \cdots + (N-1)/2^{128}) \\ &= -(1 + 2 + ... + (N-1))/2^{128} \\ &\approx -N^2/2^{129} \end{aligned}$$

So $P_N \approx e^{-N^2/2^{129}}$. For $N = 2^{63}$ the exponent here is $-2^{126}/2^{129} = -1/8$; for $N = 2^{64}$ and $N = 2^{65}$ it is $-1/2$ and -2 respectively. Thus, the probabilities are

$$\begin{aligned} P_{63} &= e^{-1/8} = 0.8825, \\ P_{64} &= e^{-1/2} = 0.6065, \\ P_{65} &= e^{-2} \quad = 0.1353. \end{aligned}$$

The probability two messages have the same checksum is $1 - P_N$.

19. The problem with padding each 1-byte message with seven zero bytes before encrypting is that we now are transmitting only 256 possible different encrypted blocks and a codebreaking attack is quite straightforward.

Here are some better options. Each involves encrypting a full block for each plaintext byte transmitted; the first two also require that we transmit a full block.

1. We could pad each plaintext byte with 7 random bytes before encrypting. This is quite effective, if the random bytes are truly random.

2. We could make use of cipher block chaining, as in Figure 8.7, padding each plaintext byte p_i with seven zero-bytes before xoring with the previous Cipher$_{i-1}$ block. A roughly equivalent alternative, perhaps more like the previous option, is to pad p_i with seven bytes from Cipher$_{i-1}$, and omit the xor.

3. So-called cipher-feedback (CFB) mode is sometimes used. Let c_i denote the ith encrypted byte. Given p_i, we first use DES to encrypt the block $\langle c_{i-8} \cdots c_{i-1} \rangle$, and then let c_i be the xor of p_i and the first byte of this encryption result. CFB makes assumptions about the pseudorandomness of DES output that may not be initially apparent.

20. (a) Here is one possible approach. We add the following fields to the packets (which presumably contain a Packet Type field and a Block Number field already):
    Sender's time when the connection was initiated
    Receiver's time when the connection was initiated
    Keyed MD5 checksum field

The latter consists of the MD5 checksum of everything else in the packet concatenated with the sender's own key. The packet sent does not include the sender's key. The recipient is able to recompute this checksum, because we have assumed both keys are known to both parties.

The checksum field provides both message integrity and authentication; at least it confirms that whoever created the packet knew the sender's key.

The timestamps guard against replay attacks. Both sides must exchange timestamps through a three-way handshake before any data is sent, much like ISNs in TCP. If we include only the client's timestamp, then the server could only detect replay attacks by keeping track of the previous client timestamp used. With both timestamps, each party is assured of a new connection as long as *it* has chosen a new timestamp.

(b) The timestamps guard against late packets from a prior incarnation; older incarnations would have at least one timestamp wrong. However, they do nothing to protect against sequence number wraparound within a connection.

22. (a) An internal telnet client can presumably use any port ≥ 1024. In order for such clients to be able to connect to the outside world, the firewall must pass their return, inbound traffic. If the firewall bases its filtering solely on port numbers, it must thus allow inbound TCP connections to any port ≥ 1024.

(b) If the firewall is allowed access to the TCP header Flags bits, then to block all inbound TCP connections it suffices to disallow inbound packets with SYN set but not ACK. This prevents an outside host initiating a connection with an inside host, but allows any outbound connection. No port filtering is needed.

23. (a) The FTP client uses the PORT command to tell the server what port to use for the data connection. If this port can be taken from a limited range, in which we are sure there are no other servers to which an outsider might attempt to connect, then a firewall can be configured to allow outside access to this range (by examining both port numbers and the TCP Flags) without unduly compromising security. This range cannot be limited to a single port, though, because otherwise that port would frequently be in TIME_WAIT and unavailable.

(b) Instead of using the PORT command to tell the FTP server to what client port it should connect for data transfer, an FTP client can send the server the PASV ("passive") command. The server response, assuming it supports PASV, is to send an acknowledgement containing a server port number. The client then initiates the data-transfer connection to this server port. As this is typically an outbound connection through the firewall it can be safely permitted.

24. The routers are configured as follows:

R1 blocks inbound traffic to the telnet port, unless the destination subnet is net2.

R2 blocks all telnet traffic from net 2 to net 1.

25. The ISP might want to prohibit attacks (such as the IP spoofing attack described in Exercise 5.17 or, for that matter, email spamming) launched by its own customers.

26. A filtering firewall can prevent a spoofer from masquerading as an internal host if it is configured to block inbound IP packets with source address from within the organization. However, the description in the Exercise here suggests that the internal host is to trust IP-address-based authentication of the *external* host corresponding to Remote Company User. If this is the case, then the spoofer just masquerades as Remote Company User and the organization's filtering firewall cannot do much about it.

    On the other hand, for a proxy firewall to allow internal hosts to do IP-address-based authentication of external hosts directly is somewhat unusual, as normally the internal host doesn't even see the IP address of the external host; the final sentence of the Exercise is somewhat unclear. The more usual case is for the proxy server to do (coarse-grained, *eg* subnet-based) IP-based authentication and to have the internal host trust whatever the proxy server allows to connect with it. However, the situation vis-à-vis spoofing in both cases is similar: the spoofer needs only masquerade as the Remote User to the proxy server. Suppose a spoofer succeeds at this masquerade in the second, more usual, case, and sends a request to be executed by the internal host. The proxy server accepts the request, and presumably forwards it directly to the internal host (even as the spoofed connection now dies, for lack of ACKs), which treats it as authenticated and executes it. In the first case, the same happens except that the internal host does an additional IP-address-based check. But the IP address for the spoofer that it uses is the address reported to it by the proxy server, which is that of the Remote Company User, which presumably passes muster.

Solutions for Chapter 9

1. ARP traffic is always local, so ARP retransmissions are confined to a small area. Subnet broadcasts every few minutes are not a major issue either in terms of bandwidth or cpu, so a small cache lifetime does not create an undue burden.

   Much of DNS traffic is nonlocal; limiting such traffic becomes more important for congestion reasons alone. There is also a sizable total cpu-time burden on the root nameservers. And an active web session can easily generate many more DNS queries than ARP queries. Finally, DNS provides a method of including the cache lifetime in the DNS zone files. This allows a short cache lifetime to be used when necessary, and a longer lifetime to be used more commonly.

   If the DNS cache-entry lifetime is too long, however, then when a host's IP address changes the host is effectively unavailable for a prolonged interval.

2. DNS servers will now take on ARP's role, by in effect supplying both subnet number and physical address of hosts in its domain. DNS servers must therefore now monitor hosts for possibly changed physical addresses.

   A fairly common method in IPv4 of finding ones DNS server is via static configuration, *eg* the Unix /etc/resolv.conf files. If this mechanism were still used in IPv6, changing the Ethernet address of a local DNS server would now involve considerable updating, both of the local clients and also the DNS parent. IPv6 clients, however, are likely to find their DNS server dynamically, *eg* via DHCP, instead.

3. The lookup method here requires trusting of the remote site's DNS PTR data, which may not be trustworthy. Suppose, for example, that it is known that cicada.cs.princeton.edu trusts gnat.cs.princeton.edu. A request for authentication might arrive at cicada from, say, IP address 147.126.1.15, which is *not* part of the princeton.edu domain. If cicada followed the strategy of the exercise here, it would look up the string 15.1.126.147.in-addr.arpa in the DNS PTR data. This query would eventually reach the DNS server for PTR zone 1.126.147.in-addr.arpa, which if suborned or malicious might well return the string gnat.cs.princeton.edu regardless of the fact that it had no connection with princeton.edu. Hostname strings returned by DNS servers for PTR searches are arbitrary, and need not be related to the server's assigned domain name.

   A forward DNS lookup to confirm the result of the reverse DNS lookup would, however, be reasonably safe.

4. There is little if any relationship, formally, between a domain and an IP network, although it is nonetheless fairly common for an organization (or department) to have its DNS server resolve names for all the hosts in its network (or subnet), and no others. The DNS server for cs.princeton.edu could, however, be on a different network entirely (or even on a different continent) from the hosts whose names it resolves. Alternatively, each *x*.cs.princeton.edu host could be on a different network, and each host that *is* on the same network as the cs.princeton.edu nameserver could be in a different DNS domain.

   If the reverse-mapping PTR records are used, however, then the same nameserver can handle both forward and reverse lookups only when DNS zones do correspond to groups of subnets.

5. If a host uses a nonlocal nameserver, then the host's queries don't go into the local nameserver's cache (although this is only relevant if there is some reason to believe some other local host might make use of the cached entries). Queries have farther to travel, too. Otherwise there is no penalty for having the "wrong" DNS server.

   The DNS traffic volume will be the same for the nonlocal nameserver as for a local nameserver, if the nonlocal nameserver is "on the way" to the nameserver that ultimately holds the address requested. Use of a nonlocal nameserver could result in *less* DNS traffic if the nonlocal nameserver has the entries in its cache, and isn't too far away, but local nameserver does not. This might be the case if, for example, a large group of people with similar interests all used the same nonlocal nameserver.

6. Figure 9.4 is "really" a picture of the domain hierarchy again. Nameservers have been abstracted, effectively, into one per zone (duplicates are consolidated, and a nameserver serving multiple zones would appear in multiple entries).

   Without this abstraction, a graph of all nameservers would simply be all DNS servers joined by edges corresponding to NS records, from zone parent to child. It would not necessarily be acyclic, even as a directed graph.

7. Old versions of whois are still common; typically they contain a hardcoded reference to an obsolete server. Try whois -h rs.internic.net if necessary.

   Here is an example based on arizona.edu, the domain of Larry Peterson's former institution. whois arizona.edu returns:

   ```
   University of Arizona (ARIZONA-DOM)                ARIZONA.EDU
   University of Arizona (TELCOM) ARIZONA.EDU          128.196.128.233
   ```

   To look up these individual entries, we put "!" before the names in parentheses, in the whois command. whois !ARIZONA-DOM returns

   ```
   University of Arizona (ARIZONA-DOM)
       Computer Center
       Tucson, AZ 85721
       Domain Name: ARIZONA.EDU
       Administrative Contact, Technical Contact, Zone Contact:
       ...
   ```

   The class B IP network address for the University of Arizona is 128.196; US Internet number assignments are at whois.arin.net. whois -h whois.arin.net 128.196 returns:

   ```
   University of Arizona (NET-UNIV-ARIZ)
       Center for Computing and Information Technology...
       Tucson, AZ 85721
       Netname: UNIV-ARIZ
       Netnumber: 128.196.0.0
       ...
   ```

As of late 1999, whois !TELCOM (the other "handle" returned by the whois arizona.edu search) still returns

> University of Arizona (TELCOM)
> Hostname: ARIZONA.EDU
> Nicknames: TELCOM.ARIZONA.EDU
> Address: 128.196.128.233
> ...
> Host Administrator:
> **Peterson, Larry L.**

8. One would first look up the IP address of the web server, using, say, nslookup. One would then use whois to look up who is assigned that IP address, and compare the resulting identification to that obtained by using whois to look up the webserver domain name.

9. (a) One could organize DNS names geographically (this hierarchy exists already; chi.il.us is the zone for many sites in the Chicago area), or else organize by topic or service or product type. The problems with these alternatives are that they tend to be harder to remember, and there *is* no natural classification for corporations. Geography doesn't work as large corporations are not localized geographically. Classifying by service or product has also never been successful; this changes too quickly as corporations merge or enter new areas or leave old ones.

   (b) With multiple levels there are lots more individual nameserver queries, and the levels are typically harder to remember.

10. If we just move the .com entries to the root nameserver, things wouldn't be much different than they are now, in practice. In theory, the root nameservers now could refer all queries about the .com zone to a set of .com-specific servers; in practice the root nameservers (*x*.root-servers.net for *x* from a to m) all do answer .com queries directly. (They do not, however, answer .int queries directly.) The proposal here simply makes this current practice mandatory, and shouldn't thus affect current traffic at all, although it might leave other zones such as .org and .net and .edu with poorer service someday in the future.

    The main problem with moving the host-level entries, such as for www.cisco, to a single root nameserver entry such as cisco, is that this either limits organizations to a single externally visible host, or else (if the change is interpreted slightly differently) significantly increases root nameserver traffic as it returns some kind of block of multiple host addresses. In effect this takes DNS back to a single central server. Perhaps just as importantly, the updating of the IP addresses corresponding to host names is now out of the hands of the organizations owning the host names, leading to a considerable administrative bottleneck.

    However, if we're just browsing the web and need only one address for each organization, the traffic would be roughly equivalent to the way DNS works now. (We are assuming that local resolvers still exist and still maintain request caches; the loss of local caches would put an intolerable burden on the root nameservers.)

11. DNS records contain a TTL value, specified by the DNS server, representing how long a DNS record may be kept in the client cache. RFC 1034 puts it this way:

If a change can be anticipated, the TTL can be reduced prior to the change to minimize inconsistency during the change, and then increased back to its former value following the change.

12. Strictly speaking one also needs to specify authoritative answers only; otherwise a nameserver that holds the final answer in its cache will supply that answer even if norecurse was specified. In practice it suffices to search for NS records for partial domains: edu., princeton.edu., and cs.princeton.edu., and to send each query to the nameserver returned in the NS record for the preceding query. Note also that the trailing "." shown here may not be optional.

Here is a series of nslookup queries. User input follows the > prompt, comments are in italic, and the remaining lines are edited nslookup responses.

```
> set norecurse
> set query=NS                    get NS records only
> edu.                            find nameserver for EDU. domain

edu        nameserver = A.ROOT-SERVERS.NET
...
A.ROOT-SERVERS.NET      internet address = 198.41.0.4
...                             also B.root-servers.net...M.root-servers.net

we now point nslookup to the nameserver returned above
> server a.root-servers.net
> princeton.edu.                  find nameserver for PRINCETON.EDU. domain

princeton.edu           nameserver = DNS.princeton.edu
...
DNS.princeton.edu       internet address = 128.112.129.15
...

> server dns.princeton.edu.       again, point nslookup there
> cs.princeton.edu.               ask for CS.PRINCETON.EDU. domain

cs.princeton.edu        nameserver = engram.cs.princeton.edu
cs.princeton.edu        nameserver = cs.princeton.edu
...
engram.cs.princeton.edu internet address = 128.112.136.12
cs.princeton.edu        internet address = 128.112.136.10

> server cs.princeton.edu.
> set query=A                     now we're down to host lookups
> www.cs.princeton.edu

Name:  glia.CS.Princeton.EDU
```

```
Address:  128.112.168.3
Aliases:  www.cs.princeton.edu
```

13. Both SMTP and HTTP are already largely organized as a series of requests sent by the client, and attendant server reply messages. Some attention would have to be paid in the request/reply protocol, though, to the fact that SMTP and HTTP data messages can be quite large (though not so large that we can't determine the size before beginning transmission).

    We might also need a MessageID field with each message, to identify which request/reply pairs are part of the same transaction. This would be particularly an issue for SMTP.

    It would be quite straightforward for the request/reply transport protocol to support persistent connections: once one message was exchanged with another host, the connection might persist until it was idle for some given interval of time.

    Such a request/reply protocol might also include support for variable-sized messages, without using flag characters (CRLF) or application-specific size headers or chunking into blocks. HTTP in particular currently includes the latter as an application-layer issue.

15. Existing SMTP headers that help resist forgeries include mainly the Received: header, which gives a list of the hosts through which the message has actually passed, by IP address.

    A mechanism to identify the specific user of the machine (as is provided by the identd service), would also be beneficial.

16. If an SMTP host cannot understand a command, it responds with

    500 Syntax error, command unrecognized

    This has (or is supposed to have) no other untoward consequences for the connection. A similar message is sent if a command parameter is not understood.

    This allows communicating SMTPs to query each other as to whether certain commands are understood, in a manner similar to the WILL/WONT protocol of, say, telnet.

    RFC 1869 documents a further mechanism: the client sends EHLO (Extended HELO), and an EHLO-aware server responds with a list of SMTP extensions it supports. One advantage of this is that it better supports command pipelining; it avoids multiple exchanges for polling the other side about what it supports.

17. Further information on command pipelining can be found in RFC 2197.

    (a) We could send the HELO, FROM, and TO all together, as these messages are all small and the cost of unnecessary transmission is low, but it would seem appropriate to examine the response for error indications before bothering to send the DATA.

    (b) The idea here is that a server reading with gets() in this manner would be unable to tell if two lines arrived together or separately. However, a TCP buffer flush immediately after the first line was processed could wipe out the second; one way this might occur is if the connection were handed off at that point to a child process. Another possibility is that the server busyreads after reading the first line but before sending back its response; a server that willfully refused to accept pipelining might demand that this busyread return 0 bytes. This is arguably beyond the scope of gets(), however.

(c) When the client sends its initial EHLO command (itself an extension of HELO), a pipeline-safe server is supposed to respond with 250 PIPELINING, included in its list of supported SMTP extensions.

18. MX records supply a list of hosts able to receive email; each listed host has an associated numeric "mail preference" value. This is documented further in RFC 974. Delivery to the host with the lowest-numbered mail preference value is to be attempted first.

    For HTTP, the same idea of supporting multiple equivalent servers with a single DNS name might be quite useful for load-sharing among a cluster of servers; however, one would have to ensure that the servers were in fact truly stateless. Another possibility would be for a WEB query to return a list of HTTP servers each with some associated "cost" information (perhaps related to geographical distance); a client would prefer the server with the lowest cost.

19. Implementers are free to add new subtypes to MIME, but certain default interpretations may apply. For example, unrecognized subtypes of the application type are to be treated as being equivalent to application/octet-stream. New experimental types and subtypes can be introduced; names of such types are to begin with X- to mark them as such. New image and text subtypes may be formally registered with the IANA; senders of such subtypes may also be encouraged to send the data in one of the "standard" formats as well, using multipart/alternative.

20. We quote from RFC 1521:

    > NOTE: From an implementor's perspective, it might seem more sensible to reverse this ordering, and have the plainest alternative last. However, placing the plainest alternative first is the friendliest possible option when multipart/alternative entities are viewed using a non-MIME-conformant mail reader. While this approach does impose some burden on conformant mail readers, interoperability with older mail readers was deemed to be more important in this case.

    It seems likely that anyone who has received MIME messages through text-based non-MIME-aware mail readers would agree.

21. The base64 encoding actually defines 65 transmission characters; the 65th, "=", is used as a pad character. The data file is processed in input blocks of three bytes at a time; each input block translates to an output block of four 6-bit pieces in the base64 encoding process. If the final input block of the file contains one or two bytes, then zero-bits are first added to bring the data to a 6-bit boundary (if the final block is one byte, we add four zero bits; if the final block is two bytes, we add two zero bits). The two or three resulting 6-bit pieces are then encoded in the usual way, and two or one "=" characters are appended to bring the output block to the required four pieces. In other words, if the encoded file ends with a single $=$, then the original file size was $\equiv 2 \pmod 3$; if the encoded file ends with two $=$s then the original file size was $\equiv 1 \pmod 3$.

22. When the server initiates the close, then it is the server that must enter the TIMEWAIT state. This requires the server to keep extra records; a server that averaged 100 connections per second would need to maintain about 6000 TIMEWAIT records at any one moment. HTTP 1.1

has a variable-sized message transfer mechanism; the size and endpoint of a message can be inferred from the headers. The server can thus transfer a file and wait for the client to detect the end and close the connection. Any request-reply protocol that could be adapted to support arbitrarily large messages would also suffice here.

23. For supplying an alternative error page, consult www.apache.org or the documentation for almost any other web server; apache provides a setting for ErrorDocument in httpd.conf.

    RFC 2068 (on HTTP) states:

    10.4.5 404 Not Found

    > The server has not found anything matching the Request-URI [Uniform Resource Identifier, a more general form of URL].

    However, nothing in RFC 2068 requires that the part of a URL following the host name be interpreted as a *file* name. In other words, HTTP servers are allowed to interpret "matching", as used above, in whatever manner they wish; in particular, a string representing the name of a nonexistent file may be said to "match" a designated ErrorDocument. Another example of a URL that does not represent a filename is a dynamic query.

24. One server may support multiple web sites with multiple hostnames, a technique known as *virtual hosting*. HTTP GET requests are referred by the server to the appropriate directory based on the hostname contained in the request.

25. A TCP endpoint can *abort* the connection, which entails the sending of a RST packet rather than a FIN. The endpoint then moves directly to TIMEWAIT.

    To abort a connection using the Berkeley socket library, one first sets the SO_LINGER socket option, with a linger time of 0. At this point an application close() triggers an abort as above, rather than the sending of a FIN.

26. (a) Enabling arbitrary SMTP relaying allows "spammers" to send unsolicited email via someone else's machine.

    (b) One simple solution to this problem would be the addition of a password option as part of the opening SMTP negotiation.

    (c) As of 1999, most solutions appear to be based on some form of VPN, or IP tunneling, to make ones external client IP address appear to be internal. The ssh ("secure shell", www.ssh.net) package supports a port-redirection feature to forward data from a local client port to a designated server port. Another approach is that of PPTP (Point-to-Point Tunneling Protocol), a protocol with strong Microsoft ties; see www.microsoft.com.

27. (a) A mechanism within HTTP would of course require that the client browser be aware of the mechanism. The client could ask the primary server if there were alternate servers, and then choose one of them. Or the primary server might *tell* the client what alternate to use. The parties involved might measure "closeness" in terms of RTT, in terms of measured throughput, or (less conveniently) in terms of preconfigured geographical information.

(b) Within DNS, one might add a WEB record that returned multiple server addresses. The client resolver library call (*eg* gethostbyname()) would choose the "closest", determined as above, and return the single closest entry to the client application as if it were an A record.

28. (b) Use the name of each object returned as the snmpgetnext argument in the subsequent call.

29. I tried alternating SNMP queries with telnet connections to an otherwise idle machine, and watched tcp.tcpPassiveOpens and tcp.tcpInSegs tick up appropriately. One can also watch tcp.tcpOutSegs.

30. By polling the host's SNMP server, one could find out what rsh connections had been initiated. A host that receives many such connections might be a good candidate for attack, although finding out the hosts doing the connecting would still require some guesswork.

    Someone able to use SNMP to *set* a host's routing tables or ARP tables, etc, would have many more opportunities.

31. (a) An application might want to send a burst of audio (or video) that needed to be spread out in time for appropriate playback.

    (b) An application might send video and audio data at slightly different times that needed to be synchronized, or a single video frame might be sent in multiple pieces over time.

32. This allows the server to make accurate measurements of jitter. This in turn allows an early warning of transient congestion; see the solution to Exercise 35 below. Jitter data might also allow finer control over the size of the playback buffer, although it seems unlikely that great accuracy is needed here.

33. Each receiver gets 5% of 1/1000 of 20 K/sec, or 1 byte/sec, or one RTCP packet every 84 sec. At 10K recipients, it's one packet per 840 sec, or 14 minutes.

34. (a) The answer here depends on how closely frame transmission is synchronized with frame display. Assuming playback buffers on the order of a full frame or larger, it seems likely that receiver frame-display finish times would not be synchronized with frame transmission times, and thus would not be particularly synchronized from receiver to receiver. In this case, receiver synchronization of RTCP reports with the end of frame display would not result in much overall synchronization of RTCP traffic.

    In order to achieve such synchronization, it would be necessary to have both a very uniform latency for all receivers and a rather low level of jitter, so that receivers were comfortable maintaining a negligible playback buffer. It would also be necessary, of course, to disable the RTCP randomization factor. The number of receivers, however, should not matter.

    (b) The probability that any one receiver sends in the designated 5% subinterval is 0.05, assuming uniform distribution; the probability that all 10 send in the subinterval is 0.05^{10}, which is negligible.

(c) The probability that one designated set of five receivers sends in the designated interval, and the other five do not, is $(.05)^5 \times (.95)^5$. There are (10 choose 5) = 10!/5!5! ways of selecting five designated receivers, and so the probability that *some* set of five receivers all transmit in the designated interval is (10 choose 5) $\times (.05)^5 \times (.95)^5 = 252 \times 0.0000002418 = 0.006\%$. Multiplying by 20 gives a rough estimate of about 0.12% for the probability of an upstream traffic burst rivaling the downstream burst, in any given reply interval.

35. If most receivers are reporting high loss rates, a server might consider throttling back. If only a few receivers report such losses, the server might offer referrals to lower-bandwidth/lower-resolution servers. A regional group of receivers reporting high losses might point to some local congestion; as RTP traffic is often tunneled, it might be feasible to address this by rerouting traffic.

    As for jitter measurements, we quote RFC 1889:

    > The interarrival jitter field provides a second short-term measure of network congestion. Packet loss tracks persistent congestion while the jitter measure tracks transient congestion. The jitter measure may indicate congestion before it leads to packet loss.

36. A standard application of RFC 1889 mixers is to offer a reduced-bandwidth version of the original signal. Such mixers could then be announced via SAP along with the original signal; clients could then select the version of the signal with the appropriate bandwidth.

38. For audio data we might send sample[n] for odd n in the first packet, and for even n in the second. For video, the first packet might contain sample[i,j] for i+j odd and the second for i+j even; dithering would be used to reconstruct the missing sample[i,j] if only one packet arrived.

    JPEG-type encoding (for either audio or video) could still be used on each of the odd/even sets of data; however, each set of data would separately contain the least-compressible low-frequency information. Because of this redundancy, we would expect that the total compressed size of the two odd/even sets would be significantly larger than the compressed size of the full original data.