Solutions to Selected Exercises

Chapter 1

1.1 The value is 2^7-2 (126) plus 2^{14} - 2 (16,382) plus 2^{21} - 2 (2,097,150) for a total of 2,113,658. We subtract 2 in each calculation since a network ID of all zero bits or all one bits is invalid.

1.2 Figure D.1 shows a plot of the values through August 1993.

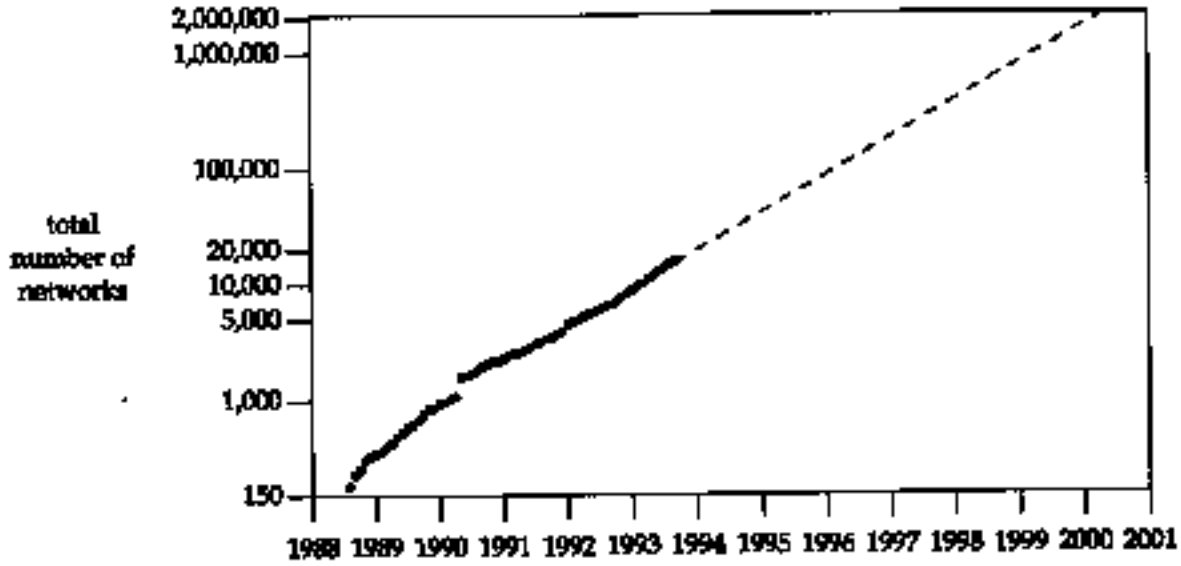

Figure D.1 Number of networks announced to NSFNET.

The dashed line estimates that the maximum number of networks will be reached in the year 2000, if the exponential growth continues.

1.3 "Be liberal in what you accept, and conservative in what you send."

Chapter 3

3.1 No, any class A address with a network ID of 127 is OK, although most systems use 127.0.0.1.

3.2 `kpno` has five interfaces: three point-to-point links and two Ethernets. RIO has four Ethernet interfaces, `gateway` has three interfaces: two point-to-point links and one Ethernet. Finally, `netb` has one Ethernet interface and two point-to-point links.

3.3 There's no difference: both have a subnet mask of 255.255.255.0, as does a class C address that is not subnetted.

3.5 It's valid and it's called a *noncontiguous subnet mask* since the 16 bits for the subnet mask are not contiguous. The RECs, however, recommend against using noncontiguous subnet masks.

3.6 It's a historical artifact. The value is 1024+512 but the MTU values printed include any required headers. Solaris 2.2 sets the MTU of the loopback interface to 8232 (8192 + 40), which allows room for 8192 bytes of user data along with the normal 20-byte IP header and 20-byte TCP header.

3.7 First, datagrams eliminate the need for connection state in the routers. Second, datagrams provide the basic building block on which unreliable (UDP) and reliable (TCP) transport layers can be built. Third, datagrams represent the minimal network layer assumption, allowing a wide range of data-link layers to be used.

Chapter 4

4.1 Issuing an `rsh` command establishes a TCP connection with the other host. Doing that causes IP datagrams to be exchanged between the two hosts. This requires the ARP cache on the other host to have an entry for our host. Therefore, even if the ARP cache was empty before we executed the `rsh` command, it's guaranteed to have an entry for our host when the `rsh` server executes the `arp` command.

4.2 Make sure that your host does not have an entry in its ARP cache for some other host on its Ethernet, say `foo`. Make sure `foo` sends a gratuitous ARP request when it bootstraps, perhaps running tcpdump on another host when `foo` bootstraps. Then shut down the host `foo` and enter an incorrect entry into the ARP cache on your system for `foo`, using the arp command and being sure to specify the temp option. Bootstrap `foo` and when it's up, look at your host's ARP cache entry for it to see whether the incorrect entry has been corrected.

4.3 Read Section 2.3.2.2 of the Host Requirements REC and Section 11.9 of this text.

4.4 Assuming that a completed ARP entry existed for the server on the client when the server was taken down, if we continually try to contact the (down) server, the ARP timeout gets extended for another 20 minutes. When the server finally reboots with a new hardware address, if it doesn't issue a gratuitous ARP, the old, invalid ARP entry will still exist on the client. We won't be able to contact the server at its new hardware address until we either manually delete the ARP cache entry or stop trying to contact it for 20 minutes.

Chapter 5

5.1 A separate frame type is not an absolute requirement, since the *op* field in Figure 4.3 has a different value for all four operations (ARP request, ARP reply, RARP request, and RARP reply). But the implementation of an RARP server, separate from the kernel's ARP server, is made easier with the different frame type field.

5.2 Each RARP server can delay for a small random time before sending a response.

As a refinement, one RARP server can be designated the primary and the others as secondaries. The primary server can respond without a delay, and the secondaries with random delays.

As yet another refinement, with a primary and secondaries, the secondaries can be programmed to respond only to a duplicate request received in a short time frame. This assumes that the reason for the duplicate request is that the primary is down.

Chapter 6

6.1 If there were one hundred hosts on the local cable, each could try to send an ICMP port unreachable at about the same time. Many of these transmissions could lead to collisions (if an Ethernet is being used), which can render the network useless for a second or two.

6.2 It is a "should."

6.3 An ICMP error is always sent with a TOS of 0, as we indicated in Figure 3.2. An ICMP query request can be sent with any TOS, and the corresponding reply should be sent with the same TOS.

6.4 `netstat -s` is the common way to see the per-protocol statistics. On a SunOS 4.1.1 host (`gemini`) that has received 48 million IP datagrams, the ICMP statistics are:

```
Output histogram:
        echo reply: 1757
        destination unreachable: 700
        time stamp reply: 1
Input histogram:
        echo reply: 211
        destination unreachable: 3071
        source quench: 249
        routing redirect: 2789
```

```
echo: 1757
#10: 21
time exceeded: 56
time stamp: 1
```

The 21 input messages of type 10 are router solicitations that SunOS 4.1.1 doesn't support.

SNMP can also be used (Figure 25.26) and some systems, such as Solaris 2.2, generate `netstat -s` output that uses SNMP variable names.

Chapter 7

7.2 86 bytes divided by 960 bytes/sec, times 2 gives 179.2 ms. When ping is run at this speed, the printed values are 180 ms.

7.3 (86 + 48) bytes divided by 960 bytes/sec, times 2 gives 279.2 ms. The additional 48 bytes are because the final 48 bytes of the 56 bytes in the data portion must be escaped: `0xc0` is the SLIP END character.

7.4 CSLIP only compresses the TCP and IP headers for TCP segments. It has no effect on the ICMP messages used by `ping`.

7.5 On a SPARCstation ELC a ping of the loopback address yields an RTT of 1.310 ms, while a ping of the host's Ethernet address yields an RTT of 1.460 ms. This difference is the additional processing done by the Ethernet driver, to determine that the datagram is really destined for the local host. You need a version of ping that outputs microsecond resolution to measure this.

Chapter 8

8.1 If an incoming datagram has a TTL of 0, doing the decrement and then test would set the TTL to 255 and let the datagram continue. Although a router should never receive a datagram with a TTL of 0, it has occurred.

8.2 We noted that `traceroute` stores 12 bytes of data in the data portion of the UDP datagram, part of which is the time the datagram was sent. From Figure 6.9, however, we see that ICMP only returns the first 8 bytes of the IP datagram that was in error, and we noted there that this is the 8-byte UDP header. Therefore the time value stored by `traceroute` is not returned in the ICMP error message. `traceroute` saves the time when it sends a packet, and when an ICMP reply is received, fetches the current time and

subtracts the two value to get the RTT.

Recall from Chapter 7 that `ping` stored the time in the outgoing ICMP echo request and this data was echoed by the server. This allowed `ping` to print the correct RTT, even if the packets were returned out of order.

8.3 The first line of output is correct and identifies R1. The next probe starts with a TTL of 2, and this is decremented by R1. When R2 receives this it decrements the TTL from 1 to 0 but incorrectly forwards it to R3. R3 sees that the incoming TTL is 0 and sends back the time exceeded. This means the second line of output (for the TTL of 2) identifies R3, not R2. The third line of output correctly identifies R3. The clue that this bug is present is two consecutive lines of output that identify the same router.

8.4 In this case the TTL of 1 identifies R1, the TTL of 2 identifies R2, and the TTL of 3 identifies R3; but when the TTL is 4 the UDP datagram gets to the destination with an incoming TTL of 1. The ICMP port unreachable is generated, but its TTL is 1 (incorrectly copied from the incoming TTL). This ICMP message goes to R3 where the TTL is decremented and the message discarded. An ICMP time exceeded is *not* generated, since the datagram that was discarded was an ICMP error message (port unreachable). A similar scenario occurs for the probe with a TTL of 5, but this time the outgoing port unreachable starts with a TTL of 2 (the incoming TTL) and makes it back to R2, where it's discarded. The port unreachable corresponding to the probe with a TTL of 6 makes it back to R1, where it's discarded. Finally the port unreachable for the probe with a TTL of 7 makes it all the way back, where it arrives with an incoming TTL of 1. (`traceroute` considers an arriving ICMP message with a TTL of 0 or 1 to be suspicious, so it prints an exclamation point after the RTT.) In summary, the lines for a TTL of 1, 2, and 3 correctly identify R1, R2, and R3, followed by three lines each containing three timeouts, followed by the line for a TTL of 7 that identifies the destination.

8.5 It appears that all these routers initialize the outgoing TTL of an ICMP message to 255. This is common. The incoming value of 255 from `netb` is what we expect, but the value of 253 from `butch` means there is probably a missing router between it and `netb`. Otherwise we would expect an incoming TTL of 254 at this point. Similarly, from `enss142.UT.westnet.net` we expect a value of 252, not 249. It appears these missing routers are not handling the outbound UDP datagram correctly, but they are decrementing the TTL on the returned ICMP message correctly.

We must be careful when looking at the incoming TTL, since sometimes a value other than what we expect can be caused by the return ICMP message taking a different path from the outbound UDP datagram. In this example, however, it confirms what we suspect-there are missing routers that `traceroute` is not finding when the loose source routing option is used.

8.7 The `ping` client sets the identifier field in the ICMP echo request message (Figure 7.1) to its process ID. The ICMP echo reply contains this identifier field. Each client looks at this returned identifier field and handles only those that it sent.

The `traceroute` client sets its UDP source port number to the logical-OR of its process ID and 32768. Since the returned ICMP message always contains the first 8 bytes of the IP datagram that generated the error (Figure 6.9), which includes the entire UDP header, this source port number is returned in the ICMP error.

8.8 The `ping` client sets the optional data portion of the ICMP echo request message to the time at which the packet is sent. This optional data must be returned in the ICMP echo reply. This allows `ping` to calculate the accurate round-trip time, even if packets are returned out of order.

The `traceroute` client can't operate this way because all that's returned in the ICMP error is the UDP header (Figure 6.9), none of the UDP data. Therefore `traceroute` must remember when it sends a request, wait for the reply, and calculate the time difference.

This illustrates another difference between Ping and Traceroute: Ping sends one packet a second, regardless of whether it receives any replies, while Traceroute sends a request and then waits for either a reply or a timeout before sending the next request.

8.9 Since Solaris 2.2 starts ephemeral UDP port numbers at 32768 by default, there is a much greater chance that the destination port is in use on the destination host.

Chapter 9

9.1 When the ICMP standard was first specified, RFC 792 [Postel 1981b], subnetting was not in use. Also, using a single network redirect instead of *N* host redirects (for all *N* hosts on the destination network) saves some space in the routing table.

9.2 The entry is not required, but if it is removed, all IP datagrams to `slip` are sent to the default router (`sun`), which then forwards them to the router `bsdi`. Since `sun` is forwarding a datagram out the same interface on which it was received, it sends an ICMP redirect to `svr4`. This creates the same routing table entry on svr4 that we removed, although this time it is created by a redirect instead of being added at bootstrap time.

9.3 When the 4.2BSD host receives the datagram destined for 140.1.255.255 it finds that it has a route to the network (140.1) so it tries to forward the datagram. To do this it sends an ARP broadcast looking for 140.1.255.255. No reply is received for this ARP request, so the datagram is eventually discarded. If there are many of these 4.2BSD hosts on the

cable, every one sends out this ARP broadcast at about the same time, swamping the network temporarily.

9.4 This time a reply is received for each ARP request, telling each 4.2BSD host to send the datagram to the specified hardware address (the Ethernet broadcast). If there are *k* of these 4.2BSD hosts on the cable, all receive their own ARP reply, causing each one to generate another broadcast. Each host receives each broadcast IP datagram destined to 140.1.255.255, and since every host now has an ARP cache entry, the datagram is forwarded again to the broadcast address. This continues and generates an *Ethernet meltdown.* [Manber 1990] describes other forms of chain reactions in networks.

Chapter 10

10.1 Thirteen of the routes came from `kpno`: all except 140.252.101.0 and 140.252.104.0, the other networks to which `gateway` is directly connected.

10.2 Sixty seconds will pass before the 25 routes advertised in the lost datagram are updated. This isn't a problem because RIP normally requires 3 minutes without an update before it declares a route dead.

10.3 RIP runs on top of UDP, and UDP provides an optional checksum for the data portion of the UDP datagram (Section 11.3). OSPF, however, runs on top of IP. The IP checksum covers only the IP header, so OSPF must add its own checksum field.

10.4 Load balancing increases the chances of packets being delivered out of order, and possibly distorts the round-trip times calculated by the transport layer.

10.5 This is called simple split horizon.

10.6 In Figure 12.1 we show that each of the 100 hosts processes the broadcast UDP datagram through the device driver, IP layer, and UDP layer, where it'll finally be discarded when it's discovered that UDP port 520 is not in use.

Chapter 11

11.1 Since there are 8 additional bytes of header when IEEE 802 encapsulation is used, 1465 bytes of user data is the smallest size that causes fragmentation.

11.3 There are 8200 bytes of data for IP to send, the 8192 bytes of user data and the 8-byte UDP header. Using the tcpdump notation, the first fragment is 1480@0+ (1480 bytes of data, offset of 0, with the "more fragments" bit set). The second is 1480@1480+, the third is 1480@2960+, the fourth is 1480@4440+, the fifth is 1480@5920+, and the

sixth is 800@7400. 1480 x 5+ 800 = 8200, which is the number of bytes to send.

11.4 Each 1480-byte fragment is divided into three pieces: two 528-byte fragments and one 424-byte fragment. The largest multiple of 8 less than 532 (552 - 20) is 528. The 800-byte fragment is divided into two pieces: a 528-byte fragment and a 272-byte fragment. Thus, the original 8192-byte datagram becomes 17 frames across the SLIP link.

11.5 No. The problem is that when the application times out and retransmits, the IP datagram generated by the retransmission has a new identification field. Reassembly is done only for fragments with the same identification field.

11.6 The identification field in the IP header (47942) is the same.

11.7 First, from Figure 11.4 we see that `gemini` does not have outgoing UDP checksums enabled. It's highly probable that the operating system on this host (SunOS 4.1.1) is one that never verifies an incoming UDP checksum unless outgoing UDP checksums are enabled. Second, it could be that most of the UDP traffic is local traffic, instead of WAN traffic, and therefore not subject to all the vagaries of WANs.

11.8 The loose and strict source routing options are copied into each fragment. The timestamp option and the record route option are not copied into each fragment-they appear only in the first fragment.

11.9 No. We saw in Section 11.12 that many implementations can filter incoming datagrams destined for a given UDP port number based on the destination IP address, source IP address, and source port number.

Chapter 12

12.1 Broadcasting by itself does not add to network traffic, but it adds extra host processing. Broadcasting can lead to additional network traffic if the receiving hosts incorrectly respond with errors such as ICMP port unreachables. Also, routers normally don't forward broadcast packets, whereas bridges normally do, so broadcasts on a bridged network can travel much farther than they would on a routed network.

12.2 Every host receives a copy of every broadcast. The interface layer receives the frame, and passes it to the device driver. If the type field is for some other protocol, it is the device driver that discards the frame.

12.3 First execute `netstat -r` to see the routing table. This shows the names of all the interfaces. Then execute `ifconfig` (Section 3.8) for each interface: the flags tell if the interface supports broadcasting, and if so the broadcast address is also output.

12.4 Berkeley-derived implementations do not allow a broadcast datagram to be fragmented. When we specified the length of 1472 bytes, the resulting IP datagram was exactly 1500 bytes, the Ethernet MTU. Refusing to allow a broadcast datagram to be fragmented is a policy decision-there is no technical reason (other than a desire to reduce the number of broadcast packets).

12.5 Depending on the multicasting support in the various Ethernet interface cards in the 100 hosts, the multicast datagram can be ignored by the interface card, or discarded by the device driver.

Chapter 13

13.1 Use some host-unique value when generating the random value. The IP address and link-layer address are two values that should differ on every host. The time-of-day is a bad choice, especially if all the hosts run a protocol such as NTP to synchronize their clocks.

13.2 They added an application protocol header that included a sequence number and a timestamp.

Chapter 14

14.1 A resolver is always a client, but a name server is both a client and server.

14.2 The question is returned, which accounts for the first 44 bytes. The single answer occupies the remaining 31 bytes: a 2-byte pointer for the domain name (i.e., a pointer to the domain name in the question), 10 bytes for the fixed-size fields (type, class, TTL, and resource length), and 19 bytes for the resource data (a domain name). Notice that the domain name in the resource data (`svr4.tuc.noao.edu.`) doesn't share a suffix with the domain name in the question (`34.13.252.140.in-addr.arpa.`) so a pointer can't be used.

14.3 Reversing the order means using the DNS first, and if that fails, trying to convert the argument as a dotted-decimal number. This means every time a dotted-decimal number is specified, the DNS is used, involving a name server. This is a waste of resources.

14.4 Section 4.2.2 of RFC 1035 specifies that a 2-byte length precedes the actual DNS message.

14.5 When a name server starts it normally reads the (possibly out of date) list of root servers from a disk file. It then tries to contact one of these root servers, requesting the name server records (a query type of NS) for the root domain. This returns the current up-to-date list of root servers. Minimally this requires one of the root server entries in the

start-up disk file to be current.

14.6 The registration services of the InterNIC updates the root servers three times a week.

14.7 Since the resolver comes and goes, as applications come and go, if the system is configured to use multiple name servers and the resolver maintains no state, the resolver cannot keep track of the round-trip times to its various name servers. This can lead to timeouts for resolver queries that are too short, causing unnecessary retransmissions.

14.8 Sorting the A records should be done by the resolver, not the name server, since the resolver normally knows more than the server about the network topology of the client. (Newer releases of BIND provide for resolver sorting of A records.)

Chapter 15

15.1 TFTP requests sent to the broadcast address should be ignored. As stated in the Host Requirements RFC, responding to a broadcast request can create a significant security hole. A problem, however, is that not all implementations and APIs provide the destination address of a UDP datagram to the process that receives the datagram (Section 11.12). For this reason many TFTP servers don't enforce this restriction.

15.2 Unfortunately, the RFC says nothing about this block number wrap. Implementations should be able to transfer files up through 33,553,920 bytes (65535 x 512). Many implementations fail when the size of the file exceeds 16,776,704 (32767 x 512) since they incorrectly maintain the block number as a signed 16-bit integer instead of an unsigned integer.

15.3 This simplifies coding a TFTP client to fit in read-only memory, because the server is the sender of the bootstrap files, so the server must implement the timeout and retransmission.

15.4 With its stop-and-wait protocol, TFTP can transfer a maximum of 512 bytes per client-server round trip. The maximum throughput of TFTP is then 512 bytes divided by the round-trip time between the client and server. On an Ethernet, assuming a round-trip time of 3 ms, the maximum throughput is around 170,000 bytes/sec.

Chapter 16

16.1 A router could forward an RARP request to some other host on one of the router's other attached networks, but sending a reply then becomes a problem. The router would also have to forward RARP replies.

BOOTP doesn't have this reply problem since the address to reply to is a normal IP address that the routers know how to forward anyway. The problem is that RARP uses only link-layer addresses, and routers don't normally know these values for hosts on other, nonattached, networks.

16.2 It could use its own hardware address, which should be unique, and which is set in the request and returned in the reply.

Chapter 17

17.1 All are mandatory except the UDP checksum. The IP checksum covers only the IP header, while the others start immediately after the IP header.

17.2 The source IP address, source port number, or protocol field might have been corrupted.

17.3 Many Internet applications use a carriage return and linefeed to mark the end of each application record. This is NVT ASCII coding (Section 26.4). An alternative technique is to prefix each record with a byte count, which is used by the DNS (Exercise 14.4) and Sun RPC (Section 29.2).

17.4 As we saw in Section 6.5, an ICMP error must return at least the first 8 bytes beyond the IP header of the IP datagram that caused the error. When TCP receives an ICMP error it needs to examine the two port numbers to determine which connection the error corresponds to, so the port numbers must be in the first 8 bytes of the TCP header.

17.5 There are options at the end of the TCP header, but there are no options in the UDP header.

Chapter 18

18.1 The ISN is a 32-bit counter that wraps around from 4,294,912,000 to 8,704 approximately 9.5 hours after the system was bootstrapped. After approximately another 9.5 hours it will wrap around to 17,408, then 26,112 after another 9.5 hours, and so on. Since the ISN starts at 1 when the system is bootstrapped, and since the lowest order digit cycles through 4, 8, 2, 6, and 0, the ISN should always be an odd number.

18.2 In the first case we used our sock program, and by default it transmits the Unix newline character as itself-the single ASCII character 012 (octal). In the second case we used the Telnet client and it converts the Unix newline into two ASCII characters-a carriage return (octal 015) followed by a linefeed (octal 012).

18.3 On a half-closed connection one end has sent a FIN and is waiting for either data or a FIN from the other end. A half-open connection is when one end crashes, unbeknown to the other end.

18.4 The 2MSL wait state is only entered for a connection that has gone through the ESTABLISHED state.

18.5 First, the daytime server does the active close of the TCP connection after writing the time and date to the client. This is indicated by the message printed by our sock program: "connection closed by peer." The client's end of the connection goes through the passive close states. This puts the socket pair in the TIME_WAIT state on the server, not the client.

Next, as shown in Section 18.6, most Berkeley-derived implementations allow a new connection request to arrive for a socket pair currently in the TIME_WAIT state, which is exactly what's happening here.

18.6 A reset is sent in response to the FIN, because the FIN arrived for a connection that was CLOSED.

18.7 The party that dials the number does the active open. The party whose telephone rings does the passive open. Simultaneous opens are not permitted, but a simultaneous close is OK.

18.8 We would only see ARP requests, not TCP SYN segments, but the ARP requests would have the same timing as in the figure.

18.9 The client is on the host `solaris` and the server is on the host `bsdi`. The client's ACK of the server's SYN is combined with the first data segment from the client (line 3). This is perfectly legal under the rules of TCP, although most implementations don't do this. Next, the client sends its FIN (line 4) before waiting for the ACK of its data. This allows the server to acknowledge both the data and the FIN in line 5.

This exchange (sending one segment of data from the client to the server) requires seven segments. The normal connection establishment and termination (Figure 18.13), along with a single data segment and its acknowledgment, requires nine segments.

18.10 First, the server's ACK of the client's FIN is normally not delayed (we discuss delayed ACKs in Section 19.3) but sent as soon as the FIN arrives. It takes the application a while to receive the EOF and tell its TCP to close its end of the connection. Second, the server that receives the FIN does not have to close its end of the connection on receiving the FIN from the client. As we saw in Section 18.5, data can still be sent.

18.11 If an arriving segment that generates an RST has an ACK field, the sequence number of the RST is the arriving ACK field. The ACK value of 1 in line 6 is relative to the ISN of 26368001 in line 2.

18.12 See [Crowcroft et al. 1992] for comments on layering.

18.13 Five queries are issued. Assume there are three packets to establish the connection, one for the query, one to ACK the query, one for the response, one to ACK the response, and four to terminate the connection. This means II packets per query, for a total of 55 packets. Using UDP reduces this to 10 packets.

This can be reduced to 10 packets per query if the ACK of the query is combined with the response (Section 19.3).

18.14 The limit is about 268 connections per second: the maximum number of TCP port numbers (65536 -1024 = 64512, ignoring the well-known ports) divided by the TIME_WAIT state of 2MSL.

18.15 The duplicate FIN is acknowledged and the 2MSL timer is restarted.

18.16 The receipt of an RST while in the TIME_WAIT state causes the state to be prematurely terminated. This is called *TIME_WAIT assassination.* RFC 1337 [Braden 1992a] discusses this in detail and shows the potential problems. The simple fix proposed by this RFC is to ignore RST segments while in the TIME_WAIT state.

18.17 It's when the implementation does not support a half-close. Once the application causes a FIN to be sent, the application can no longer read from the connection.

18.18 No. Incoming data segments are demultiplexed using the source IP address, source port number, destination IP address, and destination port number. For incoming connection requests we saw in Section 18.11 that a TCP server can normally prevent connections from being accepted based on the destination IP address.

Chapter 19

19.1 Two application writes, followed by a read, cause a delay because the Nagle algorithm will probably be invoked. The first segment (with 8 bytes of data) is sent and its ACK is waited for before sending the 12 bytes of data. If the server implements delayed ACKs, there can be a delay of up to 200 ms (plus the RTT) before this ACK is received.

19.2 Assuming 5-byte CSLIP headers (IP and TCP) and 2 bytes of data, the RTT across the SLIP link for these segments is about 14.5 ms. We have to add to this the RTT across

the Ethernet (normally 5-10 ms), plus the routing time on sun and bsdi. So yes, the observed times do appear correct.

19.3 In Figure 19.6 the time difference between segments 6 and 9 is 533 ms. In Figure 19.8 the time difference between segments 8 and 12 is 272 ms. (We measured the time for the F2 key, not the Fl key, since the first echo of the Fl key was lost in the second figure.)

Chapter 20

20.1 Byte number 0 is the SYN and byte number 8193 is the FIN. The SYN and FIN each occupy 1 byte in the sequence number space.

20.2 The first application write causes the first segment to be sent with the PUSH flag. Since BSD/386 always uses slow start, it waits for the first ACK before sending any more data. During this time the next three application writes occur, and the sending TCP buffers the data to send. The next three segments do not contain the PUSH flag since there is more data in the buffer to send. Eventually slow start catches up with the application writes and every application write causes a segment to be sent, and since that segment is the last one in the buffer, the PUSH flag is set.

20.3 Solving the bandwidth-delay equation for the capacity, it is 1,920 bytes for the first case, and 2,062 for the satellite case. It appears that the receiving TCP is only advertising a window of 2,048 bytes.

A window greater than 16,000 bytes should be able to saturate the satellite link.

20.4 No, because TCP can repacketize data after a timeout, as we'll see in Section 21.11.

20.5 Segment 15 is a window update sent automatically by the TCP module as a result of the application reading data, which causes the window to open. This is similar to segment 9 in that figure. Segment 16, however, is a result of the application closing its end of the connection.

20.6 This can cause the sender to inject packets into the network at a rate faster than the network can really handle. This is called *ACK compression* or *ACK smashing* [Mogul 1993, Sec. 15.8.13]. This reference indicates that ACK compression occurs on the Internet, although it rarely leads to congestion.

Chapter 21

21.1 The next timeout is for 48 seconds: 0+4x12. The factor of 4 is the next multiplier in

the exponential backoff.

21.2 It appears SVR4 still uses the factor *ID* instead of 4D in the calculation of *RTO*.

21.3 The stop-and-wait protocol used by TFTP is limited to 512 bytes of data per round trip. 32768/512 x 1.5 is 96 seconds.

21.4 Show four segments, numbered 1,2,3, and 4. Assume the order of receipt is 1, 3, 2, and 4. The ACKs generated by the receiver will be ACK 1 (a normal ACK), ACK 1 (a duplicate ACK when segment 3 is received out of order), ACK 3 when segment 2 is received (acknowledging both segments 2 and 3), and then ACK 4. Here one duplicate ACK is generated. If the order of receipt were 1, 3, 4, 2, two duplicate ACKs would be generated.

21.5 No, because the slope is still up and to the right, not downward.

21.6 See Figure E.1.

21.7 In Figure 21.2 the segments contain 256 bytes of data, which takes approximately 250 ms to transfer across the 9600 bits/sec CSLIP link between slip and bsdi. Assuming the data segments are not queued somewhere between bsdi and vangogh, they arrive at vangogh about 250 ms apart. Since this exceeds the 200-ms delayed ACK timer, each segment is acknowledged when the next delayed ACK timer expires.

Chapter 22

22.1 The ACKs are probably all delayed on the host bsdi, because there is no reason to send them immediately. That's why the relative times have 0.170 and 0.370 as the fractional part. It also appears that the 200-ms timer on bsdi is running about 18 ms behind the same timer on sun.

22.2 The FIN flag, just like the SYN flag, occupies 1 byte in the sequence number space. "The advertised window appears to be 1 byte smaller because TCP allows room for the 1 byte of sequence number space occupied by the FIN flag.

Chapter 23

23.1 It is usually simpler to invoke the keepalive option than explicitly coding application probes; the keepalive probes take less network bandwidth than application probes (since keepalive probes and answers contain no data); no probes are sent unless the connection is idle.

23.2 The keepalive option can cause a perfectly good connection to be dropped because of a temporary network outage; the probe interval (2 hours) is normally not configurable on an application basis;

Chapter 24

24.1 It means the sending TCP supports the window scale option, but doesn't need to scale its window for this connection. The other end (that receives this SYN) can then specify a window scale factor (that can be 0 or nonzero).

24.2 64000: the receive buffer size (128000) right shifted 1 bit. 55000: the receive buffer size (220000) right shifted 2 bits.

24.3 No. The problem is that acknowledgments are not reliably delivered (unless they're piggybacked with data) so a scale change appearing on an ACK could get lost.

24.4 2^{32}x8/120 equals 286 Mbits/sec, 2.86 times the FDDI data rate.

24.5 Each TCP would have to remember the last timestamp received on any connection from each host. Read Appendix B.2 of RFC 1323 for additional details.

24.6 The application must set the size of the receive buffer *before* establishing the connection with the other end, since the window scale option is sent in the initial SYN segment.

24.7 If the receiver ACKs every second data segment, the *throughput* is 1,118,881 bytes/sec. Using a window of 62 segments, with an ACK for every 31 segments, the value is 1,158,675.

24.8 With this option the timestamp echoed in the ACK is always from the segment that caused the ACK. There is no ambiguity about which retransmitted segment the ACK is for, but the other part of Karn's algorithm, dealing with the exponential backoff on retransmission, is still required.

24.9 The receiving TCP queues the data, but it cannot be passed to the application until the three-way handshake is complete: when the receiving TCP moves into the ESTABLISHED state.

24.10 Five segments are exchanged:

1. Client to server: SYN, data (request), and FIN. The server must queue the data as described in the previous exercise.
2. Server to client: SYN and ACK of client's SYN.

3. Client to server: ACK of server's SYN and client FIN (again). This causes the server to move to the ESTABLISHED state, and the queued data from segment 1 is passed to the server application.
4. Server to client: ACK of client FIN (which also acknowledges client data), data (server's reply), and server's FIN. This assumes that the SPT is short enough to allow this delayed ACK. When the client TCP receives this segment, the reply is passed to the client application, but the total time has been twice the RTT plus the SPT.
5. Client to server: ACK of server's FIN.

24.11 16,128 transactions per second (64,512 divided by 4).

24.12 The transaction time using T/TCP cannot be faster than the time required to exchange a UDP datagram between the two hosts. T/TCP should always take longer, since it still involves state processing that UDP doesn't do.

Chapter 25

25.1 If a system is running both a manager and agent, they are probably different processes. The manager listens on UDP port 162 for traps, and the agent listens on UDP port 161 for requests. If the same port were used for both traps and requests, separating the manager from the agent would be hard.

25.2 Refer to the section "Table Access" in Section 25.7.

Chapter 26

26.1 We expect segments 2, 4, and 9 from the server to be delayed. The time difference between segments 2 and 4 is 190.7 ms and the time difference between segments 2 and 9 is 400.7 ms.

All the ACKs from the client to the server appear to be delayed: segments 6, II, 13, 15, 17, and 19. The time differences of the last five from segment 6 are 400.0, 600.0, 800.0, 1000.0, and 2.600 ms.

26.2 If one end of a connection is in TCP's urgent mode, then every time a segment is received, one is sent. This segment does not tell the receiver anything new (it is not acknowledging new data, for example), and it contains no data, it just reiterates that urgent mode has been entered.

26.3 There are only 512 of these reserved ports (512-1023), limiting a host to 512 Rlogin clients. The limit is normally less than 512 in real life, since some of the port numbers in

this range are used as well-known ports by various servers, such as the Rlogin server.

TCP's limitation is that the socket pair defining a connection (the 4-tuple) must be unique. Since the Rlogin server always uses the same well-known port (513) multiple Rlogin clients on a given host can use the same reserved port only if they're connected to different server hosts. Rlogin clients, however, don't use this technique of trying to reuse reserved ports. If this technique were used, the theoretical limit is a maximum of 512 Rlogin clients at any one time that are all connected to the same server host.

Chapter 27

27.1 Theoretically the connection cannot be established while the socket pair is in the 2MSL wait on either end. Realistically, however, we saw in Section 18.6 that most Berkeley-derived implementations do accept a new SYN for a connection in the TIME_WAIT state.

27.2 These lines are not part of a server reply that begins with a 3-digit reply code, so they cannot be from the server.

Chapter 28

28.1 A *domain literal is* a dotted-decimal IP address within square brackets. For example: `mail rstevens@[140.252.1.54]`.

28.2 Six round trips: the HELO command, MAIL, RCPT, DATA, body of the message, and QUIT.

28.3 This is legal and is called *pipelining* [Rose 1993, Sec. 4.4.4]. Unfortunately there exist brain-damaged SMTP receiver implementations that clear their input buffer after each command is processed, causing this technique to fail. If this technique is used, naturally the client cannot discard the message until all the replies have been checked to verify that the message was accepted by the server.

28.4 Consider the first five network round trips from Exercise 28.2. Each is a small command (probably a single segment) that places little load on the network. If all five make it through to the server without retransmission, the congestion window could be six segments when the body is sent. If the body is large, the client could send the first six segments at once, which the network might not be able to handle.

28.5 Newer releases of BIND shuffle the MX records with the same value, as a form of load balancing.

Chapter 29

29.1 No, because tcpdump cannot distinguish an RPC request or reply from any other UDP datagram. The only time it interprets the contents of UDP datagrams as NFS packets is when the source or destination port number is 2049. Random RPC requests and replies can use an ephemeral port number on each end.

29.2 From Section 1.9 recall that a process must have superuser privileges to assign itself a port number less than 1024 (a well-known port). While this is OK for system-provided servers, such as the Telnet server, the FTP server, and the Port Mapper, we wouldn't want this restriction for all RPC servers.

29.3 Two concepts in this example are that the client ignores any server reply that doesn't have the XID that the client is waiting for, and UDP queues received datagrams (up to some limit) until the application reads the datagram. Also, the XID does not change on a timeout and retransmission, it changes only when another server procedure is called.

The events performed by the client stub are as follows: time 0: send request 1; time 4: time out and retransmit request 1; time 5: receive server's reply 1, return reply to application; time 5: send request 2; time 9: time out and retransmit request 2; time 10: receive server's reply 1, but ignore it since we're waiting for reply 2; time II: receive server's reply 2, return reply to application.

The events at the server are as follows: time 0: receive request 1, start operation; time 5: send reply 1; time 5: receive request 1 (from client's retransmission at time 4), start operation; time 10: send reply 1; time 10: receive request 2 (from client's transmission at tune 5), start operation; time II: send reply 2; time II: receive request 2 (from client's retransmission at time 9), start operation; time 12: send reply 2. This final server reply is just queued by the client's UDP for the next receive done by the client. When the client reads it, the XID will be wrong, and the client will ignore it.

29.4 Changing the server's Ethernet card changes its physical address. Even though we noted in Section 4.7 that SVR4 does not send a gratuitous ARP on bootstrap, it still must send an ARP request for the physical address of sun before it can reply to its NFS requests. Since sun already has an ARP entry for svr4, it updates this entry with the sender's (new) hardware address from the ARP request.

29.5 The second of the client's block I/O daemons (reading at offset 73728) is out of sync with the first by about 0.74 seconds. That is, this second daemon times out 0.74 seconds after the first in lines 131-145. It appears the server never saw the request in line 167, but did see the request in line 168. The second block I/O daemon won't retransmit until 0.74 seconds after line 168, and in the mean time the first block I/O daemon continues issuing requests.

29.6 If TCP is used, and the TCP segment containing the server's reply is lost in the network, the server's TCP will time out and retransmit the reply when it doesn't receive an ACK from the client's TCP. Eventually the segment will arrive at the client's TCP The difference here is that the two TCP modules do the timeout and retransmission, not the NFS client and server. (When UDP is used, the NFS client code performs the timeout and retransmission.) Therefore the NFS client never knows that the reply was lost and had to be retransmitted.

29.7 It is possible for the NFS server to obtain a different port number after the reboot. This would complicate the client, because it would have to know that the server crashed and contact the server's port mapper after the reboot to find the NFS server's new port number.

This scenario, where the server crashes and reboots and a server RPC application obtains a new ephemeral port, can happen to any RPC application that doesn't use a well-known port.

29.8 No. The NFS client can reuse the same local, reserved port number for different servers. TCP requires the 4-tuple of local IP address, local port, foreign IP address, and foreign port to be unique, and the foreign IP address is different for each server host.

Chapter 30

30.1 Type `whois "net 88"`. Class A network IDs 64 through 95 are reserved.

30.2 Type `whois whitehouse-dom`. Either the `host` command or `nslookup` can query the DNS.

30.3 No, `xscope` can run on a different host from the server. If the hosts are different, then `xscope` can also use TCP port 6000 for its incoming connection.

Configurable Options

We've seen many features of TCP/IP that we've had to describe with the qualifier "it depends on the configuration." Typical examples are whether or not UDP checksums are enabled (Section 11.3), whether destination IP addresses with the same network ID but a different subnet ID are local or nonlocal (Section 18.4), and whether directed broadcasts are forwarded or not (Section 12.3). Indeed, many operating characteristics of a given TCP/IP implementation can be modified by the system administrator.

This appendix lists some of the configurable options for the various TCP/IP implementations that have been used throughout the text. As you might expect, every vendor does things differently from all others. Nevertheless, this appendix gives an idea of the types of parameters different implementations allow one to modify. A few options that are highly implementation specific, such as the low-water mark for the memory buffer pool, are not described.

These variables are described for informational purposes only. Their names, default values, or interpretation can change from one release to the next. Always check your vendor's documentation (or bug them for adequate documentation) for the final word on these variables.

This appendix does not cover the initialization that takes place every time the system is bootstrapped: the initialization of each network interface using `ifconfig` (setting the IP address, the subnet mask, etc.), entering static routes into the routing table, and the like. Instead, this appendix focuses on the configuration options that affect how TCP/IP operates.

E.1 BSD/386 Version 1.0

This system is an example of the "classical" BSD configuration that has been used since 4.2BSD. Since the source code is distributed with the system, configuration options are specified by the administrator, and the kernel is recompiled. There are two types of options: constants that are defined in the kernel configuration file (see the `config`(8) manual page), and variable initializations in various C source files. Brave and knowledgeable administrators can also change the values of these C variables in either the running kernel or the kernel's disk image, using a debugger, to avoid rebuilding the kernel. Here are the constants that can be changed in the kernel's configuration file.

`IPFORWARDING`
The value of this constant initializes the kernel variable `ipforwarding`. If 0 (default), IP datagrams are not forwarded. If 1, forwarding is always enabled.

GATEWAY
If defined, causes IPFORWARDING to be set to 1. Additionally, defining this constant causes certain system tables (the ARP cache and the routing table) to be larger.

SUBNETSARELOCAL
The value of this constant initializes the kernel variable subnetsarelocal. If 1 (default), a destination IP address with the same network ID as the sending host but a different subnet ID is considered local. If 0, only destination IP addresses on an attached subnet are considered local. This is summarized in Figure E.1.

Network IDs	Subnet IDs	subnetsarelocal		Comment
		1	0	
same	same	local	local	always local
same	different	local	nonlocal	depends on configuration
different	-	nonlocal	nonlocal	always nonlocal

Figure E.1 Interpretation of subnetsarelocal kernel variable.

This affects the MSS selected by TCP. When sending to local destinations, TCP chooses the MSS based on the MTU of the outgoing interface. When sending to nonlocal destinations, TCP uses the variable tcp_mssdflt as the MSS.

IPSENDREDIRECTS
The value of this constant initializes the kernel variable ipsendredirects. If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

DIRECTED_BROADCAST
If 1 (default), received datagrams whose destination address is the directed broadcast address of an attached interface are forwarded as a link-layer broadcast. If 0, these datagrams are silently discarded.

The following variables can also be modified. These variables are spread throughout different files in the /usr/src/sys/netinet directory.

tcprexmtthresh	The number of consecutive ACKs that triggers the fast retransmit and fast recovery algorithm. The default value is 3.
tcp_ttl	The default value for the TTL field for TCP segments. Default value is 60.
tcp_mssdflt	The default TCP MSS for nonlocal destinations. Default value is 512.

`tcp_keepidle`	Number of 500-ms clock ticks before sending a keepalive probe. Default value is 14400 (2 hours).
`tcp_keepintvl`	Number of 500-ms clock ticks between successive keepalive probes, when no response is received. Default value is 150 (75 seconds).
`tcp_sendspace`	The default size of the TCP send buffer. Default value is 4096.
`tcp_recvspace`	The default size of the TCP receive buffer. This affects the window size that is offered. Default value is 4096.
`udpcksum`	If nonzero, UDP checksums are calculated for outgoing UDP datagrams, and incoming UDP datagrams containing nonzero checksums have their checksum verified. If 0, outgoing UDP datagrams do not contain a checksum, and no checksum verification is performed on incoming UDP datagrams, even if the sender calculated a checksum. Default is 1.
`udp_ttl`	The default value for the TTL field in UDP datagrams. Default value is 30.
`udp_sendspace`	The default size of the UDP send buffer. Defines the maximum UDP datagram that can be sent. Default is 9216.
`udp_recvspace`	The default size of the UDP receive buffer. The default is 41600, allowing for 40 1024-byte datagrams.

E.2 SunOS 4.1.3

The method used with SunOS 4.1.3 is similar to what we saw with BSD/386. Since most of the kernel sources are not distributed, all the C variable initializations are contained in a single C source file that is provided.

The administrator's kernel configuration file (see the `config`(8) manual page) can define the following variables. After modifying your configuration file, a new kernel must be made and rebooted.

`IPFORWARDING`
The value of this constant initializes the kernel variable `ip_forwarding`. If -1, IP datagrams are never forwarded. Furthermore, the variable is never changed. If 0 (default), IP datagrams are not forwarded, but the variable's value is changed to 1 if multiple interfaces are up. If 1, forwarding is always enabled.

`SUBNETSARELOCAL`
The value of the kernel variable `ip_subnetsarelocal` is initialized from this constant. If 1 (default), a destination IP address with the same network ID as the sending

host but a different subnet ID is considered local. If 0, only destination IP addresses on an attached subnet are considered local. This is summarized in Figure E.1. When sending to local destinations, TCP chooses the MSS based on the MTU of the outgoing interface. When sending to nonlocal destinations, TCP uses the variable `tcp_default_mss`.

`IPSENDREDIRECTS`
The value of this constant initializes the kernel variable `ip_sendredirects`. If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

`DIRECTED_BROADCAST`
The value of this constant initializes the kernel variable `ip_dirbroadcast`. If 1 (default), received datagrams whose destination .address is the directed broadcast address of an attached interface are forwarded as a link-layer broadcast. If 0, these datagrams are silently discarded.

The file `/usr/kvm/sys/netinet/in_proto.c` defines the following variables that can be changed. Once these variables are changed, a new kernel must be made and rebooted.

`tcp_default_mss`	The default TCP MSS for nonlocal destinations. Default value is 512.
`tcp_sendspace`	The default size of the TCP send buffer. Default value is 4096.
`tcp_recvspace`	The default size of the TCP receive buffer. This affects the window size that is offered. Default value is 4096.
`tcp_keeplen`	A keepalive probe to a 4.2BSD host must contain a single byte of data to get a response. Set the variable to 1 for compatibility with these older implementations. Default value is 1.
`tcp_ttl`	The default value for the TTL field for TCP segments. Default value is 60.
`tcp_nodelack`	If nonzero, ACKs are not delayed. Default value is 0.
`tcp_keepidle`	Number of 500-ms clock ticks before sending a keepalive probe. Default value is 14400 (2 hours).
`tcp_keepintvl`	Number of 500-ms clock ticks between successive keepalive probes, when no response is received. Default value is 150 (75 seconds).

udp_cksum	If nonzero, UDP checksums are calculated for outgoing UDP datagrams, and incoming UDP datagrams containing nonzero checksums have their checksum verified. If 0, outgoing UDP datagrams do not contain a checksum, and no checksum verification is performed on incoming UDP datagrams, even if the sender calculated a checksum. Default is 0.
udp_ttl	The default value for the TTL field in UDP datagrams. Default value is 60.
udp_sendspace	The default size of the UDP send buffer. Defines the maximum UDP datagram that can be sent. Default is 9000.
udp recvspace	The default size of the UDP receive buffer. The default is 18000, allowing for two 9000-byte datagrams.

E.3 System V Release 4

The TCP/IP configuration of SVR4 is similar to the previous two systems, but fewer options are available. In the file `/etc/conf/pack.d/ip/space.c` two constants can be defined, and the kernel must then be rebuilt and rebooted.

`IPFORWARDING`
The value of this constant initializes the kernel variable `ipforwarding`. If 0 (default), IP datagrams are not forwarded. If 1, forwarding is always enabled.

`IPSENDREDIRECTS`
The value of this constant initializes the kernel variable `ipsendredirects`. If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

Many of the variables that we've described in the previous two sections are defined in the kernel, but one must patch the kernel to modify them. For example, there is a variable named `tcp_keepidle` with a value of 14400.

E.4 Solaris 2.2

Solaris 2.2 is typical of the newer Unix systems that provide a program for the administrator to run to change the configuration options of the TCP/IP system. This allows reconfiguration without having to modify source files and rebuild a kernel.

The configuration program is `ndd`(l). We can run the program to see what parameters we can examine or modify in the UDP module:

```
solaris % ndd /dev/udp \?
udp_wroff_extra                 (read and write)
udp_def_ttl                     (read and write)
udp_first_anon_port             (read and write)
udp_trust_optlen                (read and write)
udp_do_checksum                 (read and write)
udp_status                      (read only)
```

There are five modules we can specify: `/dev/ip`, `/dev/icmp`, `/dev/arp`, `/dev/udp`, and `/dev/tcp`. The question mark argument (which we have to prevent the shell from interpreting by preceding it with a backslash) tells the program to list all the parameters for that module. An example that queries the value of a variable is:

```
solaris % ndd /dev/tcp tcp_mss_def
536
```

To change the value of a variable we need superuser privilege and type:

```
solaris # ndd -set /dev/ip ip_forwarding 0
```

These variables can be divided into three categories:

1. Configuration variables that a system administrator can change (e.g., `ip_forwarding`).
2. Status variables that can only be displayed (e.g., the ARP cache). Normally this information is provided in an easier to understand format by the commands `ifconfig`, `netstat`, and `arp`.
3. Debugging variables intended for those with kernel source code. Enabling some of these generates kernel debug output at runtime, which can degrade performance.

We now describe the parameters in each module. All parameters are read-write, unless marked "(Read only)." The read-only parameters are the status variables from case 2 above. We also mark the "(Debug)" variables from case 3. Unless otherwise noted, all the timing variables are specified in milliseconds, which differs from the other systems that normally specify times as some number of 500-ms clock ticks.

/dev/ip

`ip_cksum_choice`
(Debug) Selects between two independent implementations of the IP checksum

algorithm.

`ip_debug`
(Debug) Enables printing of debug output by the kernel, if greater than 0. Larger values generate more output. Default is 0.

`ip_def_ttl`
Default TTL for outgoing IP datagrams, if not specified by transport layer. Default is 255.

`ip_forward_directed_broadcasts`
If 1 (default), received datagrams whose destination address is the directed broadcast address of an attached interface are forwarded as a link-layer broadcast. If 0, these datagrams are silently discarded.

`ip_forward_src_routed`
If 1 (default), received datagrams containing a source route option are forwarded. If 0, these datagrams are discarded.

`ip_forwarding`
Specifies whether the system forwards incoming IP datagrams: 0 means never forward, 1 means always forward, and 2 (default) means only forward when two or more interfaces are up.

`ip_icmp_return_data_bytes`
The number of bytes of data beyond the IP header that are returned in an ICMP error. Default is 64.

`ip_ignore_delete_time`
(Debug) Minimum lifetime of an IP routing table entry (IRE). Default is 30 seconds. (This parameter is in seconds, not milliseconds.)

`ip_ill_status`
(Read only) Displays the status of each IP lower layer data structure. There is one lower layer structure for each interface.

`ip_ipif_status`
(Read only) Displays the status of each IP interface data structure (IP address, subnet mask, etc.). There is one of these structures for each interface.

`ip_ire_cleanup_interval`
(Debug) The interval at which the IP routing table entries are scanned for possible deletions. Default is 30000 ms (30 seconds).

`ip_ire_flush_interval`
The interval at which ARP information in unconditionally flushed from the IP routing table. Default is 1200000 ms (20 minutes).

`ip_irepathmtu_interval`
The interval at which the path MTU discovery algorithm tries to increase the MTU. Default is 30000 ms (30 seconds).

`ip_ire_redirect_interval`
The interval at which IP routing table entries that are from ICMP redirects are deleted. Default is 60000 ms (60 seconds).

`ip_ire_status`
(Read only) Displays all the IP routing table entries.

`ip_local_cksum`
If 0 (default), IP does not calculate the IP checksum or the higher layer protocol checksum (i.e., TCP, UDP, ICMP, or IGMP) for datagrams sent or received through the loopback interface. If 1, these checksums are calculated.

`ip_mrtdebug`
(Debug) Enables printing of debug output concerning multicast routing by the kernel, if 1. Default is 0.

`ip_path_mtu_discovery`
If 1 (default), path MTU discovery is performed by IP. If 0, IP never sets the "don't fragment" bit in outgoing datagrams.

`ip_respond_to_address_mask`
If 0 (default), the host does not respond to ICMP address mask requests. If 1, it does respond.

`ip_respond_to_echo_broadcast`
If 1 (default), the host responds to ICMP echo requests that are sent to a broadcast address. If 0, it does not respond.

`ip_respond_to_timestamp`
If 0 (default), the host does not respond to ICMP timestamp requests. If 1, the host responds.

`ip_respond_to_timestamp_broadcast`
If 0 (default), the host does not respond to ICMP timestamp requests that are sent to a

broadcast address. If 1, it does respond.

`ip_rput_pullups`
(Debug) Count of number of buffers from the network interface driver that needed to be pulled up to access the full IP header. Initialized to 0 at bootstrap time, and can be reset to 0.

`ip_send_redirects`
If 1 (default), the host sends ICMP redirects when acting as a router. If 0, these are not sent.

`ip_send_source_quench`
If 1 (default), the host generates ICMP source quench errors when incoming datagrams are discarded. If 0, these are not generated.

`ip_wroff_extra`
(Debug) Number of bytes of extra space to allocate in buffers for IP headers. Default is 32.

/dev/icmp

`icmp_bsd_compat`
(Debug) If 1 (default), the length field in the IP header of received datagrams is adjusted to exclude the length of the IP header. This is compatible with Berkeley-derived implementations and is for applications reading raw IP or raw ICMP packets. If 0, the length field is not changed.

`icmp_def_ttl`
The default TTL for outgoing ICMP messages. Default is 255.

`icmp_wroff_extra`
(Debug) Number of bytes of extra space to allocate in buffers for IP options and data-link headers. Default is 32.

/dev/arp

`arp_cache_report`
(Readonly) The ARP cache.

`arp_cleanup_interval`
The interval after which ARP entries are discarded from ARP's cache. Default is 300000 ms (5 minutes). (IP maintains its own cache of completed ARP translations; see ip_ire flush_interval.)

`arp_debug`
(Debug) If 1, enables printing of debug output by the ARP driver. Default is 0.

/dev/udp

`udp_def_ttl`
The default TTL for outgoing UDP datagrams. Default value is 255.

`udp_do_checksum`
If 1 (default), UDP checksums are calculated for outgoing UDP datagrams. If 0, outgoing UDP datagrams do not contain a checksum. (Unlike most other implementations, this UDP checksum flag does not affect incoming datagrams. If a received datagram has a nonzero checksum, it is always verified.)

`udp_largest_anon_port`
Largest port number to allocate for UDP ephemeral ports. Default is 65535.

`udp_smallest_anon_port`
Starting port number to allocate for UDP ephemeral ports. Default is 32768.

`udp_smallest_nonpriv_port`
A process requires superuser privilege to assign itself a port number less than this. Default is 1024.

`udp_status`
(Read only) The status of all local UDP end points: local IP address and port, foreign IP address and port.

`udp_trust_optlen`
(Debug) No longer used.

`udp_wroff_extra`
(Debug) Number of bytes of extra space to allocate in buffers for IP options and data-link headers. Default is 32.

/dev/tcp

`tcp_close_wait_interval`
The 2MSL value: the time spent in the TIME_WAIT state. Default is 240000 ms (4 minutes).

`tcp_conn_grace_period`

(Debug) Additional time added to the timer interval when sending a SYN. Default is 500 ms.

`tcp_conn_req_max`
The maximum number of pending connection requests queued for any listening end point. Default is 5.

`tcp_cwnd_max`
The maximum value of the congestion window. Default is 32768.

`tcp_debug`
(Debug) If 1, enables printing of debug output by TCP. Default is 0.

`tcp_deferred_ack_interval`
The time to wait before sending a delayed ACK. Default is 50 ms.

`tcp_dupack_fast_retransmit`
The number of consecutive duplicate ACKs that triggers the fast retransmit, fast recovery algorithm. Default is 3.

`tcp_eager_listeners`
(Debug) If 1 (default), TCP completes the three-way handshake before returning a new connection to an application with a pending passive open. "This is the way most TCP implementations operate. If 0, TCP passes an incoming connection request (received SYN) to the application, and does not complete the three-way handshake until the application accepts the connection. (Setting this to 0 might break many existing applications.)

`tcp_ignore_path_mtu`
(Debug) If 1, path MTU discovery ignores received ICMP fragmentation needed messages. If 0 (default), path MTU discovery is enabled for TCP.

`tcp_ip abort_cinterval`
The total retransmit timeout value when TCP is performing an active open. Default is 240000 ms (4 minutes).

`tcp_ip_abort_interval`
The total retransmit timeout value for a TCP connection after it is established. Default is 120000 ms (2 minutes).

`tcp_ip_notify_cinterval`
The timeout value when TCP is performing an active open after which TCP notifies IP to find a new route. Default is 10000 ms (10 seconds).

`tcp_ip_notify_interval`
The timeout value for an established connection after which TCP notifies IP to find a new route. Default is 10000 ms (10 seconds).

`tcp_ip_ttl`
The TTL to use for outgoing TCP segments. Default is 255.

`tcp_keepalive_interval`
The time that a connection must be idle before a keepalive probe is sent. Default is 7200000 ms (2 hours).

`tcp_largest_anon_port`
Largest port number to allocate for TCP ephemeral ports. Default is 65535.

`tcp_maxpsz_multiplier`
(Debug) Specifies the multiple of the MSS into which the stream head packetizes the application's write data. Default is 1.

`tcp_mss_def`
Default MSS for nonlocal destinations. Default is 536.

`tcp_mss_max`
The maximum MSS. Default is 65495.

`tcp_mss_min`
The minimum MSS. Default is 1.

`tcp_naglim_def`
(Debug) Maximum value of the per-connection Nagle algorithm threshold. Default is 65535. The per-connection value starts out as the minimum of the MSS or this value. The per-connection value is set to 1 by the TCP_NODELAY socket option, which disables the Nagle algorithm.

`tcp_old_urp_interpretation`
(Debug) If 1 (default), the older (but more common) BSD interpretation of the urgent pointer is used: it points 1 byte beyond the last byte of urgent data. If 0, the Host Requirements RFC interpretation is used; it points to the last byte of urgent data.

`tcp_rcv_push_wait`
(Debug) Maximum number of bytes received without the PUSH flag set before the data is passed to the application. Default is 16384.

`tcp_rexmit_interval_initial`
(Debug) Initial retransmit timeout interval. Default is 500 ms.

`tcp_rexmit_interval_max`
(Debug) Maximum retransmit timeout interval. Default is 60000 ms (60 seconds).

`tcp_rexmit_interval_min`
(Debug) Minimum retransmit timeout interval. Default is 200 ms.

`tcp_rwin_credit_pct`
(Debug) Percentage of receive window that must be buffered before flow control is checked on every received segment. Default is 50%.

`tcp_smallest_anon_port`
Starting port number to allocate for TCP ephemeral ports. Default is 32768.

`tcp_smallest_nonpriv_port`
A process requires superuser privilege to assign itself a port number less than this. Default is 1024.

`tcp_snd_lowat_fraction`
(Debug) If nonzero, the send buffer low-water mark is the send buffer size divided by this value. Default is 0 (disabled).

`tcp_status`
(Read only) Information on all TCP connections.

`tcp_sth_rcv_hiwat`
(Debug) If nonzero, the value to set the stream head high-water mark to. Default is 0.

`tcp_sth_rcv_lowat`
(Debug) If nonzero, the value to set the stream head low-water mark to. Default is 0.

`tcp_wroff_xtra`
(Debug) Number of bytes of extra space to allocate in buffers for IP options and data-link headers. Default is 32.

E.5 AIX 3.2.2

AIX 3.2.2 allows network options to be set at runtime using the no command. It can display the value of an option, set the value of an option, or set an option value back to its default. For example, to display an option we type:

```
aix % no -o udp_ttl
udp_ttl = 30
```

The following options can be modified.

`arpt_killc`
The time (in minutes) before an inactive completed ARP entry is removed. Default is 20.

`ipforwarding`
If 1 (default), IP datagrams are always forwarded. If 0, forwarding is disabled.

`ipfragttl`
The time to live (in seconds) for IP fragments awaiting reassembly. Default is 60.

`ipsendredirects`
If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

`loop_check_sum`
If 1 (default), the IP checksum is calculated for datagrams sent through the loop-back interface. If 0, this checksum is not calculated.

`nonlocsrcroute`
If 1 (default), received datagrams containing a source route option are forwarded. If 0, these datagrams are discarded.

`subnetsarelocal`
If 1 (default), a destination IP address with the same network ID as the sending host but a different subnet ID is considered local. If 0, only destination IP addresses on an attached subnet are considered local. This is summarized in Figure E.1. When sending to local destinations, TCP chooses the MSS based on the MTU of the outgoing interface. When sending to nonlocal destinations, TCP uses the default (536) as the MSS.

`tcp_keepidle`
Number of 500-ms clock ticks before sending a keepalive probe. Default value is 14400 (2 hours).

`tcp_keepintvl`
Number of 500-ms clock ticks between successive keepalive probes, when no response is received. Default value is 150 (75 seconds).

`tcp_recvspace`
The default size of the TCP receive buffer. This affects the window size that is offered.

Default value is 16384.

`tcp_sendspace`
The default size of the TCP send buffer. Default value is 16384.

`tcp_ttl`
The default value for the TTL field for TCP segments. Default value is 60.

`udp_recvspace`
The default size of the UDP receive buffer. The default is 41600, allowing for 40 1024-byte datagrams.

`udp_sendspace`
The default size of the UDP send buffer. Defines the maximum UDP datagram that can be sent. Default is 9216.

`udp_ttl`
The default value for the TTL field in UDP datagrams. Default value is 30.

E.6 4.4BSD

4.4BSD is the first of the Berkeley releases to provide dynamic configuration for numerous kernel parameters. The `sysctl`(8) command is used. The names for the parameters were chosen to look like MIB names from SNMP. To examine a parameter we type:

```
vangogh % sysctl net.inet.ip.forwarding
net.inet.ip.forwarding = 1
```

To change a parameter we need superuser privilege and then type:

```
vangogh # sysctl -w nat.inet.ip.ttl=128
```

The following parameters can be changed.

`net.inet.ip.forwarding`
If 0 (default), IP datagrams are not forwarded. If 1, forwarding is enabled.

`net.inet.ip.redirect`
If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

`net.inet.ip.tti`
The default TTL for both TCP and UDP. The default is 64.

`net.inet.icmp.maskrepi`
If 0 (default), the host does not respond to ICMP address mask requests. If 1, it does respond.

`net.inet.udp.checksum`
If 1 (default), UDP checksums are calculated for outgoing UDP datagrams, and incoming UDP datagrams containing nonzero checksums have their checksum verified. If 0, outgoing UDP datagrams do not contain a checksum, and no checksum verification is performed on incoming UDP datagrams, even if the sender calculated a checksum.

Additionally, numerous variables that we've described earlier in this appendix are scattered among various source files (`tcp_keepidle`, `subnetsarelocal`, etc.) and can be modified.

Source Code Availability

This book uses many publicly available software packages. This appendix provides additional details on how to obtain this software.

The technique used to obtain this software is called *anonymous FTP*, where FTP is the standard Internet File Transfer Protocol (Chapter 27). Section 27.3 shows an example of anonymous FTP. For a background on Internet resources in general, and specifically anonymous FTP, refer to any of the recently available books on the Internet, such as [LaQuey 1993] or [Krol 1992].

The hosts listed here are believed to be the primary site where the package is available. There may be many other sites where the software is also available. The Internet Archie service can locate additional versions. Also, the versions listed below are the ones used for the examples in the text.

Newer versions may have been released by the time you read this.

You should use the FTP `dir` command to see if newer versions exist on that specified host.

This appendix is ordered by the chapter or section number where the resource was used in this text.

RFCs (Section 1.11)

Section 1.11 provides the electronic mail address to send a request to. The reply details numerous sites from which the RFCs can be obtained using either e-mail or anonymous FTP.

Remember that the starting place is to obtain the current index and look up the RFC that you want in the index. This entry tells you if that RFC has been obsoleted or updated by a newer RFC.

BSD Net/2 Source Code (Section 1.14)

The BSD Net/2 source code, which includes the kernel implementation of the TCP/IP protocols, along with the standard utilities (Telnet client and server, FTP client and server, etc.), is available from `ftp.uu.net` in the directory tree starting at `systems/unix/bsd-sources`.

SLIP (Section 2.4)

The version of SLIP used in this text is available from `ftp.ee.lbl.gov`. The filename begins with `cslip`, since it supports compressed SLIP (Section 2.5).

`icmpaddrmask` Program (Section 6.3)

Refer to the final entry of this section.

`icmptime` Program (Section 6.4)

Refer to the final entry of this section.

`ping` Program (Chapter 7)

The BSD version of `ping` normally has more options and features than the version supplied by many vendors. The host `ftp.uu.net` contains the latest BSD version in the file `systems/unix/bsd-sources/sbin/ping`.

`traceroute` Program (Chapter 8)

The traceroute program is available from `ftp.ee.lbl.gov`. Refer to the final entry of this section for the version used in Section 8.5 that allows loose and strict source routing.

Router Discovery Daemon (Section 9.6)

A program is available that provides host support and router support for the router discovery messages. The host is `gregorio.stanford.edu` and the file is `gw-discovery/nordmark-rdisc.tar`. The program was written by Sun Microsystems and made publicly available.

`gated` Daemon (Section 10.3)

The `gated` routing daemon, mentioned in Section 10.3, is available from the host `gated.cornell.edu`.

`traceroute.pmtu` Program (Section 11.7)

Refer to the final entry of this section.

IP Multicasting Software (Chapter 13)

The modifications required to support IP multicasting for SunOS 4.x and Ultrix are available from `gregorio.stanford.edu` in the directory `vmtp-ip`. This directory also contains the source code modifications required to implement IP multicasting in a Berkeley Unix system.

BIND Name Server (Chapter 14)

The BIND name server, the named daemon, is available from the host `ftp.uu.net` in the file `networking/ip/dns/bind/bind.4.8.3.tar.Z`.

A newer version, 4.9, is available from `gatekeeper.dec.com` in the directory `pub/BSD/bind/4.9`.

`host` Program (Chapter 14)

The `host` program is available from the host `nikhefh.nikhef.nl` in the file `host.tar.Z`.

`dig` and doc Programs (Chapter 14)

The `dig` and `doc` programs mentioned in Chapter 14 are available from the host `isi.edu` in the files `dig.2.0.tar.Z` and `doc.2.0.tar.Z`.

BOOTP Server (Chapter 16)

Various versions of the commonly used Unix BOOTP server are available from the host `lancaster.andrew.emu.edu`, in the `pub` directory.

TCP High-Speed Extensions (Chapter 24)

A publicly available source code implementation of the TCP window scale option, time-stamp option, and PAWS algorithm is available as a set of patches to the BSD Net/2 release from the host `uxc.cso.uiuc.edu` in the file `pub/tcplw.shar.Z`.

ISODE SNMP Manager and Agent (Chapter 25)

The SNMP manager and agent described in Section 25.7 are part of the ISODE 8.0 distribution. This is available from many FTP archive sites, such as `ftp.uu.net` in the

networking/osi/isode directory.

MIME Software and Examples (Section 28.4)

A program named MetaMail that provides MIME capabilities for many different user agents is available on the host thumper.bellcore.com in the pub/nsb directory. Also in this directory is additional information on MIME.

Sun RPC (Section 29.2)

A version of the RPC 4.0 sources (which use the sockets API) is available from the host ftp.uu.net in the systems/sun/sextape/rpc4.0 directory. A version of the TI-RPC sources (which use the TLI API) is available from the host ftp.uu.net in the networking/rpc directory.

Sun NFS (Chapter 29)

A publicly available implementation of an NFS client and server is provided as part of the BSD Net/2 Source Code described earlier in this appendix.

tcpdump Program (Appendix A)

The version of tcpdump used in this text is from the host ftp.ee.lbl.gov in the file tcpdump-2.2.1.tar.Z.

BSD Packet Filter (Section A.1)

The BSD packet filter is part of the tcpdump distribution.

sock Program (Appendix C)

Refer to the final entry of this section.

ttcp Program

(This program was not used in the text, but is a useful tool of which readers should be aware.) ttcp is a benchmarking tool for measuring TCP and UDP performance between two systems. It was created at the U.S. Army Ballistics Research Lab (BRL) and is in the public domain. Copies are available from many anonymous FTP sites but an enhanced version is available from ftp.sgi.com in the directory sgi/src/ttcp.

Author-Written Software

The author-written software used in the book is available from the host `ftp.uu.net` in the file `published/books/stevens.tcpipiv1.tar.Z`.

ACRONYMS

ACK	acknowledgment flag, TCP header
API	application program interface
ARP	Address Resolution Protocol
ARPANET	Advanced Research Projects Agency network
AS	autonomous system
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
BGP	Border Gateway Protocol
BIND	Berkeley Internet Name Domain
BOOTP	Bootstrap Protocol
BPF	BSD Packet Filter
BSD	Berkeley Software Distribution
CIDR	classless interdomain routing
CIX	Commercial Internet Exchange
CLNP	Connectionless Network Protocol
CRC	cyclic redundancy check
CSLIP	compressed SLIP
CSMA	carrier sense multiple access
DCE	Distributed Computing Environment
DDN	Defense Data Network
DF	don't fragment flag, IP header
DHCP	Dynamic Host Configuration Protocol
DLPI	Data Link Provider Interface
DNS	Domain Name System
DSAP	Destination Service Access Point
DTS	Distributed Time Service
DVMRP	Distance-Vector Multicast Routing Protocol
EBONE	European IP Backbone
EGP	Exterior Gateway Protocol
EOL	end of option list
FCS	frame check sequence
FDDI	Fiber Distributed Data Interface
FIFO	first in, first out

FIN	finish flag, TCP header
FQDN	fully qualified domain name
FTP	File Transfer Protocol
HDLC	high-level data link control
HELLO	routing protocol
IAB	Internet Architecture Board
IANA	Internet Assigned Number Authority
ICMP	Internet Control Message Protocol
IDRP	Interdomain Routing Protocol
IEEE	Institute of Electrical and Electronics Engineers
IEN	Internet Experiment Notes
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IGP	interior gateway protocol
IP	Internet Protocol
IRTF	Internet Research Task Force
IS-IS	Intermediate System to Intermediate System Protocol
ISN	initial sequence number
ISO	International Organization for Standardization
ISOC	Internet Society
LAN	local area network
LBX	low bandwidth X
LCP	link control protocol
LFN	long fat network
LIFO	last in, first out
LLC	logical link control
LSRR	loose source and record route
MBONE	multicast backbone
MIB	management information base
MILNET	Military Network
MIME	multipurpose Internet mail extensions
MSL	maximum segment lifetime
MSS	maximum segment size
MTA	message transfer agent
MTU	maximum transmission unit

NCP	Network Control Protocol
NFS	Network File System
NIC	Network Information Center
NIT	network interface tap
NNTP	Network News Transfer Protocol
NOAO	National Optical Astronomy Observatories
NOP	no operation
NSFNET	National Science Foundation network
NSI	NASA Science Internet
NTP	Network Time Protocol
NVT	network virtual terminal
OSF	Open Software Foundation
OSI	open systems interconnection
OSPF	open shortest path first
PAWS	protection against wrapped sequence numbers
PDU	protocol data unit
POSIX	Portable Operating System Interface
PPP	Point-to-Point Protocol
PSH	push flag, TCP header
RARP	Reverse Address Resolution Protocol
RFC	Request for Comment
RIP	Routing Information Protocol
RPC	remote procedure call
RR	resource record
RST	reset flag, TCP header
RTO	retransmission time out
RTT	round-trip time
SACK	selective acknowledgment
SLIP	Serial Line Internet Protocol
SMI	structure of management information
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SSAP	source service access point
SSRR	strict source and record route
SWS	silly window syndrome
SYN	synchronize sequence numbers flag, TCP header

TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TLI	Transport Layer Interface
TOS	type-of-service
TTL	time-to-live
TUBA	TCP and UDP with bigger addresses
Telnet	remote terminal protocol
UDP	User Datagram Protocol
URG	urgent pointer flag, TCP header
UTC	Coordinated Universal Time
UUCP	Unix-to-Unix Copy
WAN	wide area network
WWW	World Wide Web
XDR	external data representation
XID	transaction ID
XTI	X/Open Transport Layer Interface