Fall 2005 Handout 3

The IP version 6 address

```
1$ cat -n /usr/include/netinet/in.h | sed -n 79,101p
    79  struct in6_addr {
    80      union {
    81          /*
    82           * Note: Static initalizers of "union" type assume
    83           * the constant on the RHS is the type of the first member
    84           * of union.
    85           * To make static initializers (and efficient usage) work,
    86           * the order of members exposed to user and kernel view of
    87           * this data structure is different.
    88           * User environment sees specified uint8_t type as first
    89           * member whereas kernel sees most efficient type as
    90           * first member.
    91           */
    92  #ifdef _KERNEL
    93          uint32_t _S6_u32[4];  /* IPv6 address */
    94          uint8_t      _S6_u8[16];  /* IPv6 address */
    95  #else
    96          uint8_t      _S6_u8[16];  /* IPv6 address */
    97          uint32_t _S6_u32[4];  /* IPv6 address */
    98  #endif
    99          uint32_t __S6_align;  /* Align on 32 bit boundary */
   100      } _S6_un;
   101  };

2$ grep 'typedef.*uint8_t;' /usr/include/sys/int_types.h
typedef unsigned char uint8_t;
```

The IP version 6 header

```
1$ cat -n /usr/include/netinet/ip6.h | sed -n '23,38p' | more
    23  struct  ip6_hdr {
    24      union {
    25          struct ip6_hdrctl {
    26              uint32_t ip6_un1_flow;   /* 4 bits version, */
    27                                       /* 8 bits tclass, and */
    28                                       /* 20 bits flow-ID */
    29              uint16_t ip6_un1_plen;   /* payload length */
    30              uint8_t     ip6_un1_nxt;    /* next header */
    31              uint8_t     ip6_un1_hlim;   /* hop limit */
    32          } ip6_un1;
    33          uint8_t  ip6_un2_vfc; /* 4 bits version and */
    34                          /* top 4 bits of tclass */
    35      } ip6_ctlun;
    36      struct in6_addr ip6_src; /* source address */
    37      struct in6_addr ip6_dst; /* destination address */
    38  };
```

▼ Homework 3.1: compute the offset of each field

Here the offset and size in bytes of each field of the header of an Ethernet frame. The offset of each field is the sum of the sizes of all the previous fields:

name	*offset*	*size*
ether_dhost	0	6
ether_shost	6	6
ether_type	12	2

Here is the header of an ARP packet:

name	*offset*	*size*
ar_hrd	0	2
ar_pro	2	2
ar_hln	4	1
ar_pln	5	1
ar_op	6	2
arp_sha	8	6
arp_spa	14	4
arp_tha	18	6
arp_tpa	24	4

Make similar tables for an IP datagram, UDP datagram, TCP segment, and at least the first three fields of an ICMP message. Assume that the IP datagram and TCP segment contain no options. If two fields occupy the same byte, list them on the same line. For example, the fields of the header of an IP datagram begin with

name	*offset*	*size*
ip_v *and* **ip_hl**	0	1
ip_tos	1	1
ip_len	2	2
etc.		

We'll need these numbers when we give bitwise command line arguments to **snoop**.

▲

Display selected packets

```
1$ cd ~mm64/public_html/x52.9547/src/snoop
2$ pwd
3$ ls -l tcp.snoop
-r--r--r--   1 mm64     users        904 May 29  2003 tcp.snoop
```

-ta is absolute time; **-V** is the second-to-highest level of verbosity.

```
4$ snoop -i tcp.snoop -ta -V | fold -b -w 80 | more
5$ snoop -i tcp.snoop -ta -V host 192.168.0.5 | fold -b -w 80 | more
6$ snoop -i tcp.snoop -ta -V to host 192.168.0.5 | fold -b -w 80 | more
7$ snoop -i tcp.snoop -ta -V from host aixmita1.urz.uni-heidelberg.de |
    fold -b -w 80 | more
```

(1) **ip[2:2]** is the two-byte number occupying the third and fourth bytes of the header of the IP datagram. The bytes are numbered starting at zero; **:2** means ‘‘a two-byte number’’. **ip[2:2]** is the total length of the IP datagram; see line 53 of **ip.h** in Handout 2, p. 8. The expression must be in **’**single quotes**’** because of characters such as **[** and **]** which have a special meaning to the shell. Display the IP datagrams that are at least 64 bytes long:

```
8$ snoop -i tcp.snoop -ta 'ip[2:2] >= 64' | fold -b -w 80 | more
1 14:18:42.92447  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
2 14:18:43.16891 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727 Thu May 29 20:18:41
```

(2) **ip[0:1]** is the one-byte number occupying the first byte of the header of the IP datagram. **:1** means ‘‘just one byte’’. The four highest bits of this byte give the version number of IP; see line 49−50 of **ip.h** in Handout 2, p. 8. We use the ‘‘bitwise and’’ in Handout 1, pp. 7−8 to isolate the version number, which should be 4 for IPv4.

```
   01000101 == ip[0:1]
&  11110000 == 0xF0
-----------
   01000000 == 0x40
```
Bits 4−7 survive unchanged; bits 0−3 are turned off.

This should output all nine packets:

```
9$ snoop -i tcp.snoop -ta 'ip[0:1] & 0xF0 == 0x40' | fold -b -w 80 | more
```

(3) **tcp[13:1]** is the one-byte number occupying the fourteenth byte of the header of the TCP segment. (The bytes are numbered starting at zero.) **tcp[13:1]** contains the TCP flags; see lines 43−49 of **tcp.h** in Handout 2, p. 26. Line 45 shows that the synchronize flag is bit 1. Recall that bit 0 is the rightmost bit (Handout 1, p. 1).

```
  00010010 == tcp[13:1]; maybe several of the flags are turned on.
& 00000010 == 0x02
----------
  00000010 == 0x02
```
Bit 1 survives unchanged; the other seven are turned off.

```
  00010000 == Here's a tcp[13:1] whose bit 1 is turned off.
& 00000010 == 0x02
----------
  00000000 == 0x00
```
Once again, bit 1 survives unchanged; the other seven are turned off.

Display the TCP segments where this flag is turned on:

```
10$ snoop -i tcp.snoop -ta 'tcp[13:1] & 0x02 == 0x02' | fold -b -w 80 | more
1 14:18:42.92447  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
2 14:18:43.04544 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727
```

(4) Display the packet(s) where the synchronize flag (bit 1) and the acknowledgement flag (bit 4) are both on:

```
    00010010 == 0x12

11$ snoop -i tcp.snoop -ta 'tcp[13:1] & 0x12 == 0x12' | fold -b -w 80 | more
1 14:18:43.04544 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727
```

(5) Display the packet(s) where the synchronize flag (bit 1) or the fin flag (bit 0) are on. This will pick out the packets at the start and end of each TCP conversation.

```
    00000011 == 0x03

12$ snoop -i tcp.snoop -ta 'tcp[13:1] & 0x03 != 0x00' | fold -b -w 80 | more
1 14:18:42.92447  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
2 14:18:43.04544 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727
3 14:18:43.16945 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727
4 14:18:43.16987  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
```

(6) The four lowest bits of **`ip[0:1]`**, when multiplied by four, give the length of the IP header. Use **`& 0x0F`** to zero out the four highest bits of this byte:

```
  01000101 == ip[0:1]
& 00001111 == 0x0F
  --------
  00000101
```
The four lowest bits survive unchanged; the other four are turned off.

Display the IP datagrams whose headers are exactly 20 bytes long:

```
13$ snoop -ta -i packetfile 'ip[0:1] & 0x0F == 5' | fold -b -w 80 | more
14$ snoop -ta -i packetfile '4 * (ip[0:1] & 0x0F) == 20' | fold -b -w 80 | more
```
all nine packets

(7) **`tcp[12:1]`** is the one-byte number occupying the thirteenth byte of the header of the TCP segment. The four highest bits of **`tcp[12:1]`**, when multiplied by four, give the length of the TCP header. See lines 40−41 of **`tcp.h`** in Handout 2, p. 26. Use **`& 0xF0`** to zero out the four lowest bits of this byte. Unfortunately, the four highest bits are already multiplied by $16 = 2^4$ because of the four bits below them. We should therefore *divide* the four highest bits by four to end up with a net multiplication of 4. For example, to display the TCP segments whose headers are longer than 32,

```
15$ snoop -ta -i packetfile '(tcp[12:1] & 0xF0) / 4 > 32' | fold -b -w 80 | more
```

But **`snoop`** has no division or shift, so we get rid of the division by multiplying both sides of the inequality by 4:

```
16$ snoop -i tcp.snoop -ta '(tcp[12:1] & 0xF0) > 4 * 32' | fold -b -w 80 | more
1 14:18:42.92447  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
2 14:18:43.04544 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727
```

(8) Display the packet(s) where the TCP segment carries cargo:

```
17$ snoop -i tcp.snoop -ta  \
    'ip[2:2] - 4 * (ip[0:1] & 0x0F) > (tcp[12:1] & 0xF0) / 4' | \
    fold -b -w 80 | more
```

Instead of dividing the above right side by 4, you have to multiply the left side by 4:

```
18$ snoop -i tcp.snoop -ta \
    '4 * (ip[2:2] - 4 * (ip[0:1] & 0x0F)) > tcp[12:1] & 0xF0' | \
    fold -b -w 80 | more
1 14:18:43.16891 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727 Thu May 29 20:18:41
```

▼ Homework 3.2: steal their password

The file **`~mm64/public_html/x52.9547/src/snoop/login.snoop`** has a snoop of someone logging into some host via **`telnet`** (TCP port 23). What host, and what is their login name and secret password? To begin with, how many packets are in this file? How many of them carry **`telnet`** packets? How many of them carry **`telnet`** packets to the host into which the poor soul is logging in?

Use the command line arguments of **`snoop`** and/or the Unix utilities (**`grep`**, etc).

▲

The echo servers (RFC 862)

There's a UDP echo server and a TCP echo server, both bound to port 7:

```
1$ awk '$1 == "echo"' /etc/services
echo    7/tcp
echo    7/udp
```

To talk to the TCP echo server, run the **`telnet`** client:

```
2$ telnet www.unipv.it 7
Trying 193.204.35.36...
Connected to www.unipv.it.
Escape character is '^]'.
Hello, there!
Hello, there!
Goodbye now.
Goodbye now.
^]                          Control-] is the telnet escape character.

telnet> quit
Connection closed.
3$
```

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9547/src/findecho`

```
1 #!/bin/ksh
```



```
 2 #Try to connect to the echo server of every host on the network.
 3
 4 for host in ‘localhosts.pl 131.211.194.00 24 | awk ’{print $2}’‘
 5 do
 6     telnet $host 7 &
 7     sleep 1
 8     if ps -p $! > /dev/null   #if the telnet is still running
 9     then
10         sleep 4
11         kill -9 $!            #kill the telnet
12     fi
13 done
```

The TCP echo server is easier to talk to than the UDP echo server: simply **`telnet`** to port 7. But the UDP echo server is easier to snoop on: there are fewer packets.

No fragmentation necessary

An Ethernet frame can carry at most 1500 bytes of cargo. An IP datagram in an Ethernet frame can therefore be at most 1500 bytes, so the IP datagram can carry at most 1500 − 20 = 1480 bytes of cargo. A UDP datagram in an IP datagram in an Ethernet frame can therefore be at most 1480 bytes, so the UDP datagram can carry at most 1480 − 8 = 1472 bytes of cargo.

The client program **`fragment.pl`** talks to the UDP echo server on another host.

The **`x`** in line 22 is the Perl ‘‘string multiply’’ operator. The function **`sockaddr_in`** in lines 29 and 36 can convert in either direction. **`sockaddr_in`** and the **`inet_ntoa`** in line 39 do no error checking.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9547/src/fragment/fragment.pl`

```
 1 #!/bin/perl
 2 #Send a UDP datagram with the specified number of bytes of cargo
 3 #to an echo server.  Display the UDP datagram that comes back.
 4
 5 use Socket;
 6
 7 if (@ARGV != 2) {
 8     die "$0: arguments: host bytecount"
 9 }
10
11 $name = $ARGV[0];
12 $size = $ARGV[1];
13
14 $ip = inet_aton($name) or die "$0: couldn’t get IP address for $name";
15
16 $port = getservbyname(’echo’, ’udp’)
17     or die "$0: couldn’t find udp service daytime";
18
19 socket(S, AF_INET, SOCK_DGRAM, 0) or die "$0: $!";
20 $address = sockaddr_in($port, $ip);
21
22 $n = send(S, ’A’ x $size, 0, $address);
23 defined $n or die "$0: $!";
24 if ($n != $size) {
25     die "$0: sent $n bytes instead of $size bytes\n";
26 }
27
```

```
28 $address = getsockname(S) or die "$0: $!";   #Can't do this before the send.
29 ($port, $ip) = sockaddr_in($address);
30 print "I sent a datagram from my port $port.\n";
31
32 $SIG{ALRM} = 'handle_sigalrm';
33 alarm(15);
34
35 $address = recv(S, $buffer, $size, 0) or die "$0: $!";
36 ($port, $ip) = sockaddr_in($address);
37
38 print "I then received a datagram containing the following ", length($buffer),
39     " bytes\nfrom port $port of ", inet_ntoa($ip);
40
41 $name = gethostbyaddr($ip, AF_INET);
42 if (defined $name) {
43     print " ($name)";
44 }
45
46 print "\n$buffer\n";
47 exit 0;
48
49 sub handle_sigalrm {
50     if ($_[0] ne 'ALRM') {
51         die "$0: received signal $_[0] instead of ALRM";
52     }
53     die "$0: The datagram I was waiting for didn't arrive in 15 seconds.\n";
54 }
```

```
1$ cd ~mm64/public_html/x52.9547/src/fragment
2$ snoop -o fragment1.snoop host cumulus.let.uu.nl &
[1] 12345

3$ fragment.pl cumulus.let.uu.nl 1472                    biggest possible cargo
I sent a datagram from my port 40224.
I then received a datagram containing the following 1024 bytes
from port 7 of 131.211.194.40 (cumulus.let.uu.nl)
(1024 A's)

4$ kill 12345
```

The conversation consists of two packets, one outgoing and one incoming. Here's a summary:

```
5$ cd ~mm64/public_html/x52.9547/src/fragment
6$ snoop -i fragment1.snoop | more
1   0.00000  192.168.0.5 -> cumulus.let.uu.nl ECHO C port=40224 AAAAAAAAAAAAAAAAAAAA
2   0.19812 cumulus.let.uu.nl -> 192.168.0.5  ECHO R port=40224 AAAAAAAAAAAAAAAAAAAA
```

Packet 1

The operating system selects a different ephemeral port number (line 35) each time we run the client (**40224** and **40225**).

```
1$ snoop -i fragment1.snoop -p 1,1 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 1 arrived at 15:25:45.23
     4  ETHER:  Packet size = 1514 bytes
     5  ETHER:  Destination = 0:10:4b:74:37:a,
     6  ETHER:  Source      = 0:d0:9:ff:57:5,
     7  ETHER:  Ethertype = 0800 (IP)
     8  ETHER:
     9  IP:   ----- IP Header -----
    10  IP:
    11  IP:   Version = 4
    12  IP:   Header length = 20 bytes
    13  IP:   Type of service = 0x00
    14  IP:         xxx. .... = 0 (precedence)
    15  IP:         ...0 .... = normal delay
    16  IP:         .... 0... = normal throughput
    17  IP:         .... .0.. = normal reliability
    18  IP:         .... ..0. = not ECN capable transport
    19  IP:         .... ...0 = no ECN congestion experienced
    20  IP:   Total length = 1500 bytes
    21  IP:   Identification = 30503
    22  IP:   Flags = 0x4
    23  IP:         .1.. .... = do not fragment
    24  IP:         ..0. .... = last fragment
    25  IP:   Fragment offset = 0 bytes
    26  IP:   Time to live = 255 seconds/hops
    27  IP:   Protocol = 17 (UDP)
    28  IP:   Header checksum = f83f
    29  IP:   Source address = 192.168.0.5, 192.168.0.5
    30  IP:   Destination address = 131.211.194.40, cumulus.let.uu.nl
    31  IP:   No options
    32  IP:
    33  UDP:  ----- UDP Header -----
    34  UDP:
    35  UDP:  Source port = 40224
    36  UDP:  Destination port = 7 (ECHO)
    37  UDP:  Length = 1480
    38  UDP:  Checksum = B4F1
    39  UDP:
    40  ECHO:  ----- ECHO:   -----
    41  ECHO:
    42  ECHO:  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    43  ECHO:
```

Packet 2

The echo server on **cumulus** echoed only 1032 − 8 = 1024 out of our 1472 bytes back to us (line 37), but who cares?

```
1$ snoop -i fragment1.snoop -p 2,2 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 2 arrived at 15:25:45.43
     4  ETHER:  Packet size = 1070 bytes
     5  ETHER:  Destination = 0:d0:9:ff:57:5,
     6  ETHER:  Source      = 0:10:4b:74:37:a,
     7  ETHER:  Ethertype = 0800 (IP)
     8  ETHER:
     9  IP:   ----- IP Header -----
    10  IP:
    11  IP:   Version = 4
    12  IP:   Header length = 20 bytes
    13  IP:   Type of service = 0x00
    14  IP:         xxx. .... = 0 (precedence)
    15  IP:         ...0 .... = normal delay
    16  IP:         .... 0... = normal throughput
    17  IP:         .... .0.. = normal reliability
    18  IP:         .... ..0. = not ECN capable transport
    19  IP:         .... ...0 = no ECN congestion experienced
    20  IP:   Total length = 1052 bytes
    21  IP:   Identification = 38861
    22  IP:   Flags = 0x4
    23  IP:         .1.. .... = do not fragment
    24  IP:         ..0. .... = last fragment
    25  IP:   Fragment offset = 0 bytes
    26  IP:   Time to live = 242 seconds/hops
    27  IP:   Protocol = 17 (UDP)
    28  IP:   Header checksum = e659
    29  IP:   Source address = 131.211.194.40, cumulus.let.uu.nl
    30  IP:   Destination address = 192.168.0.5, 192.168.0.5
    31  IP:   No options
    32  IP:
    33  UDP:  ----- UDP Header -----
    34  UDP:
    35  UDP:  Source port = 7
    36  UDP:  Destination port = 40224
    37  UDP:  Length = 1032
    38  UDP:  Checksum = D18A
    39  UDP:
    40  ECHO:  ----- ECHO:   -----
    41  ECHO:
    42  ECHO:  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    43  ECHO:
```

Fragmentation

```
1$ cd ~mm64/public_html/x52.9547/src/fragment
2$ snoop -o fragment2.snoop host cumulus.let.uu.nl &
[1] 12345
```

```
3$ fragment.pl cumulus.let.uu.nl 1473                    one byte too big
I sent a datagram from my port 40225.
I then received a datagram containing the following 1024 bytes
from port 7 of 131.211.194.40 (cumulus.let.uu.nl)
(1024 A's)

4$ kill 12345
```

This time there are two outgoing packets. **MF** is the ‘‘more fragments’’ flag in Handout 2, p. 8, line 57 of **ip.h**.

```
5$ cd ~mm64/public_html/x52.9547/src/fragment
6$ snoop -i fragment2.snoop | more
1   0.00000  192.168.0.5 -> cumulus.let.uu.nl UDP IP fragment ID=30504 Offset=0    MF=1 TOS=0x0 TT
2   0.00002  192.168.0.5 -> cumulus.let.uu.nl UDP IP fragment ID=30504 Offset=1480 MF=0 TOS=0x0 TT
3   0.20033 cumulus.let.uu.nl -> 192.168.0.5  ECHO R port=40225 AAAAAAAAAAAAAAAAAAAA
```

Packet 1

For the first time, the ‘‘more fragments’’ bit is on in line 24. Packets 1 and 2 have the same IP identification number (line 21).


```
1$ snoop -i fragment2.snoop -p 1,1 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 1 arrived at 15:25:52.97
     4  ETHER:  Packet size = 1514 bytes
     5  ETHER:  Destination = 0:10:4b:74:37:a,
     6  ETHER:  Source      = 0:d0:9:ff:57:5,
     7  ETHER:  Ethertype = 0800 (IP)
     8  ETHER:
     9  IP:   ----- IP Header -----
    10  IP:
    11  IP:   Version = 4
    12  IP:   Header length = 20 bytes
    13  IP:   Type of service = 0x00
    14  IP:         xxx. .... = 0 (precedence)
    15  IP:         ...0 .... = normal delay
    16  IP:         .... 0... = normal throughput
    17  IP:         .... .0.. = normal reliability
    18  IP:         .... ..0. = not ECN capable transport
    19  IP:         .... ...0 = no ECN congestion experienced
    20  IP:   Total length = 1500 bytes
    21  IP:   Identification = 30504
    22  IP:   Flags = 0x6
    23  IP:         .1.. .... = do not fragment
    24  IP:         ..1. .... = more fragments
    25  IP:   Fragment offset = 0 bytes
    26  IP:   Time to live = 255 seconds/hops
    27  IP:   Protocol = 17 (UDP)
    28  IP:   Header checksum = d83e
    29  IP:   Source address = 192.168.0.5, 192.168.0.5
    30  IP:   Destination address = 131.211.194.40, cumulus.let.uu.nl
    31  IP:   No options
    32  IP:
    33  UDP:  ----- UDP Header -----
    34  UDP:
    35  UDP:  Source port = 40225
    36  UDP:  Destination port = 7 (ECHO)
    37  UDP:  Length = 1481 (Not all data contained in this fragment)
    38  UDP:  Checksum = 73EE
    39  UDP:
    40  ECHO:  ----- ECHO:   -----
    41  ECHO:
    42  ECHO:  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    43  ECHO:
```

Packet 2

The ‘‘more fragments’’ bit in line 24 is back to zero. But for the first time, the fragmentation offset in line 25 is non-zero.

```
1$ snoop -i fragment2.snoop -p 2,2 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 2 arrived at 15:25:52.97
     4  ETHER:  Packet size = 35 bytes
     5  ETHER:  Destination = 0:10:4b:74:37:a,
     6  ETHER:  Source      = 0:d0:9:ff:57:5,
     7  ETHER:  Ethertype = 0800 (IP)
     8  ETHER:
     9  IP:   ----- IP Header -----
    10  IP:
    11  IP:   Version = 4
    12  IP:   Header length = 20 bytes
    13  IP:   Type of service = 0x00
    14  IP:         xxx. .... = 0 (precedence)
    15  IP:         ...0 .... = normal delay
    16  IP:         .... 0... = normal throughput
    17  IP:         .... .0.. = normal reliability
    18  IP:         .... ..0. = not ECN capable transport
    19  IP:         .... ...0 = no ECN congestion experienced
    20  IP:   Total length = 21 bytes
    21  IP:   Identification = 30504
    22  IP:   Flags = 0x4
    23  IP:         .1.. .... = do not fragment
    24  IP:         ..0. .... = last fragment
    25  IP:   Fragment offset = 1480 bytes
    26  IP:   Time to live = 255 seconds/hops
    27  IP:   Protocol = 17 (UDP)
    28  IP:   Header checksum = fd4c
    29  IP:   Source address = 192.168.0.5, 192.168.0.5
    30  IP:   Destination address = 131.211.194.40, cumulus.let.uu.nl
    31  IP:   No options
    32  IP:
    33  UDP:  [1 byte(s) of data, continuation of IP ident=30504]

2$ snoop -i fragment2.snoop -p 2,2 -x 34
  2   0.00000  192.168.0.5 -> cumulus.let.uu.nl UDP IP fragment ID=30504 Offset=148

       0: 41                                             A
```

Packet 3

```
1$ snoop -i fragment2.snoop -p 3,3 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 3 arrived at 15:25:53.17
     4  ETHER:  Packet size = 1070 bytes
     5  ETHER:  Destination = 0:d0:9:ff:57:5,
     6  ETHER:  Source      = 0:10:4b:74:37:a,
     7  ETHER:  Ethertype = 0800 (IP)
     8  ETHER:
     9  IP:   ----- IP Header -----
    10  IP:
    11  IP:   Version = 4
    12  IP:   Header length = 20 bytes
    13  IP:   Type of service = 0x00
    14  IP:         xxx. .... = 0 (precedence)
    15  IP:         ...0 .... = normal delay
    16  IP:         .... 0... = normal throughput
    17  IP:         .... .0.. = normal reliability
    18  IP:         .... ..0. = not ECN capable transport
    19  IP:         .... ...0 = no ECN congestion experienced
    20  IP:   Total length = 1052 bytes
    21  IP:   Identification = 38862
    22  IP:   Flags = 0x4
    23  IP:         .1.. .... = do not fragment
    24  IP:         ..0. .... = last fragment
    25  IP:   Fragment offset = 0 bytes
    26  IP:   Time to live = 242 seconds/hops
    27  IP:   Protocol = 17 (UDP)
    28  IP:   Header checksum = e658
    29  IP:   Source address = 131.211.194.40, cumulus.let.uu.nl
    30  IP:   Destination address = 192.168.0.5, 192.168.0.5
    31  IP:   No options
    32  IP:
    33  UDP:  ----- UDP Header -----
    34  UDP:
    35  UDP:  Source port = 7
    36  UDP:  Destination port = 40225
    37  UDP:  Length = 1032
    38  UDP:  Checksum = D189
    39  UDP:
    40  ECHO:  ----- ECHO:   -----
    41  ECHO:
    42  ECHO:  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    43  ECHO:
```

The "time to live" (TTL) field of an IP datagram

The TTL field was in Handout 2, p. 8, line 58 of **`ip.h`**. Here's the command that tells the default TTL for outgoing IPv4 datagrams. On i5.nyu.edu,

```
1$ /usr/sbin/ndd /dev/ip ip_def_ttl
open of /dev/ip failed: Permission denied
```

For **ndd**, *see Handout 2, p. 6.*

On the host where I'm the superuser,

```
2$ /usr/sbin/ndd /dev/ip '?' | more                    See all the parameters.
3$ /usr/sbin/ndd /dev/ip ip_def_ttl
255
```

To see the TTL of the IPv4 datagrams carrying UDP datagrams going out of a socket,

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/ttl/getsockopt.c

```
 1 #include <stdio.h>   /* for perror */
 2 #include <stdlib.h>
 3 #include <sys/types.h>
 4 #include <sys/socket.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     const int s = socket(AF_INET, SOCK_DGRAM, 0);
 9     int ttl;   /* time to live */
10     socklen_t length = sizeof ttl;
11
12     if (s < 0) {
13         perror(argv[0]);   /* Print message starting with argv[0]. */
14         return 1;
15     }
16
17     if (getsockopt(s, IPPROTO_IP, IP_TTL, &ttl, &length) != 0) {
18         perror(argv[0]);
19         return 2;
20     }
21
22     printf("The time to live is %d.\n", ttl);
23     return EXIT_SUCCESS;
24 }
```

```
4$ gcc -o ~/bin/getsockopt ~mm64/public_html/x52.9547/src/ttl/getsockopt.c \
    -lnsl -lsocket
5$ ls -l ~/bin/getsockopt

6$ getsockopt
The time to live is 255.

7$ echo $?
0
```

The Perl library module **Socket** in line 2 already has equivalents for the above C macros **AF_INET** and **SOCK_DGRAM**. We had to create our own equivalents for **IPPROTO_IP** and **IP_TTL** in lines 5−6.

The function **getsockopt** in line 9 returns a block of memory containing an unsigned integer. To extract the integer, we need our old friend **unpack** in line 10. The uppercase **'I'** in line 10 stands for "unsigned integer". See the **unpack**'s in Handout 1, p. 17, line 11 of **class.pl**; Handout 1, p. 26, line 20 of **localhosts.pl**.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/ttl/getsockopt.pl

```
1 #!/bin/perl
```

```
 2 use Socket;
 3
 4 #macros in /usr/include/netinet/in.h
 5 sub IPPROTO_IP {return 0;}   #IPv4
 6 sub IP_TTL     {return 4;}   #IPv4 time to live
 7
 8 socket(S, AF_INET, SOCK_DGRAM, 0) or die "$0: $!";
 9 $ttl = getsockopt(S, IPPROTO_IP, IP_TTL) or die "$0: $!";
10 $t = unpack('I', $ttl);
11 print "The time to live is $t.\n";
12 exit 0;
```

```
8$ ~mm64/public_html/x52.9547/src/ttl/getsockopt.pl
The time to live is 255.
```

▼ **Homework 3.3: is the TCP time to live different?**

To create a socket for sending and receiving IPv4 datagrams that carry TCP segments, change **SOCK_DGRAM** to **SOCK_STREAM** in the above programs. Does this change the time to live?
▲

Send out a prematurely aged packet.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/ttl/setsockopt.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/types.h>
 4 #include <sys/socket.h>
 5 #include <netinet/in.h>   /* for inet_ntop */
 6 #include <arpa/inet.h>
 7 #include <netdb.h>        /* for getservbyname */
 8 #include <signal.h>
 9
10 const char *progname;
11 void handle_sigalrm(int sig);
12
13 int main(int argc, char **argv)
14 {
15     const int s = socket(AF_INET, SOCK_DGRAM, 0);
16     int ttl = 10;   /* time to live */
17     int length = sizeof ttl;
18     const struct servent *service;
19     char name[1000] = "aixmita1.urz.uni-heidelberg.de";
20     const struct hostent *host;
21     struct sockaddr_in address;
22     socklen_t socklen = sizeof address;
23     ssize_t n;
24     char buffer[100];
25     int len;
26
27     progname = argv[0];
28     if (s < 0) {
29         perror(progname);
30         return 1;
```

```
31      }
32
33      if (setsockopt(s, IPPROTO_IP, IP_TTL, &ttl, sizeof ttl) != 0) {
34          perror(progname);
35          return 2;
36      }
37
38      if (getsockopt(s, IPPROTO_IP, IP_TTL, &ttl, &length) != 0) {
39          perror(progname);
40          return 3;
41      }
42
43      printf("The time to live was set to %d.\n", ttl);
44
45      service = getservbyname("daytime", "udp");
46      if (service == NULL) {
47          fprintf(stderr, "%s: couldn't find udp service daytime\n",
48              progname);
49          return 4;
50      }
51
52      host = gethostbyname(name);
53      if (host == NULL) {
54          fprintf(stderr, "%s: couldn't find \"%s\", h_errno == %d\n",
55              progname, name, h_errno);
56          return 5;
57      }
58
59      bzero(&address, sizeof address);
60      address.sin_family = AF_INET;
61      address.sin_port = service->s_port;
62      address.sin_addr.s_addr = *(in_addr_t *)host->h_addr_list[0];
63
64      /* Send a datagram. */
65      if (sendto(s, NULL, 0, 0,
66          (const struct sockaddr *)&address, sizeof address) != 0) {
67          perror(progname);
68          return 6;
69      }
70
71      if (getsockname(s,  (struct sockaddr *)&address, &socklen) != 0) {
72          perror(progname);
73          return 7;
74      }
75
76      printf("I sent an empty UDP datagram from my port %d.\n",
77              ntohs(address.sin_port));
78      fflush(stdout);
79
80      if (signal(SIGALRM, handle_sigalrm) == SIG_ERR) {
81          perror(progname);
82          return 8;
83      }
84
```

```
 85     alarm(15);
 86
 87     length = sizeof address;
 88     n = recvfrom(s, buffer, sizeof buffer, 0, (struct sockaddr *)&address,
 89         &length);
 90
 91     if (n < 0) {
 92         perror(progname);
 93         return 9;
 94     }
 95
 96     if (inet_ntop(AF_INET, &address.sin_addr, name, sizeof name) == NULL) {
 97         perror(progname);
 98         return 10;
 99     }
100
101     printf("I then received a datagram containing the following %d bytes\n"
102         "from port %d of %s", n, ntohs(address.sin_port), name);
103
104     host = gethostbyaddr((const char *)&address.sin_addr,
105         sizeof address.sin_addr, AF_INET);
106     if (host != NULL) {
107         printf(" (%s)", host->h_name);
108     }
109
110     printf("\n%s\n", buffer);
111     return EXIT_SUCCESS;
112 }
113
114 void handle_sigalrm(int sig)
115 {
116     if (sig != SIGALRM) {
117         fprintf(stderr, "%s: signal handler called with signal %d "
118             "instead of %d\n", progname, sig, SIGALRM);
119         exit(11);
120     }
121
122     fprintf(stderr, "%s: The datagram I was waiting for "
123         "didn't arrive in 15 seconds.\n", progname);
124     exit(12);
125 }
```

```
1$ cd ~mm64/public_html/x52.9547/src/ttl/setsockopt.c
2$ gcc -o ~/bin/setsockopt setsockopt.c -lsocket -lnsl
3$ ls -l ~/bin/setsockopt

4$ setsockopt
The time to live was set to 10.
I sent an empty UDP datagram from my port 34840.
setsockopt: The datagram I was waiting for didn't arrive in 15 seconds.

5$ echo $?
12
```

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/ttl/setsockopt.pl

```
 1 #!/bin/perl
 2 #Send a UDP datagram to the daytime server on another host.
 3 #Display the UDP datagram that comes back.
 4 use Socket;
 5
 6 socket(S, AF_INET, SOCK_DGRAM, 0) or die "$0: $!";
 7
 8 sub IPPROTO_IP {return 0;}   #IPv4
 9 sub IP_TTL     {return 4;}   #IPv4 time to live
10
11 setsockopt(S, IPPROTO_IP, IP_TTL, 10) or die "$0, $!";
12 $ttl = getsockopt(S, IPPROTO_IP, IP_TTL) or die "$0, $!";
13 $t = unpack('I', $ttl);
14 print "The time to live was set to $t.\n";
15
16 $port = getservbyname('daytime', 'udp')
17     or die "$0: couldn't find udp service daytime";
18
19 $name = 'aixmita1.urz.uni-heidelberg.de';
20 $ip = inet_aton($name) or die "$0: couldn't get IP address for $name";
21
22 $address = sockaddr_in($port, $ip);
23 $n = send(S, '', 0, $address);
24 defined $n or die "$0: $!";
25 if ($n != 0) {
26     die "$0: sent $n bytes instead of 0 bytes\n";
27 }
28
29 $address = getsockname(S) or die "$0: $!";   #Can't do this before the send.
30 ($port, $ip) = sockaddr_in($address);
31 print "I sent an empty UDP datagram from my port $port.\n";
32
33 $SIG{ALRM} = 'handle_sigalrm';
34 alarm(15);
35
36 $address = recv(S, $buffer, 100, 0) or die "$0: $!";
37 ($port, $ip) = sockaddr_in($address);
38
39 print "I then received a datagram containing the following ", length($buffer),
40     " bytes\nfrom port $port of ", inet_ntoa($ip);
41
42 $name = gethostbyaddr($ip, AF_INET);
43 if (defined $name) {
44     print " ($name)";
45 }
46
47 print "\n$buffer\n";
48 exit 0;
49
50 sub handle_sigalrm {
51     if ($_[0] ne 'ALRM') {
52         die "$0: received signal $_[0] instead of ALRM";
```



```
53      }
54      die "$0: The datagram I was waiting for didn't arrive in 15 seconds";
55  }
```

```
6$ snoop -o setsockopt.snoop \
    host aixmita1.urz.uni-heidelberg.de or icmp &
[1] 12345

7$ setsockopt.pl
The time to live was set to 10.
I sent an empty UDP datagram from my port 61434.
setsockopt.pl: The datagram I was waiting for didn't arrive in 15 seconds.

8$ echo $?                          The Korn shell variable $? holds the exit status.
255                                 The exit status 255 came from the die in line 54.

9$ kill 12345
```

Packet 1

The TTL in line 24 is 10:

```
1$ cd ~mm64/public_html/x52.9547/src/ttl
2$ snoop -i setsockopt.snoop -p 1,1 -ta -v | cat -n | more
```

```
 1  ETHER:  ----- Ether Header -----
 2  ETHER:
 3  ETHER:  Packet 1 arrived at 16:21:42.08
 4  ETHER:  Packet size = 42 bytes
 5  ETHER:  Destination = 0:10:4b:74:37:a,
 6  ETHER:  Source      = 0:d0:9:ff:57:5,
 7  ETHER:  Ethertype = 0800 (IP)
 8  ETHER:
 9  IP:   ----- IP Header -----
10  IP:
11  IP:   Version = 4
12  IP:   Header length = 20 bytes
13  IP:   Type of service = 0x00
14  IP:         xxx. .... = 0 (precedence)
15  IP:         ...0 .... = normal delay
16  IP:         .... 0... = normal throughput
17  IP:         .... .0.. = normal reliability
18  IP:         .... ..0. = not ECN capable transport
19  IP:         .... ...0 = no ECN congestion experienced
20  IP:   Total length = 28 bytes
21  IP:   Identification = 56690
22  IP:   Flags = 0x4
23  IP:         .1.. .... = do not fragment
24  IP:         ..0. .... = last fragment
25  IP:   Fragment offset = 0 bytes
26  IP:   Time to live = 10 seconds/hops
27  IP:   Protocol = 17 (UDP)
28  IP:   Header checksum = 7642
29  IP:   Source address = 192.168.0.5, 192.168.0.5
30  IP:   Destination address = 129.206.218.160, aixmita1.urz.uni-heidelberg.de
31  IP:   No options
32  IP:
33  UDP:  ----- UDP Header -----
34  UDP:
35  UDP:  Source port = 61434
36  UDP:  Destination port = 13 (DAYTIME)
37  UDP:  Length = 8
38  UDP:  Checksum = F2B9
39  UDP:
40  DAYTIME:  ----- DAYTIME:   -----
41  DAYTIME:
42  DAYTIME:  ""
43  DAYTIME:
44
```

To see the first 28 bytes of the IP datagram, we must first chop off the 14-byte Ethernet header. Byte 0 (the first byte), **0x45**, is the IP version number in line 11 and the header length (measured in 4-byte words) in line 12. Byte 8 (the ninth byte), **0x0a**, is the time to live in line 26. Bytes 10−11 , **7642**, is the checksum.

```
3$ snoop -i setsockopt.snoop -p 1,1 -x 14,28 | tail +3
       0: 4500 001c dd72 4000 0a11 7642 c0a8 0005    E....r@...vB....
      16: 81ce daa0 effa 000d 0008 f2b9              ............
```

Packet 2

I was expecting a UDP answer from port 13 (**daytime**) on the host **aixmita1.urz.uni-heidelberg.de**. Instead, a host I've never heard of (line 29), somewhere between here and Germany, sent me an ICMP packet carrying the **ICMP_TIMXCEED** error code from line 157 of the **ip_icmp.h** in Handout 2, p. 9.

```
1$ snoop -i setsockopt.snoop -p 2,2 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 2 arrived at 16:21:42.24
     4  ETHER:  Packet size = 74 bytes
     5  ETHER:  Destination = 0:d0:9:ff:57:5,
     6  ETHER:  Source      = 0:10:4b:74:37:a,
     7  ETHER:  Ethertype = 0800 (IP)
     8  ETHER:
     9  IP:   ----- IP Header -----
    10  IP:
    11  IP:   Version = 4
    12  IP:   Header length = 20 bytes
    13  IP:   Type of service = 0x00
    14  IP:         xxx. .... = 0 (precedence)
    15  IP:         ...0 .... = normal delay
    16  IP:         .... 0... = normal throughput
    17  IP:         .... .0.. = normal reliability
    18  IP:         .... ..0. = not ECN capable transport
    19  IP:         .... ...0 = no ECN congestion experienced
    20  IP:   Total length = 56 bytes
    21  IP:   Identification = 0
    22  IP:   Flags = 0x0
    23  IP:         .0.. .... = may fragment
    24  IP:         ..0. .... = last fragment
    25  IP:   Fragment offset = 0 bytes
    26  IP:   Time to live = 247 seconds/hops
    27  IP:   Protocol = 1 (ICMP)
    28  IP:   Header checksum = b578
    29  IP:   Source address = 216.200.116.214, pos2-0.pr1.fra1.de.above.net
    30  IP:   Destination address = 192.168.0.5, 192.168.0.5
    31  IP:   No options
    32  IP:
    33  ICMP:  ----- ICMP Header -----
    34  ICMP:
    35  ICMP:  Type = 11 (Time exceeded)
    36  ICMP:  Code = 0 (in transit)
    37  ICMP:  Checksum = e0fd
    38  ICMP:
    39
```

The ICMP message tells us even more. Lines 12 and 20 tell us that the cargo of the IP datagram is an ICMP message of 36 bytes. The header of the ICMP message is eight bytes. (The first four bytes are in lines 35−37; the remaining four are 0.) This leaves 28 bytes for cargo. The cargo is the 20-byte header and the first 8 bytes of cargo of the IP datagram that caused the ICMP message. To see these 28 bytes in hex, we must first chop off the first 14 + 20 + 8 = 42 bytes. The time to live in byte 8 of the IP header has counted down to **0x01**. (Incidentally, this causes the checksum **7f42** in bytes 10−11 to differ from the original checksum.)


```
2$ snoop -i setsockopt.snoop -p 2,2 -x 42,28 | tail +3
       0: 4500 001c dd72 4000 0111 7f42 c0a8 0005    E....r@....B....
      16: 81ce daa0 effa 000d 0008 23f2              .....ú....#.
```

traceroute

`traceroute` sends out three UDP datagrams in three prematurely aged IP datagrams with a TTL of 1. They get no farther than one hop. The router in which they expire sends us back three ICMP packets carrying the ‘‘time exceeded’’ error code.

Then **`traceroute`** sends out three more datagrams, this time with a TTL of 2. They get a little farther, and again the router in which they expire sends us back a trio of error messages.

Unfortunately, some routers don’t send us back ICMP errors when a datagram that’s passing through expires. In this case, **`traceroute`** prints asterisks.

`ftcg5faculty2.edlab.its.nyu.edu` is a Mac OSX at the Ed Site computer center. On i5.nyu.edu, I said

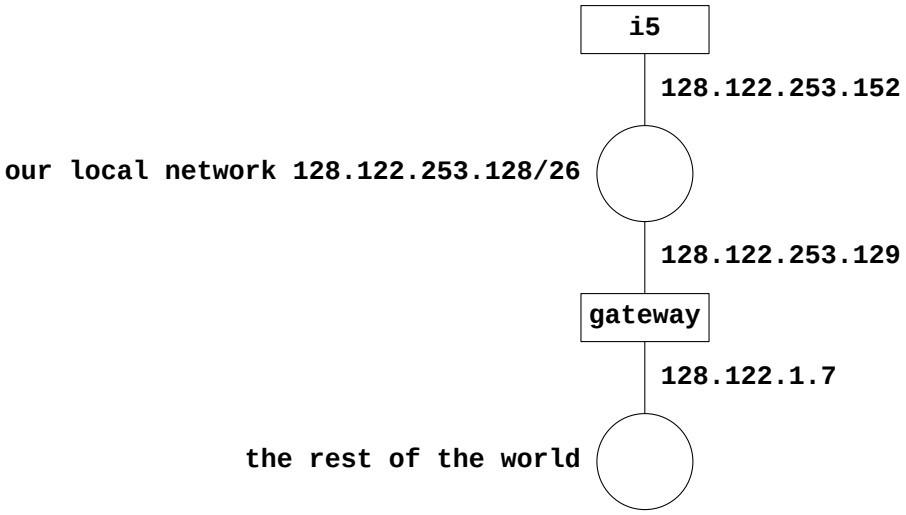
```
1$ traceroute ftcg5faculty2.edlab.its.nyu.edu
traceroute to ftcg5faculty2.edlab.its.nyu.edu (192.168.20.196), 30 hops max, 40 byte pa
 1  WWITSGW-VLAN-13.NET.NYU.EDU (128.122.253.129)  0.536 ms  0.367 ms  0.444 ms
 2  COREGWD-WP-GEC-4.NET.NYU.EDU (128.122.1.34)  0.412 ms  0.353 ms  0.345 ms
 3  GODDARDGW-FE-6-0-1.NET.NYU.EDU (128.122.1.104)  0.564 ms  0.471 ms  0.462 ms
 4  FTCG5FACULTY2.EDLAB.ITS.NYU.EDU (192.168.20.196)  0.766 ms  0.581 ms  0.426 ms
```

`128.122.253.129` is the first host on our local network; we saw its IP address in Handout 1, p. 21. All traffic leaving our local network goes out through this host: it’s our gateway (i.e., router) to the outside world.

On **`ftcg5faculty2.edlab.its.nyu.edu`**, I traced the route back to i5.nyu.edu :

```
2$ traceroute i5.nyu.edu
traceroute to i5.nyu.edu (128.122.253.152), 30 hops max, 40 byte packets
 1  goddardgw-ether-1-2.nyu.net (192.168.20.1)  2.914 ms  0.443 ms  0.321 ms
 2  coregwd-wp-vlan904.net.nyu.edu (128.122.1.97)  0.494 ms  0.463 ms  0.384 ms
 3  wwitsgw-ge-3-1.net.nyu.edu (128.122.1.7)  0.647 ms  0.478 ms  0.482 ms
 4  i5.nyu.edu (128.122.253.152)  0.611 ms  0.591 ms  0.573 ms
```

If our local area network has only one gateway, **`128.122.1.7`** must be the IP address of the other network interface of that gateway on the side farther from us.



▼ Homework 3.4: prove that `128.122.253.129` and `128.122.1.7` are the same machine

Where do you go if you try to trace the route from **`i5.nyu.edu`** to **`128.122.1.7`**?

▲

The Point-to-Point Protocol PPP

For the Point to Point Protocol, see pp. 150−169, 486−502; RFC's 1661−1663. A frame of PPP can carry any of the following:

(1) an LCP (''Link Control Protocol'') packet

(2) a NCP (''Network Control Protocol'') packet (for example, an IPCP (''IP Control Protocol'') packet)

(3) an IPv4 datagram

(4) an IPv6 datagram.

Configure PPP on Caldera Linux

(1) To see if the kernel supports PPP,

```
1$ lsmod | awk 'NR == 1 || $1 ~ /ppp/'          "list loaded modules"
Module                  Size  Used by
ppp                    18956   2
```

If not, we would have had to run

```
2$ ls -l /lib/modules/2.2.5/net/ppp.o
3$ inmod ppp                                    "install loadable kernel module"
4$ lsmod | more
```

(2) Even though p. 208 says that an **`/etc/resolv.conf`** file should never have both a **`domain`** and a **`search`** entry, i5.nyu.edu has

```
5$ cat -n /etc/resolv.conf
     1  domain nyu.edu
     2  nameserver 128.122.253.37
     3  nameserver 128.122.128.24
     4  nameserver 128.122.253.92
     5  search nyu.edu home.nyu.edu es.its.nyu.edu
```

In my Linux **`/etc/resolv.conf`** file I therefore wrote


```
1 #Copied from the i5.nyu.edu /etc/resolv.conf
2 domain nyu.edu
3
4 nameserver 128.122.253.37
5 nameserver 128.122.128.24
6 nameserver 128.122.253.92
```

(3) The **/etc/hosts** file on i5.nyu.edu contains

```
6$ cat -n /etc/hosts
     1  #
     2  # Internet host table
     3  #
     4  127.0.0.1   localhost
     5  ## 128.122.253.152   i6.home.nyu.edu  loghost
     6  128.122.253.152  i5.nyu.edu   i5 loghost
```

In my Linux **/etc/hosts** file I wrote

```
1 127.0.0.1 noname.nodomain.nowhere noname localhost
2
3 #My first IP interface
4 0.0.0.0 noname.nodomain.nowhere
```

I wrote the dummy number **0.0.0.0** because I'll get an IP address when I connect to another machine via PPP. See the **noipdefault** option of the following **pppd**.

(4) My internal modem is known as a ''winmodem'' because it works only with Windows. I therefore had to plug an external modem into the serial port **COM1**. In Linux, this port is called **/dev/ttyS0**. I then created a symbolic link named **/dev/modem**. A symbolic link in Unix is like a shortcut in Windows or an alias in Macintosh.

```
7$ cd /dev                              on my Linux box
8$ pwd

9$ ln -s ttyS0 modem                    uppercase S zero
10$ ls -l ttyS0 modem
lrwxrwxrwx   1 root     root            5 Oct 13 14:23 modem -> ttyS0
crw-rw-rw-   1 root     root       3,  48 Apr  3  1999 ttyS0
```

(5) The settings of the modem's eight dip switches are

(1) Data Terminal Ready normal

(2) Verbal result codes

(3) Display result codes

(4) Do not echo offline commands

(5) Auto answer off

(6) Carrier detect normal

(7) Load NVRAM defaults (" Non-Volatile Random Access Memory'')

(8) Dumb mode

(6) We will need to give many options to the PPP dæmon **pppd**. Instead of writing them as command line arguments, you can write them in a file called **/etc/ppp/options**. See pp. 154, 486.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/options

```
1 ###
2 # /etc/ppp/options - options for pppd
```



```
 3 #
 4 #See TCP/IP Network Administration pp. 486-496.
 5
 6 #-V sends the verbose output to the stderr
 7 connect '/usr/sbin/chat -f /etc/ppp/chat-script -T 1-212-253-4698 -V'
 8
 9 #/dev/modem is the symbolic link I created to the external modem /dev/ttyS0
10 /dev/modem
11
12 #Speed
13 115200
14
15 #Use the modem control lines:
16 #Wait for the DCD "carrier detect" signal before opening serial device;
17 #drop the DTR "data terminal ready" signal when terminating a connection.
18 modem
19
20 #Let the modem's serial port use hardware flow control (the RTS "request to
21 #send" and CTS "clear to send" pins) to indicate when it's overwhelmed with
22 #data.  The alternative would be to send special XON and XOFF characters in the
23 #data.
24 crtscts
25
26 #Tell pppd to insert the PPP link as the default route in the routing table,
27 #if there is no default route yet.
28 defaultroute
29
30 #Let the remote system determine our IP address.
31 noipdefault
32
33 user mm64
```

(7) Put the following line in the ''Password Authentication Protocol'' file. **/etc/ppp/pap-secrets**. (There could also be a ''Challenge Handshake Authentication Protocol'' file **/etc/ppp/chap-secrets**.) The second field is the hostname of the server to which we are connecting; the wildcard ***** means ''any server''.

```
your_loginname * your_secret_password
```

(8) Line 7 of the above **/etc/ppp/options** tells **pppd** to run a program named **chat**. **chat** will execute the following script, named **/etc/ppp/chat-script**. You need only the first 9 lines. I'm expecting the **BIS** in line 11 only because I want **chat** to echo the entire **CONNECT** string ending with **V42BIS**. The string **\c** is the null string. If we had tried to write the null string as **''**, it would have been a string consisting of a carriage return.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9547/src/chat-script

```
1 ABORT   BUSY
2 ABORT   "NO DIAL TONE"
3 REPORT  CONNECT
4
5 SAY "\nChat is about to dial.\n"
6
7 ""  ATZ
8 OK  ATDT\T
9 CONNECT \c
```



```
10
11 BIS \c
12 SAY \n
```

(9) When PPP comes up, the script **/etc/ppp/ip-up** is automatically executed. The parentheses redirect the standard output of all of the enclosed programs to the terminal **/dev/ttyp0**.

```
 1 #!/bin/bash
 2 (
 3     echo
 4     echo PPP is up, device $1, tty $2, speed $3, local $4, remote $5
 5     echo
 6     ps -Af | awk 'NR == 1 || $NF == "pppd"'
 7     echo
 8     ifconfig ppp0   #See the ppp0 network interface.
 9     netstat -i  #another way to see the network interfaces
10     netstat -r  #see the routing table as names
11     netstat -nr  #see the routing table as IP addresses
12 ) > /dev/ttyp0
13
14 exit 0
```

(10) To disconnect PPP, just use **ps** to find the PID number of the **pppd** dæmon and **kill** it. When PPP goes down, the script **/etc/ppp/ip-down** is automatically executed.

```
1 #!/bin/bash
2 ifconfig ppp0 down
3 echo PPP is down, device $1, tty $2, speed $3, local $4, remote $5 > /dev/ttyp0
4 exit 0
```

(11) To establish a PPP connection, run the dæmon manually or from your **.profile** file.

```
11$ pppd &
[12345]
Chat is about to dial.
ATZ                                  chat outputs these lines because it's in verbose mode.
OK
ATDT1-212-253-4698
CONNECT 53333/ARQ/V90/LAPM/V42BIS
PPP is up, interface ppp0, tty /dev/modem, speed 115200
 local 216.165.3.155
remote 216.165.0.18

    UID        PID  PPID  C STIME TTY          TIME CMD
    root     12345   940  0 07:59 ttyp0    00:00:00 pppd

    ppp0      Link encap:Point-to-Point Protocol
              inet addr:216.165.3.155  P-t-P:216.165.0.18  Mask:255.255.255.255
              UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
              RX packets:7 errors:1 dropped:0 overruns:0 frame:1
              TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:10
```

```
Kernel Interface table
Iface   MTU Met  RX-OK RX-ERR RX-DRP RX-OVR  TX-OK TX-ERR TX-DRP TX-OVR Flags
lo     3924   0     58      0      0      0     58      0      0      0 LRU
eth0   1500   0      0      0      0      0      0      0      0      0 B
ppp0   1500   0      7      1      0      0      9      0      0      0 OPRU

Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
DIALN.NET.NYU.E *               255.255.255.255 UH        0 0          0 ppp0
127.0.0.0       *               255.0.0.0       U         0 0          0 lo
default         DIALN.NET.NYU.E 0.0.0.0         UG        0 0          0 ppp0

Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
216.165.0.18    0.0.0.0         255.255.255.255 UH        0 0          0 ppp0
127.0.0.0       0.0.0.0         255.0.0.0       U         0 0          0 lo
0.0.0.0         216.165.0.18    0.0.0.0         UG        0 0          0 ppp0
```

pppd always adds a route in the routing table to the host we dialed up to. Because of the **defaultroute** option of **pppd**, it also added the default route.

Download and compile ssh (the "Secure Shell")

To get the **ssh** source code for Linux, the website

http://www.nyu.edu/its/faq/connecting/ssh.html

told me to go to

ftp://ftp.funet.fi/pub/unix/security/login/ssh/

On my Linux box, I said

```
1$ cd /opt                  The files we get will go into the current directory on my Linux box.
2$ ftp ftp.funet.fi
Name: anonymous
Password: any_password_will_work
Using binary mode to transfer files.
ftp> cd pub/unix/security/login/ssh
ftp> pwd
ftp> dir
ftp> dir .                          does same thing; dot is current directory on ftp.funet.fi
ftp> dir . filename                 Create file in current directory on Linux box.
ftp> dir . "| more"

ftp> get README.SSH2 "| more"       see on screen
ftp> get README.SSH2                copy into your current directory

ftp> get ssh-2.2.0.tar.gz           .tar.gz file with newest version number
ftp> quit
3$ ls -l ssh-2.2.0.tar.gz
-rw-r--r--   1 root     root     1736098 June 10 06:38 ssh-2.2.0.tar.gz
```

Then I followed the directions in the **README.SSH2** file:


```
4$ zcat ssh-2.2.0.tar.gz | tar tvf - | more          verbose table of contents
5$ zcat ssh-2.2.0.tar.gz | tar xf -                  Extract all the files.
6$ ls -ld ssh-2.2.0

7$ cd ssh-2.2.0
8$ pwd

9$ ./configure --enable-debug                        Dot is current directory.
10$ make
```

Then become the superuser and say:

```
11$ make install

12$ cd apps/ssh
13$ pwd
14$ make clean-up-old

15$ /usr/local/bin/ssh mm64@i5.nyu.edu
```

□

