Fall 2005 Handout 1

32-bit signed and unsigned integer values

(signed)			
`int`	*bit pattern*	**`unsigned`**	*bit pattern*
		4294967295	**11111111111111111111111111111111**
		4294967294	**11111111111111111111111111111110**
		4294967293	**11111111111111111111111111111101**
		4294967292	**11111111111111111111111111111100**
		4294967291	**11111111111111111111111111111011**
		4294967290	**11111111111111111111111111111010**
		2147483653	**10000000000000000000000000000101**
		2147483652	**10000000000000000000000000000100**
		2147483651	**10000000000000000000000000000011**
		2147483650	**10000000000000000000000000000010**
		2147483649	**10000000000000000000000000000001**
		2147483648	**10000000000000000000000000000000**
2147483647	**01111111111111111111111111111111**	2147483647	**01111111111111111111111111111111**
2147483646	**01111111111111111111111111111110**	2147483646	**01111111111111111111111111111110**
2147483645	**01111111111111111111111111111101**	2147483645	**01111111111111111111111111111101**
2147483644	**01111111111111111111111111111100**	2147483644	**01111111111111111111111111111100**
2147483643	**01111111111111111111111111111011**	2147483643	**01111111111111111111111111111011**
2147483642	**01111111111111111111111111111010**	2147483642	**01111111111111111111111111111010**
5	**00000000000000000000000000000101**	5	**00000000000000000000000000000101**
4	**00000000000000000000000000000100**	4	**00000000000000000000000000000100**
3	**00000000000000000000000000000011**	3	**00000000000000000000000000000011**
2	**00000000000000000000000000000010**	2	**00000000000000000000000000000010**
1	**00000000000000000000000000000001**	1	**00000000000000000000000000000001**
0	**00000000000000000000000000000000**	0	**00000000000000000000000000000000**
−1	**11111111111111111111111111111111**		
−2	**11111111111111111111111111111110**		
−3	**11111111111111111111111111111101**		
−4	**11111111111111111111111111111100**		
−5	**11111111111111111111111111111011**		
−6	**11111111111111111111111111111010**		
−2147483643	**10000000000000000000000000000101**		
−2147483644	**10000000000000000000000000000100**		
−2147483645	**10000000000000000000000000000011**		
−2147483646	**10000000000000000000000000000010**		
−2147483647	**10000000000000000000000000000001**		
−2147483648	**10000000000000000000000000000000**		

The bits of an integer are numbered from right to left, starting at 0. In a 32-bit integer, the leftmost bit is therefore bit 31. Bit 0 is called the *lowest* bit or the *low order* bit. Bit 31 is called the *top* bit or the *high order* bit.

Bit 0 is the *one's place,* bit 1 is the *two's place,* bit 2 is the *four's place,* etc. In signed values, the top bit is called the *sign bit.* It is 1 for negative numbers, 0 otherwise.

Why is the integer −1 represented as 11111111111111111111111111111111 (thirty-two 1's)?

Think of a car odometer running backward:

```
0003
0002
0001
0000        zero
9999        negative one
9998        negative two
```

1111 is the binary equivalent of 9999.

If you want zero to be **00000000000000000000000000000000**, positive one to be **00000000000000000000000000000001**, and positive one plus negative one to be 0, then the definition of negative one is forced on you:

```
  00000000000000000000000000000001   positive one
+ 11111111111111111111111111111111   negative one
-------------------------------------------------
  00000000000000000000000000000000   zero

  00000000000000000000000000000010   positive two
+ 11111111111111111111111111111110   negative two
-------------------------------------------------
  00000000000000000000000000000000   zero

  00000000000000000000000000000011   positive three
+ 11111111111111111111111111111101   negative three
---------------------------------------------------
  00000000000000000000000000000000   zero

  00000000000000000000000000000100   positive four
+ 11111111111111111111111111111100   negative four
--------------------------------------------------
  00000000000000000000000000000000   zero
```

Convert binary to hexadecimal and octal

Each hexadecimal digit is an abbreviation for a group of four consecutive binary digits. If the hexadecimal digit is a letter, you can write it in either uppercase or lowercase.

binary	*hex*
0000	**0**
0001	**1**
0010	**2**
0011	**3**
0100	**4**
0101	**5**
0110	**6**
0111	**7**
1000	**8**
1001	**9**
1010	**A**
1011	**B**
1100	**C**
1101	**D**
1110	**E**
1111	**F**

Each octal digit is an abbreviation for a group of three consecutive binary digits:

binary	*octal*
000	**0**
001	**1**
010	**2**
011	**3**
100	**4**
101	**5**
110	**6**
111	**7**

One binary digit is called a *bit.* Four binary digits are called a *nibble.* Eight binary digits should be called an *octet,* but most people call them a *byte.*

(1) To convert binary to hexadecimal, start at the right end and divide the binary number into nibbles. For example, **807AFD98** is the IP address of our host i5.nyu.edu: 128.122.253.152.

```
10000000011110101111110110011000           32 bits
1000 0000 0111 1010 1111 1101 1001 1000    8 nibbles
  8    0    7    A    F    D    9    8     8 hex digits
```

If the number of bits is not a multiple of four, add **0**'s to the left end of the binary number:

```
   111111111        9 bits
000111111111        12 bits
0001 1111 1111      3 nibbles
  1    F    F       3 hex digits
```

(2) To convert binary to octal, start at the right end and divide the binary number into groups of three bits:

```
111101101           9 bits
111 101 101         3 trios
 7   5   5          3 octal digits
```

If the number of bits is not a multiple of three, add **0**'s to the left end of the binary number:

```
1111101101              10 bits
001111101101            12 bits
001 111 101 101         4 trios
 1   7   5   5
```

Convert using bc

Convert binary to hex:

```
1$ bc                   “binary calculator”
obase = 16              Spaces optional.  Specify the output base before the input base.
ibase = 2
10000000011110101111110110011000            IPv4 address of our host i5.nyu.edu
807AFD98                It types this.
control-d
2$
```

Convert hex to binary:

```
3$ bc
obase = 2               Specify the output base before the input base.
ibase = 16
807AFD98                Must type hex in uppercase for bc.
10000000011110101111110110011000    It types this.
control-d
4$
```

Switch horses in midstream: convert binary to hex, and then hex to binary.

```
5$ bc
obase = 16              Specify the output base before the input base.
ibase = 2
10000000011110101111110110011000
807AFD98                It types this.
ibase=10000             Had to specify the input base in binary.
obase=2
807AFD98                Must type hex in uppercase for bc.
10000000011110101111110110011000    It types this.
control-d
6$

7$ man bc                           or visit http://i5.nyu.edu/˜mm64/man/
```

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

```
#!/bin/ksh
#Convert one number from hex to binary.
#Usage: hextobin 807AFD98

if [[ $# -ne 1 ]]               #for $#, see ksh(1) p. 11
then
    echo $0: requires one argument 1>&2   #for 1>&2, see ksh(1) p. 18
    exit 1
fi

if [[ -z $1 ]]                  #for -z, see ksh(1) p. 16
```

```
then
    echo $0: argument can\’t be empty string 1>&2
    exit 2
fi

if [[ $1 == *[!0-9A-Fa-f]* ]]   #exclamation means not: ksh(1) p. 14
then
    echo $0: argument must be hex number 1>&2
    exit 3
fi

(
    echo obase = 2
    echo ibase = 16
    echo $1 | tr ’[a-f]’ ’[A-F]’
) | bc

exit 0
```

```
8$ hextobin 807afd98                can type uppercase or lowercase
10000000011110101111110110011000
```

How many...

There are $2^1 = 2$ one-bit binary numbers.
There are $2^2 = 2 \times 2 = 4$ two-bit binary numbers.
There are $2^3 = 2 \times 2 \times 2 = 8$ three-bit binary numbers.
There are $2^4 = 2 \times 2 \times 2 \times 2 = 16$ four-bit binary numbers. Et cetera:

```
0                  00                 000                0000
1                  01                 001                0001
                   10                 010                0010
                   11                 011                0011
                                      100                0100
                                      101                0101
                                      110                0110
                                      111                0111
                                                         1000
                                                         1001
                                                         1010
                                                         1011
                                                         1100
                                                         1101
                                                         1110
                                                         1111
```

(1) We’ll cover TCP and UDP port numbers shortly. In the meantime, let’s find out how many of them there are. Each one is 16 bits (two bytes), so there are $2^{16} = 65,536$ possible port numbers:

```
1$ bc
2 ˆ 16                  You type this.  Caret means exponentiation; spaces optional.
65536                   It types this.
control-d
2$
```

(2) How many possible 13-bit IP version 4 fragmentation offset values are there? Each one is 13 bits. We therefore have $2^{13} = 8,192$ possible offsets. See the fragmentation offset field of the IP version 4 header in pp. 14, 16, 679, 681.

(3) How many possible 24-bit Ethernet OUI's ("Organizationally Unique Identifiers") are there? Each one is 24 bits (six bytes). We therefore have $2^{24} = 16,777,216$ possible OIU's.

(4) How many IP version 4 addresses are there? Each one is 32 bits (four bytes). We therefore have $2^{32} = 4,294,967,296$ possible IP version 4 addresses.

(5) How many Ethernet addresses are there? Each one is 48 bits (six bytes). We therefore have $2^{48} = 281,474,976,710,656$ possible Ethernet addresses. (An Ethernet address is an example of a Medium Access Control (MAC) address. See p. 136.)

(6) How many IP version 6 addresses are there? Each one is 128 bits (sixteen bytes). We therefore have $2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,456$ possible IP version 6 addresses. This is a 39-digit number:

```
3$ bc                                     interactive
2 ^ 128
340282366920938463463374607431768211456
control-d
4$

5$ echo '2 ^ 128' | bc                    non-interactive
340282366920938463463374607431768211456
```

How many digits was that? The **tr** will remove all non-digits, including the newline:

```
6$ echo '2 ^ 128' | bc | tr -cd '[0-9]' | wc -c
      39
```

Use **awk** to remove the annoying indentation. The **awk** will not remove the newline.

```
7$ echo '2 ^ 128' | bc | tr -cd '[0-9]' | wc -c | awk '{print $1}'
39
```

The radius of the earth is 3,956.66 miles. Each mile is 5,280 feet. Each foot is 12 inches. What is the radius of the earth in inches? See **scale** in **bc**(1). The area of a sphere is $A = 4\pi r^2$. What is the area of the earth in square inches? How many IP version 6 addresses are there per square inch?

Bitwise not

Let's turn on, turn off, and examine the bits of a number. The rules for the bitwise operations "and", "or", and "not" are simpler than the rules for addition and subtraction, because there is no carrying or borrowing. Each column has no effect on its neighbors.

I wrote the following example in hexadecimal and binary. A hexadecimal number starts with a **0x** in the languages C, C++, Perl, and Java, but not in **bc**. As we'll see shortly, **0xFFFFFFC0** is the netmask for our network: 26 **1**'s followed by six **0**'s. For legibility, I inserted a blank after every eight bits.

The **~** operator (tilde) flips the bits, giving you a photographic negative of the original number. The easiest way to work out the value of **~0xFFFFFFC0** is to convert it to binary, flip the bits, and convert the result back to hex.

```
 0xFFFFFFC0 == 11111111 11111111 11111111 11000000
~0xFFFFFFC0 == 00000000 00000000 00000000 00111111 == 0x0000003F
```

Left shift

Left shift is simpler than right shift. When you left shift, bits fall off the left end into oblivion; fresh zeroes enter from the right end. Assuming that the 32-bit number is unsigned,

```
         1       == 00000000 00000000 00000000 00000001 ==           1
         1 <<  1 == 00000000 00000000 00000000 00000010 ==           2
         1 <<  2 == 00000000 00000000 00000000 00000100 ==           4
         1 <<  3 == 00000000 00000000 00000000 00001000 ==           8
         1 <<  4 == 00000000 00000000 00000000 00010000 ==          16
         1 << 30 == 01000000 00000000 00000000 00000000 ==  1073741824
         1 << 31 == 10000000 00000000 00000000 00000000 ==  2147483648
         1 << 32 == 00000000 00000000 00000000 00000000 ==           0
```

The next example assumes that the 32-bit number is signed. −1 is all ones. **-1 << 6** will yield the netmask on p. 28 in line 25 of **localhosts.pl**.

```
        -1       == 11111111 11111111 11111111 11111111 ==          -1
        -1 <<  1 == 11111111 11111111 11111111 11111110 ==          -2
        -1 <<  2 == 11111111 11111111 11111111 11111100 ==          -4
        -1 <<  3 == 11111111 11111111 11111111 11111000 ==          -8
        -1 <<  4 == 11111111 11111111 11111111 11110000 ==         -16
        -1 <<  6 == 11111111 11111111 11111111 11000000 ==         -64
        -1 << 30 == 11000000 00000000 00000000 00000000 == -1073741824
        -1 << 31 == 10000000 00000000 00000000 00000000 == -2147483648
        -1 << 32 == 00000000 00000000 00000000 00000000 ==           0
```

Right shift

When you right shift, bits fall off the right end into oblivion; fresh bits enter from the left end. If the right-shifted number is unsigned, the fresh bits will always be zeroes. For example,

```
4294967295       == 11111111 11111111 11111111 11111111 == 4294967295
4294967295 >>  1 == 01111111 11111111 11111111 11111111 == 2147483647
4294967295 >>  2 == 00111111 11111111 11111111 11111111 == 1073741823
4294967295 >>  3 == 00011111 11111111 11111111 11111111 ==  536870911
4294967295 >>  4 == 00001111 11111111 11111111 11111111 ==  268435455
4294967295 >> 30 == 00000000 00000000 00000000 00000011 ==          3
4294967295 >> 31 == 00000000 00000000 00000000 00000001 ==          1
4294967295 >> 32 == 00000000 00000000 00000000 00000000 ==          0
```

But if the right-shifted number is signed, the fresh bits will always be copies of the original sign bit, yielding a result with the same sign as the original number. After all, right shift really means ‘‘division by 2’’, which should not change whether a number is positive or negative. Here’s an example where the sign bit is **0**. A positive number stays positive:

```
2147483647       == 01111111 11111111 11111111 11111111 ==  2147483647
2147483647 >>  1 == 00111111 11111111 11111111 11111111 ==  1073741823
2147483647 >>  2 == 00011111 11111111 11111111 11111111 ==   536870911
2147483647 >>  3 == 00001111 11111111 11111111 11111111 ==   268435455
2147483647 >>  4 == 00000111 11111111 11111111 11111111 ==   134217727
2147483647 >> 29 == 00000000 00000000 00000000 00000011 ==           3
2147483647 >> 30 == 00000000 00000000 00000000 00000001 ==           1
2147483647 >> 31 == 00000000 00000000 00000000 00000000 ==           0
```

And here’s an example where the sign bit is **1**. A negative number stays negative:

```
    -2147483648       == 10000000 00000000 00000000 00000000 == -2147483648
    -2147483648 >>  1 == 11000000 00000000 00000000 00000000 == -1073741824
    -2147483648 >>  2 == 11100000 00000000 00000000 00000000 ==  -536870912
    -2147483648 >>  3 == 11110000 00000000 00000000 00000000 ==  -268435456
    -2147483648 >>  4 == 11111000 00000000 00000000 00000000 ==  -134217728
    -2147483648 >> 30 == 11111111 11111111 11111111 11111110 ==          -2
    -2147483648 >> 31 == 11111111 11111111 11111111 11111111 ==          -1
    -2147483648 >> 32 == 11111111 11111111 11111111 11111111 ==          -1
```

Turn bits off with ‘‘bitwise and’’

Write the operands one above the other in binary. For ‘‘bitwise and’’, each bit of the result will be **`1`** if *all* of the bits above it were **`1`**. Otherwise, the resulting bit will be **`0`**. ‘‘Bitwise and’’ therefore turns off selected bits, leaving the other bits unchanged. For example, **`0x807AFD98`** is the IP address of our host i5.nyu.edu: 128.122.253.152 in *dotted quad* notation. **`0xFFFFFFC0`** is our netmask: 26 **`1`**’s followed by six **`0`**’s. The result will be the IP address of i5.nyu.edu with the six rightmost bits turned off. Later we’ll see that this is the IP address of our network.

```
0x807AFD98              == 10000000 01111010 11111101 10011000 == 128.122.253.152
0xFFFFFFC0              == 11111111 11111111 11111111 11000000 == 255.255.255.192
                           -----------------------------------
0x807AFD98 & 0xFFFFFFC0 == 10000000 01111010 11111101 10000000 == 128.122.253.128
```

Turn bits on with ‘‘bitwise or’’

For ‘‘bitwise or’’, each bit of the result will be **`0`** if *all* of the bits above it were **`0`**. Otherwise, the resulting will be **`1`**. ‘‘Bitwise or’’ therefore turns on selected bits, leaving the other bits unchanged. For example, **`0x807AFD98`** is the IP address of our host i5.nyu.edu: 128.122.253.152. **`~0xFFFFFFC0`** is a photographic negative of our netmask: 26 **`0`**’s followed by six **`1`**’s. The result will be the IP address of i5.nyu.edu with the six rightmost bits turned on. Later we’ll see that this is the broadcast address for our network.

```
 0x807AFD98              == 10000000 01111010 11111101 10011000 == 128.122.253.152
~0xFFFFFFC0              == 00000000 00000000 00000000 00111111 == 0x0000003F
                            -----------------------------------
0x807AFD98 | ~0xFFFFFFC0 == 10000000 01111010 11111101 10111111 == 128.122.253.191
```

Examine an individual bit

Suppose we wanted to know if bit 3, for example, of our host’s IP address **`0x807AFD98`** (128.122.253.152) is on or off. (It’s on.) First we create the mask **`1 << 3`**, whose bit 3 is on and whose 31 other bits are off. Then we ‘‘bitwise and’’ the IP address with the mask to obliterate every bit except bit 3. Bit 3 of the original IP address survives unchanged.

```
 0x807AFD98              == 10000000 01111010 11111101 10011000 == 128.122.253.152
     1 << 3              == 00000000 00000000 00000000 00001000
                            -----------------------------------
 0x807AFD98 & 1 << 3     == 00000000 00000000 00000000 00001000 does not equal 0
```

C, C++, Perl, and Java agree that **`<<`** has higher precedence than **`&`**, so no parentheses are required. **`bc`** does not have **`&`** and **`|`**.

A protocol is a set of rules.

> TCP provides reliability with a mechanism called *Positive Acknowledgement with Retransmission* (PAR). Simply stated, a system using PAR sends the data again *unless* it hears from the remote system that the data arrived OK. . . . After this exchange, host *A*’s TCP has positive evidence that the remote TCP is alive and ready to receive data.
>
> —Craig Hunt, *TCP/IP Network Administration, 3rd ed.* (2002), pp. 19−20

> “The Fail-Safe point is different for each group,” General Bogan explained. “They also change from day to day. There is a fixed point in the sky where the planes will orbit until they get a positive order to go in. Without it they must return to the United States. This is called Positive Control. Fail-Safe simply means that if something fails it is still safe. In short, we cannot go to war except by a direct order. No bomber can go in on its own discretion. We give that order.”
>
> —Eugene Burdick & Harvey Wheeler, *Fail-Safe* (1962), chapter 2

A *protocol* is a set of rules that two programs agree to obey when they talk to each other (p. 4). For example, here are two rules from the Ethernet protocol. The data to be transmitted between the two programs must be divided into sections called *frames,* of at most 1500 bytes each (p. 148). The first 14 bytes of each frame must be a header telling which host sent the frame and which host is to receive the frame. Each host is specified by a six-byte “Ethernet address”.

Similarly, here are two rules from the IP protocol (version 4, pp. 13−14). The data to be transmitted between the two programs must be divided into sections called *datagrams.* The first 20 or 24 bytes of each datagram must be a header telling which host sent the datagram and which host is to receive the datagram. In this protocol, however, each host is specified by means of a four-byte “IP address”.

Finally, here are two rules from the TCP protocol (pp. 19−22). The data to be transmitted between the two programs must be divided into sections called *segments.* The first 20 or so bytes of each segment must be a header telling which program on the source host sent the segment, and which program on the destination host is to receive the segment. Each program is specified by means of a two-byte “TCP port number”.

Why are there three such similar protocols? See “Why do we have more than one layer”, below.

The stack of protocols (p. 7)

Programs do not send naked TCP segments back and forth. (Why not? See “Why do we have more than one layer”, below.) Each TCP segment is contained in an IP datagram, which carries the TCP segment to the other host. In other words, each TCP segment is the payload, or contents, of an IP datagram.

Furthermore, programs do not send naked IP datagrams back and forth. Each IP datagram (or fragment thereof) is contained in an Ethernet frame, which carries the IP datagram to the other host. In other words, each IP datagram is the payload, or contents, of an Ethernet frame.

In the following diagram, **`08:00:20:c9:a0:09`** and **`00:d0:bc:c9:a0:0a`** are six-byte Ethernet addresses. **`128.122.253.152`** and **`128.122.253.153`** are four-byte IP addresses. **`10000`** and **`10001`** are two-byte TCP port numbers.

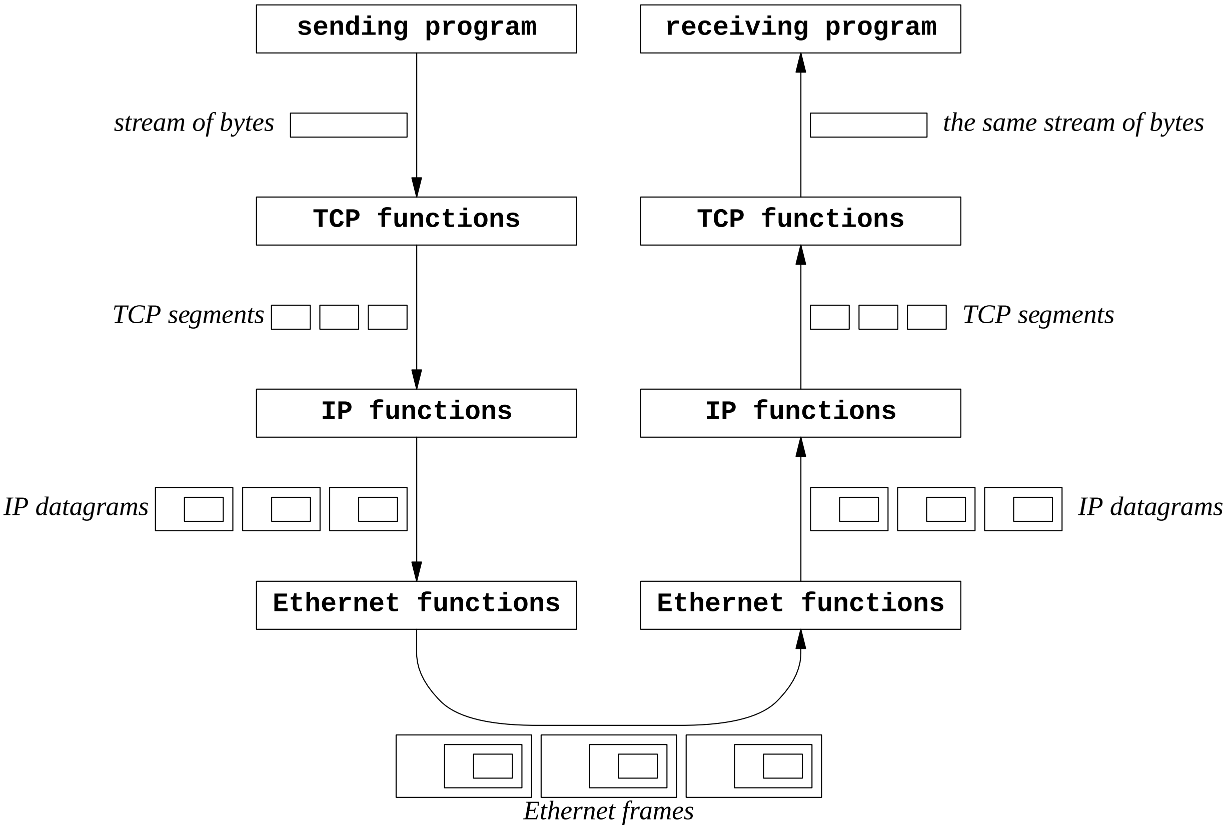
Ethernet header	IP header	TCP header	
From: 08:00:20:c9:a0:09	From: 128.122.253.152	From: port 10000	"hello"
To: 00:d0:bc:c9:a0:0a	To: 128.122.253.153	To: port 10001	

When two communicating programs are obeying the rules of the TCP protocol, we say that they’re “communicating via TCP”. The sending program calls functions (or subroutines, or procedures—the name is different in each programming language) that break the stream of data into segments, and the receiving program calls other functions that reassemble the segments back into a continuous stream of data.

When we say ‘‘TCP reassembles the segments’’, we don’t mean the TCP protocol itself. Literally speaking, the TCP protocol can’t *do* anything—it’s merely a set of rules. We use the term ‘‘TCP’’ as a shorthand for ‘‘the functions that the program calls to obey the TCP protocol’’.

The sending program calls TCP functions to break the data into TCP segments. The TCP functions then call IP functions to encapsulate (surround) each TCP segment in an IP datagram. The IP functions then call Ethernet functions to encapsulate each IP datagram in an Ethernet frame. This process is called ‘‘sending data down the protocol stack’’.

The receiving program calls other Ethernet functions to strip away the Ethernet frame that surrounds each IP datagram. The Ethernet functions then give the IP datagrams to IP functions, which strip away the IP datagram that surrounds each TCP segment. The IP functions then give the TCP segments to TCP functions, which reassemble the segments into a continuous stream. This process is called ‘‘sending data up the protocol stack’’.



Two complications:

(1) I’m sorry that each protocol has a different word for essentially the same thing: segment, datagram, frame, packet, message. See p. 11.

(2) Ethernet is the most widely used LAN (Local Area Network) hardware. But IP can ‘‘run on’’ (i.e., be carried by) many other kinds of hardware: X.25, Token Ring, etc. See p. 16.

Why do we have more than one layer?

(1) If you all you need is to send data to another host on the same Local Area Network (LAN), then all you need is Ethernet. The example we will see is ARP messages.

(2) An Ethernet frame can travel only between two hosts on the same LAN. To send data to a host on another LAN, you need IP as well as Ethernet. An IP datagram can travel across many LAN’s to its final destination. As it passes the boundary of each successive LAN, the surrounding Ethernet frame will fall off

of the datagram and be replaced by another Ethernet frame. This ‘‘molting’’ can happen up to 255 times; see the ‘‘time to live’’ field of the IP datagram header in pp. 14, 679, 681.

(3) There may be many programs running on the receiving host. Which program gets to receive the incoming IP datagram? IP gives you no way to specify this. An IP datagram is therefore delivered to the operating system on the receiving host, rather than to any specific program.

To send data to a specific program, you need TCP (or UPD) as well as IP. Each TCP segment (or UDP packet) carries a ‘‘port number’’ identifying the program that should receive it.

Now let’s talk about what the packets *don’t* carry. Neither the Ethernet, IP, TCP, nor UDP headers carry login names, so they don’t tell you which person sent or received the packets. However, login names may be transported as TCP or UDP cargo.

Don’t use more layers than you need: it would slow you down. For example, ARP packets travel only within a LAN, and they are addressed to the operating system on the receiving host. They can therefore be carried directly by Ethernet without IP or TCP. On the other hand, ICMP packets need to travel to other LAN’s, although they too are addressed to the operating system on the receiving host. They are therefore carried by Ethernet and IP, but without TCP. Finally, HTTP messages need to travel to other LAN’s, and must be delivered to one specific program on the receiving host: the web browser. They are therefore carried by Ethernet, IP, and TCP.

Shedding an ethernet frame

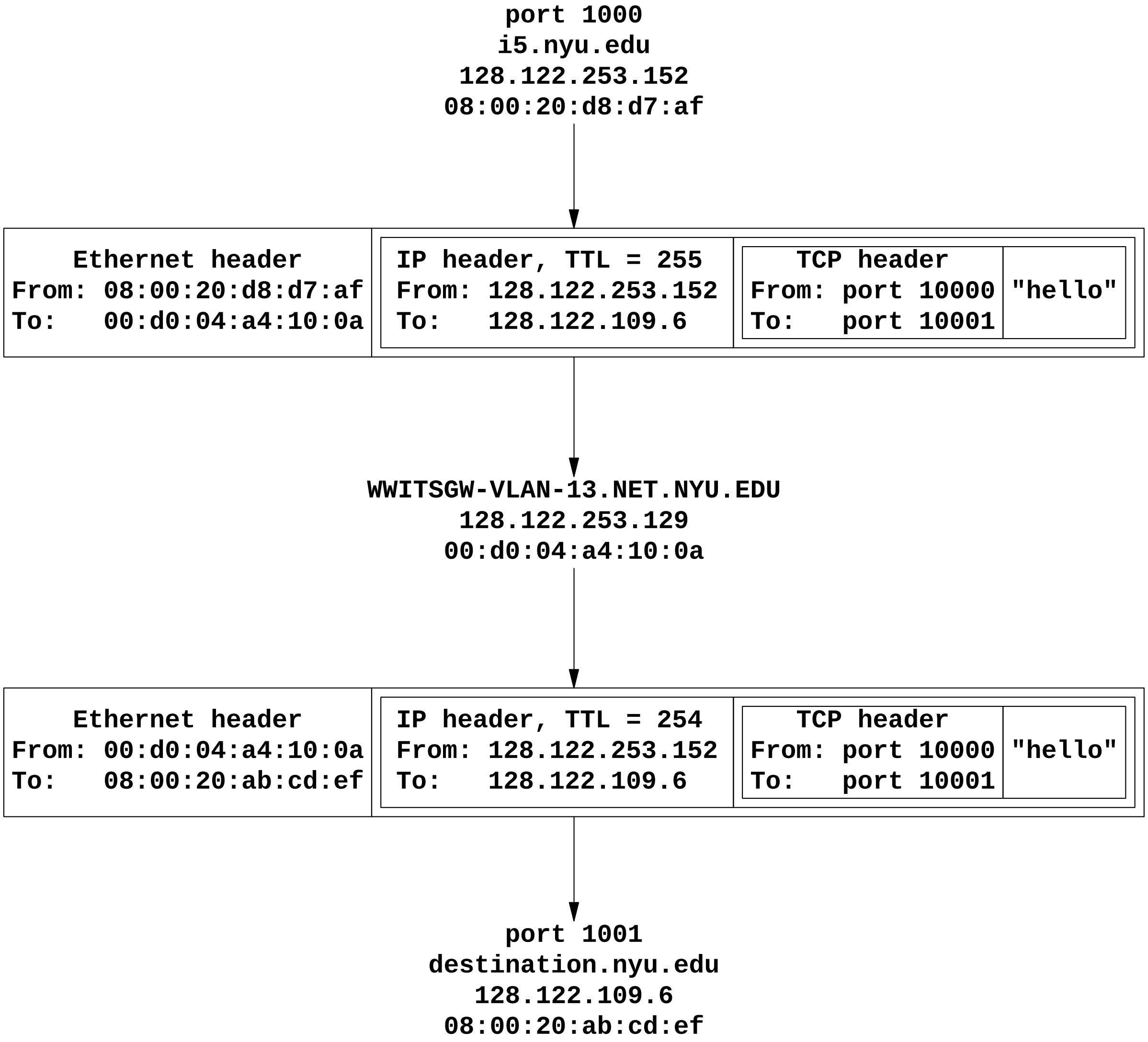
An Ethernet *frame* contains an IP *datagram,* which contains a TCP *segment.* I’m sorry we have three different words for the same thing (p. 11).

An Ethernet frame can not travel beyond the local network. At each gateway, the frame’s payload (the IP datagram) ‘‘molts’’. The IP datagram is copied into a new frame and sent onward for another ‘‘hop’’.

In the following example, **`i5.nyu.edu`** and **`WWITSGW-VLAN-13.NET.NYU.EDU`** are both on the same network. And **`WWITSGW-VLAN-13.NET.NYU.EDU`** and **`destination.nyu.edu`** are both on another network. **`WWITSGW-VLAN-13.NET.NYU.EDU`** therefore belongs to both networks, and is called a *gateway* or *router* (p. 14).

Note that we do not use any port number on the middle machine **`WWITSGW-VLAN-13.NET.NYU.EDU`**. Port numbers are used only to specify the original source and ultimate destination. The job of the middle machine is handled by its operating system.

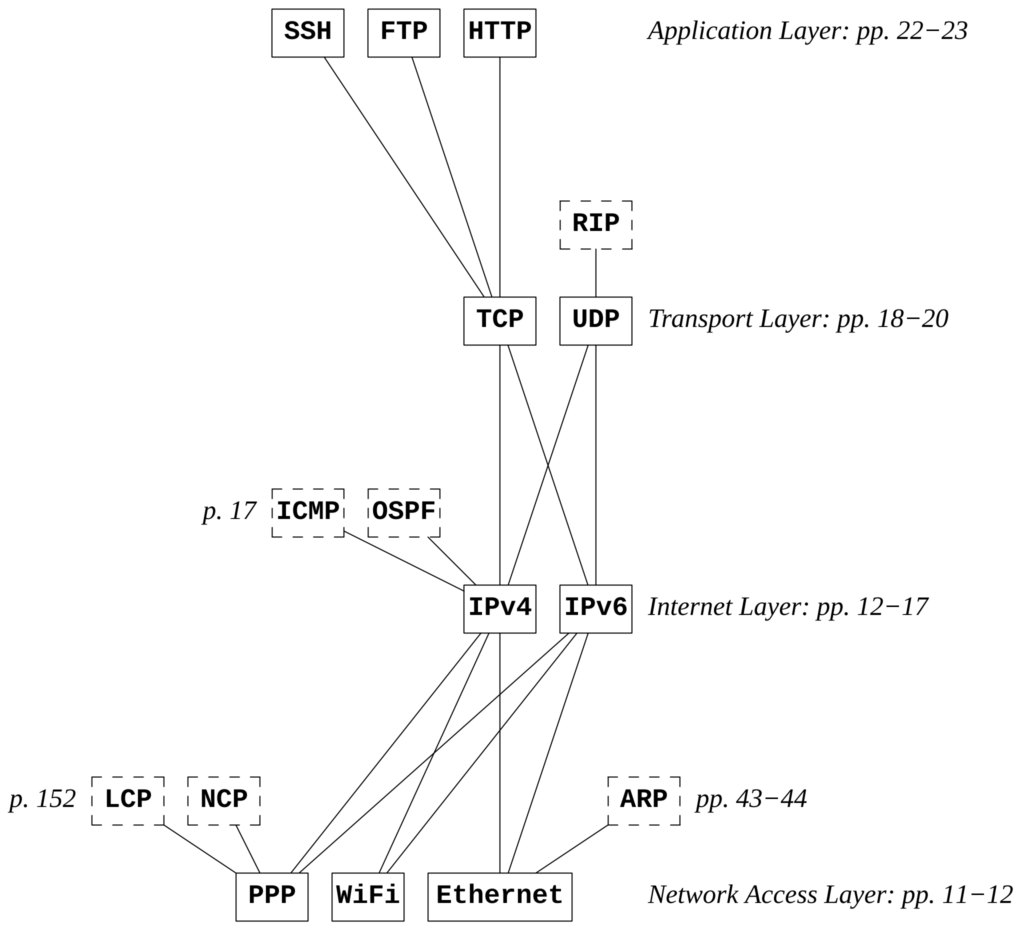
The IP datagram ages as it hops. When its time to live (TTL) goes down to zero, it dies, and an ICMP **`ICMP_TIMXCEED`** message is sent back to the original program. These messages are also used by **`traceroute`**.



Who carries whom?

Solid boxes represent protocols that can carry any kind of data: text, sound, video, etc. Dashed boxes represent protocols that can carry only special-purpose numeric codes. For example, ARP carries only Ethernet addresses and IP addresses; ICMP carries only codes such as ‘‘echo request’’, ‘‘echo reply’’, ‘‘destination unreachable’’, etc.

Each vertical and diagonal line means ‘‘is carried by’’. For example, each TCP segment is carried by an IP datagram. A more elaborate diagram is in Stevens, p. 30.


SSH
FTP
HTTP
Application Layer: pp. 22−23
RIP
TCP
UDP
Transport Layer: pp. 18−20
p. 17
ICMP
OSPF
IPv4
IPv6
Internet Layer: pp. 12−17
p. 152
LCP
NCP
ARP
pp. 43−44
PPP
WiFi
Ethernet
Network Access Layer: pp. 11−12

Names and numbers

Many things have both names and numbers:

```
1$ awk -F: '$1 == "mm64"' /etc/passwd
mm64:x:50766:15:Mark Meretzky:/home1/m/mm64:/bin/ksh

2$ awk -F: '$3 == 15' /etc/group
users::15:

3$ awk -F: '$1 == "x52954700120053"' /etc/group
x52954700120053::8836:
```

Every file has a name and an ‘‘inode number’’ (index node number). Here are two files with the same name:

```
4$ ls -li /bin/head /usr/local/bin/head
266 -r-xr-xr-x   1 root     bin         6216 Apr  6  2002 /bin/head
49 -rwxr-xr-x   1 root     other      91292 Mar  5  2001 /usr/local/bin/head
```

Here’s how I discovered the two files with the same name:

```
5$ find / -name head 2> /dev/null
```

And here is a file with two names:

```
6$ cd /opt/sfw/bin
7$ ls -li emacs emacs-21.2
2686 -r-xr-xr-x   2 root     bin      4868160 May 19  2003 emacs
2686 -r-xr-xr-x   2 root     bin      4868160 May 19  2003 emacs-21.2
```

Here’s how I discovered the other name:

```
    8$ find / -inum 2686 2> /dev/null
```

The IPv4 address of our host i5.nyu.edu is 10000000011110101111110110011000

The IP (version 4) address of our host `i5.nyu.edu` is the 32-bit number

```
    10000000011110101111110110011000          32 bits
    10000000 01111010 11111101 10011000       four “octets” of eight bits each
    128.122.253.152                           dotted quad notation: each octet in decimal
```

Find the IP address(es) and netmask(s) of your host

(1) To find the IP address and netmask of a Macintosh host running OS9,

```
    Apple Menu -> Control Panels -> TCP/IP
```

(2) To find the IP address and netmask of a Macintosh host running OSX,

```
    Apple Menu -> System Preferences... -> Network -> TCP/IP
```

(3) To find the IP address and netmask of a Windows host,

```
    Start -> Programs -> Accessories -> Command Prompt
    ipconfig        for Windows 2000 or NT
    winipcfg        for Windows 95 or 98
```

For example,

```
    C:\WINDOWS> ipconfig
    C:\WINDOWS> ipconfig /all | more

    Windows 98 IP Configuration

    0 Ethernet adapter :

        IP Address. . . . . . . . . : 192.168.20.19
        Subnet Mask . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . : 192.168.20.1
```

(4) Do the following two websites give the same answer as the above? If not, you may be behind a box that performs NAT (‘‘Network Address Translation’’).

```
    http://www.whatismyip.com/
    http://i5.nyu.edu/cgi-bin/cgiwrap/mm64/whatismyip
```

To read my CGI program, see

```
    ~mm64/public_html/cgi-bin/whatismyip
```

on `i5.nyu.edu`. The more elaborate gateway

```
    http://www.netcraft.com/oldwhats/
```

will tell you what operating system a host is running. Also try

```
http://gotomypc.com/
```

(5) On a Unix host, `ifconfig` shows you the name, IP address, and netmask of each interface. `lo` stands for “loopback”. `ge` stands for “Gigabit Ethernet”: a billion bits per second. (Giga means 1,000,000,000.) The IP and broadcast addresses are displayed by `ifconfig` in decimal, but the netmask is in hex.

```
1$ ifconfig -a | more                  -a for “all”
lo0: flags=1000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4> mtu 8232 index 1
    inet 127.0.0.1 netmask ff000000
ge0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
    inet 128.122.253.152 netmask ffffffc0 broadcast 128.122.253.191
```

It appears that the hosts on our local network have class B addresses. For example, the IP address of `i5.nyu.edu` is `128.122.253.152`. You would therefore think that they have 16 network bits and 16 host bits. In reality, however, the netmask tells us they have 26 network bits and only 6 host bits.

Sometimes the netmask is written as four decimal numbers, separated by dots. Our netmask `0xFFFFFFC0` would be `255.255.255.192`.

Ask DNS for the IP address of any host

Each network interface of a host has its own IP address. If a host has several interfaces, it will have several IP addresses. `indira.uu.nl` has three; `www.microsoft.com` has eight.

```
1$ /usr/sbin/nslookup indira.uu.nl | cat -n
     1  Server:  NYUNSB.NYU.EDU
     2  Address:  128.122.253.37
     3
     4  Non-authoritative answer:
     5  Name:    indira.uu.nl
     6  Addresses:  131.211.5.1, 131.211.16.1, 131.211.17.1
     7

2$ /usr/sbin/nslookup 131.211.16.1 | cat -n
     1  Server:  NYUNSB.NYU.EDU
     2  Address:  128.122.253.37
     3
     4  Name:    Indira.uu.nl
     5  Address:  131.211.16.1
     6

3$ /usr/sbin/dig indira.uu.nl | more              change hostname to IP address
4$ /usr/sbin/dig -x 131.211.16.1 | more           -x to change IP address to hostname
```

Your $PATH environment variable

Your `$PATH` environment variable is created in your `~/.profile` file.

```
1$ cd
2$ pwd
3$ ls -la | more                       -a for “all”, even names that start with dot
```

```
4$ echo $PATH
/home1/m/mm64/bin:/bin:/usr/bin:/usr/local/bin:/usr/openwin/bin:/opt/sfw/bin:/usr/ucb:/opt/sfw/esp/
```

`012` is the octal ASCII code for the newline character. `sfw` is “Sun Freeware”.

```
http://www.sunfreeware.com/
```

Dot is the current directory.

```
5$ echo $PATH | tr : '\012' | cat -n | more
     1  /home1/m/mm64/bin
     2  /bin
     3  /usr/bin
     4  /usr/local/bin
     5  /usr/openwin/bin
     6  /opt/sfw/bin
     7  /usr/ucb
     8  /opt/sfw/esp/bin
     9  /usr/sbin
    10  .

6$ man -s 5 environ                                  section 5 of the manual
7$ env | sort -df | more     See all your environment variables; "dictionary, fold".
b45=/home1/m/mm64/public_html/x52.9545/bio
DISPLAY=192.168.20.196:0.0
EDITOR=/bin/vi
EXINIT=set showmode
GROFF_FONT_PATH=/home1/m/mm64/45/handout
```

The IP version 4 address classes: pp. 30−32

The leftmost bits of an IP address are the *network bits;* they identify which network the host belongs to. The rightmost bits of an IP address are the *host bits;* they identify the host. You're not allowed to use all `0`'s or all `1`'s as the host bits. (All `0`'s would be the address of the network; all `1`'s would be the broadcast address of the network.) That's why the numbers in the last column are always 2 less than a power of 2.

class	*first octet*	*network bits*	*host bits*	*how many networks of this class*	*maximum number of hosts per network*
A	1−126	8	24	126	$16,777,214 = 2^{24} - 2$
B	128−191	16	16	$16,384 = 64 \times 2^{8}$	$65,534 = 2^{16} - 2$
C	192−223	24	8	$2,097,152 = 32 \times 2^{16}$	$254 = 2^{8} - 2$
loopback	127			1	1
D (multicast)	224−239				
E (experimental)	240−254				

Who were the 126 lucky organizations who got class A addresses?

See **`whois`** at

```
http://www.geektools.com/
```

1.0.0.0 to 1.255.255.255	*Internet Assigned Numbers Authority:* **http://www.iana.net/**
2.0.0.0 to 2.255.255.255	*Internet Assigned Numbers Authority:* **http://www.iana.net/**
3.0.0.0 to 3.255.255.255	*General Electric Company:* **http://www.ge.com/**
4.0.0.0 to 4.255.255.255	*Genuity:* **http://www.genuity.com/**
5.0.0.0 to 5.255.255.255	*Internet Assigned Numbers Authority:* **http://www.iana.net/**
6.0.0.0 to 6.255.255.255	*Department of Defense Network Info Center:* **http://www.nic.mil/**
7.0.0.0 to 7.255.255.255	*Department of Defense Network Info Center:* **http://www.nic.mil/**
8.0.0.0 to 8.255.255.255	*Genuity:* **http://www.genuity.com/**
9.0.0.0 to 9.255.255.255	*IBM:* **http://www.ibm.com/**
10.0.0.0 to 10.255.255.255	*Internet Assigned Numbers Authority:* **http://www.iana.net/**

Who were the 122 organizations who got class B addresses ahead of NYU?

128.0.0.0 to 128.0.255.255	*Internet Assigned Numbers Authority:* **http://www.iana.net/**
128.1.0.0 to 128.1.255.255	*BBN Communications:* **http://www.bbn.com/**
128.2.0.0 to 128.2.255.255	*Carnegie Mellon University:* **http://www.cmu.edu/**
128.3.0.0 to 128.3.255.255	*Lawrence Berkeley National Laboratory:* **http://www.lbl.gov/**
128.4.0.0 to 128.4.255.255	*University of Delaware:* **http://www.udel.edu/**
128.5.0.0 to 128.5.255.255	*Ford Motor Company:* **http://www.ford.com/**
128.6.0.0 to 128.6.255.255	*Rutgers University:* **http://www.rutgers.edu/**
128.7.0.0 to 128.7.255.255	*FGAN - FFM:* **http://www.fgan.de/**
128.8.0.0 to 128.8.255.255	*University of Maryland:* **http://www.umaryland.edu/**
128.9.0.0 to 128.9.255.255	*Information Sciences Institute:* **http://www.isi.edu/**

128.121.0.0 to 128.121.255.255	*Verio, Inc.:* **http://www.verio.net/**
128.122.0.0 to 128.122.255.255	*New York University:* **http://www.nyu.edu/**
128.123.0.0 to 128.123.255.255	*New Mexico State University:* **http://www.nmsu.edu/**

Test the bits in an IPv4 address

> ...two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of *Blefuscu;* and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed, that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the *Big-Endians* have been long forbidden, and the whole party rendered incapable by Law of holding Employments.

—Jonathan Swift, *Gulliver's Travels,* Part I: *A Voyage to Lilliput,* Chapter 4

Where do the numbers for the first octet come from in the above table? We had an easy way to tell if a signed number was negative: just see if the topmost bit is 1. Similarly, there is an easy way to see if an IP address is class A (including the loopback address) or not: just see if the topmost bit (bit 31) is 0.

```
127 == 0111 1111
128 == 1000 0000
```

If the top bit is 1, it’s not an class A address. There is now an easy way to see if it is class B or not: just see if the next bit (bit 30) is 0.

```
191 == 1011 1111
192 == 1100 0000
```

If it’s neither A nor B, there is an easy way to see if it’s class C or not: just see if the next bit (bit 29) is 0.

```
223 == 1101 1111
224 == 1110 0000
```

If it’s neither A, B, nor C there is an easy way to see if it’s class D or not: just see if the next bit (bit 28) is 0.

```
239 == 1110 1111
240 == 1111 0000
```

Use the data type `in_addr_t` in lines 12−13 of `class.c` to hold a 32-bit IPv4 address. Here’s how the name for this data type was created. The first `typedef` means that `in_addr_t` is just another name for `uint32_t`. The next `typedef` means that `uint32_t` is just another name for `unsigned int`. And an `unsigned int` on our machine is 32 bits. The `[`square brackets`]` of each wildcard contain one blank and one.

```
1$ grep 'typedef[    ]*in_addr_t;' /usr/include/netinet/in.h

2$ grep 'typedef[    ]*uint32_t;' /usr/include/sys/int_types.h
```

The `AF_INET` in line 22 is a macro that means “IP version 4”; `AF_INET6` would mean “IP version 6”.

```
3$ grep '#define[   ][   ]*AF_INET' /usr/include/sys/socket.h
#define AF_INET 2 /* internetwork: UDP, TCP, etc. */
#define AF_INET6 26 /* Internet Protocol, Version 6 */
```

The function `inet_pton` (“presentation to numeric”) in line 22 takes a string containing a dotted IPv4 address such as `"128.122.253.152"` and converts it to a 32-bit number whose four bytes are in the “big-endian” order used by the Internet (highest byte first). The manual page for each function tells which header files must be included (lines 4−8) and which libraries must be linked in with the `-l` (minus lowercase L) option of `gcc`.

```
4$ man inet_pton
5$ man -s 3socket inet_pton         Only Solaris needs the -s (for “section”).
http://i5.nyu.edu/˜mm64/man/        easiest way to read the manual
```

Unfortunately, some platforms (e.g., Pentium) need to have the four bytes in “little-endian” order to do arithmetic. The function `ntohl` (“network to host long”) in line 28 takes an Internet-order number (big-endian) and returns the same number with the four bytes rearranged into the local order. If the local order is the same as the Internet order, the return value of `ntol` is the same as its argument.

This program assumes that the loopback address is class A.

—On the Web at
`http://i5.nyu.edu/˜mm64/x52.9547/src/class.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <inttypes.h>

int main(int argc, char **argv)
{
    in_addr_t ip;   /* in network byte order */
    in_addr_t i;    /* in local byte order */
    char class;     /* IPv4 address class: A, B, C, D, or E */

    if (argc != 2) {
        fprintf(stderr, "%s: argument must be a dotted IPv4 address\n",
            argv[0]);
        return 1;
    }

    if (inet_pton(AF_INET, argv[1], &ip) != 1) {
        fprintf(stderr, "%s: argument %s must be a dotted IPv4 address\n",
            argv[0], argv[1]);
        return 2;
    }

    i = ntohl(ip);   /* network to host long */

    if ((i & 1 << 31) == 0) {
        class = ’A’;   /* Bit 31 is 0 (includes loopback address). */
    } else if ((i & 1 << 30) == 0) {
        class = ’B’;   /* Bit 31 is 1, bit 30 is 0. */
    } else if ((i & 1 << 29) == 0) {
        class = ’C’;   /* Bit 31 is 1, bit 30 is 1, bit 29 is 0. */
    } else if ((i & 1 << 28) == 0) {
        class = ’D’;   /* Bit 31 is 1, bit 30 is 1, 29 is 1, 28 is 0. */
    } else {
        class = ’E’;   /* Bit 31 is 1, bit 30 is 1, 29 is 1, 28 is 1. */
    }

    printf("%s is a class %c IPv4 address.\n", argv[1], class);
    return EXIT_SUCCESS;
}
```

```
6$ cd ˜/bin
7$ pwd

8$ lynx -source http://i5.nyu.edu/˜mm64/x52.9547/src/class.c > class.c
9$ ls -l class.c
```

Did the **lynx -source** *work?*

Minus lowercase LNSL for the Network Services Library:

```
10$ man -M /usr/local/man gcc
11$ cd ˜mm64/public_html/x52.9547/src
12$ /usr/local/bin/gcc -o ˜/bin/class class.c -lsocket -lnsl
```

```
13$ ls -l ~/bin/class
-rwx--x--x   1 mm64     users       7520 Dec  1 11:59 /home1/a/abc1234/bin/class

14$ class 128.122.253.152
128.122.253.152 is a class B IPv4 address.

15$ echo $?                See the exit status; echo $status if you're using the C shell.
0
```

Line 30 of the C program, line 13 of the Perl program, and line 18 of the Java program need the parentheses around the **i & 1 << 31** to force the **<<** and **&** to go before the **==**. If the argument is **128.122.253.152**, the line will perform

```
128.122.253.152 == 10000000 01111010 11111101 10011000
        1 << 31 == 10000000 00000000 00000000 00000000
                   -----------------------------------
                   10000000 00000000 00000000 00000000 does not equal 0
```

Line 32 of C, 15 of Perl, and 20 of Java will perform

```
128.122.253.152 == 10000000 01111010 11111101 10011000
        1 << 30 == 01000000 00000000 00000000 00000000
                   -----------------------------------
                   00000000 00000000 00000000 00000000 does equal 0
```

The library files are in **/usr/lib**:

```
16$ cd /usr/lib
17$ ls -l lib@(socket|nsl).a                          For @ see ksh(1) p. 14.
-rw-r--r--   1 root     bin      1184388 Oct  9  2003 libnsl.a
-rw-r--r--   1 root     bin        87048 Apr  6  2002 libsocket.a
```

The same program, in Perl

To find the full pathname of your Perl interpreter (line 1),

```
1$ which perl
/bin/perl
```

Every Perl statement ends with a semicolon. The name of a Perl variable or array element always starts with a dollar sign; see the **$ip** and **$ARGV[0]** in line 8, the **$i** in line 11, and the **$class** in line 14. The special variable **$0** in lines 5 and 9 is the name of the Perl program.

Perl has the same **'**single quotes**'**, **"**double quotes**"**, and **`**back quotes**`** as the Unix shell. The quotes in lines 5, 9, and 25 must be double, not single, because they enclose variables. The quotes in line 25 must also be double because they enclose an escape character (the newline **\n**).

The expression **@ARGV** in line 4 is the number of command line arguments; in this program it should be 1. If not, the **die** statement in line 5 will kill the program. The ominous **or** in line 9 means “or else”.

The array element **$ARGV[0]** in line 8 is the first command line argument; it should be a string containing a dotted IPv4 address such as **128.122.253.152**. The function **inet_aton** (“alphanumeric to numeric”) in line 8 takes the IPv4 address and returns a 32-bit number whose four bytes are in the “big-endian” order used by the Internet (highest byte first). Line 2 is needed for this and other networking functions.

Unfortunately, some platforms (e.g., Pentium) might need to have the four bytes in “little-endian” order to do arithmetic. The **unpack** in line 11 takes an Internet-order number (big-endian; the **'N'** stands for “network”) and returns the same number with the four bytes rearranged into the local order. If the local order is the same as the Internet order, the return value of this **unpack** is the same as its argument. It therefore does the same job as **htonl** in C. See **perlfunc**(1) pp. 42−48, 77.

—On the Web at
http://i5.nyu.edu/˜mm64/x52.9547/src/class.pl

```
#!/bin/perl
use Socket;

if (@ARGV != 1) {
    die "$0: argument must be a dotted IPv4 address";
}

$ip = inet_aton($ARGV[0])
    or die "$0: argument $ARGV[0] must be a dotted IPv4 address";

$i = unpack('N', $ip);

if (($i & 1 << 31) == 0) {
    $class = 'A';   #Bit 31 is 0 (includes loopback address).
} elsif (($i & 1 << 30) == 0) {
    $class = 'B';   #Bit 31 is 1, bit 30 is 0.
} elsif (($i & 1 << 29) == 0) {
    $class = 'C';   #Bit 31 is 1, bit 30 is 1, bit 29 is 0.
} elsif (($i & 1 << 28) == 0) {
    $class = 'D';   #Bit 31 is 1, bit 30 is 1, bit 29 is 1, bit 28 is 0.
} else {
    $class = 'E';   #Bit 31 is 1, bit 30 is 1, bit 29 is 1, bit 28 is 1.
}

print "$ARGV[0] is a class $class IPv4 address.\n";
exit 0;
```

```
2$ cd ˜/bin
3$ pwd

4$ lynx -source http://i5.nyu.edu/˜mm64/x52.9547/src/class.pl > class.pl
5$ ls -l class.pl                    Did the lynx -source work?
6$ chmod 755 class.pl
7$ ls -l class.pl                    Did the chmod work?

8$ class.pl 128.122.253.152
128.122.253.152 is a class B IPv4 address.

9$ echo $?                           See the exit status.
0
```

The same program, in Java

—On the Web at
http://i5.nyu.edu/˜mm64/x52.9547/src/Class.java

```
 1 import java.net.*;   //for class InetAddress in lines 12-13
 2
 3 class Class {
 4     static public void main(String argv[]) {
 5         if (argv.length != 1) {
```

```
 6             System.err.println(
 7                 "Class: argument must be a dotted IPv4 address");
 8             System.exit(1);
 9         }
10
11         try {
12             final InetAddress ip = InetAddress.getByName(argv[0]);
13             final byte[] b = ip.getAddress();   //array of 4 bytes
14             final byte i = b[0];                //the high order byte
15
16             char c;        //IPv4 address class: A, B, C, D, or E
17
18             if ((i & 1 << 7) == 0) {
19                 c = 'A';   //Bit 31 is 0 (includes loopback address).
20             } else if ((i & 1 << 6) == 0) {
21                 c = 'B';   //Bit 31 is 1, bit 30 is 0.
22             } else if ((i & 1 << 5) == 0) {
23                 c = 'C';   //Bit 31 is 1, bit 30 is 1, bit 29 is 0.
24             } else if ((i & 1 << 4) == 0) {
25                 c = 'D';   //Bit 31 is 1, bit 30 is 1, 29 is 1, 28 is 0.
26             } else {
27                 c = 'E';   //Bit 31 is 1, bit 30 is 1, 29 is 1, 28 is 1.
28             }
29
30             System.out.println(argv[0] + " is a class " + c
31                 + " IPv4 address.");
32             System.exit(0);
33         }
34
35         catch (Exception e) {   //may be thrown by getByName in line 12
36             System.err.println("argument " + argv[0]
37                 + " must be a dotted IPv4 address");
38             System.exit(2);
39         }
40     }
41 }
```

```
1$ javac Class.java                 Run the Java compiler; create Class.class

2$ ls -l Class.class
-rw-------   1 mm64     users       1168 Dec  1 11:59 Class.class

3$ java Class 128.122.253.152       Run the Java interpreter.
128.122.253.152 is a class B IPv4 address.

4$ echo $?                          See the exit status.
0
```

The format of an IPv4 address

The address of the network is stored in routing tables. The first host on our network, **`128.122.253.129`**, is our network’s gateway to the outside world; try

```
1$ /usr/sbin/traceroute www.u-tokyo.ac.jp
```

The broadcast address of the network is used by **ping**.

The limited broadcast address is used only by hosts that do not know the address or netmask of their network. Data sent to the broadcast address is usually not forwarded by a router. Data sent to the limited broadcast address is *never* forwarded by a router. The limited broadcast address is used by DHCP: pp. 80, 588.

```
                                 myself before I know my own IP address  00000000000000000000000000000000  0.0.0.0
                                                                           0   0   0   0   0   0   0   0

                                                                     8 network bits      24 host bits
                                                       loopback address  01111111000000000000000000000001  127.0.0.1
                                                                           7   F   0   0   0   0   0   1

                                                                              26 network bits      6 host bits
                                                 netmask of our network  11111111111111111111111111000000  255.255.255.192
                                                                           F   F   F   F   F   F   C   0

                                              IP address of our network  10000000011110101111110110000000  128.122.253.128/26
                                                                           8   0   7   A   F   D   8   0

                               IP address of the first host on our network  10000000011110101111110110000001  128.122.253.129
                                                                           8   0   7   A   F   D   8   1

                                      IP address of our host i5.nyu.edu  10000000011110101111110110011000  128.122.253.152
                                                                           8   0   7   A   F   D   9   8

                                IP address of the last host on our network  10000000011110101111110110111110  128.122.253.190
                                                                           8   0   7   A   F   D   B   E

                                       broadcast address of our network  10000000011110101111110110111111  128.122.253.191
                                                                           8   0   7   A   F   D   B   F

                                             limited broadcast address  11111111111111111111111111111111  255.255.255.255
                                                                           F   F   F   F   F   F   F   F
```

A netmask usually begins with bytes of 255 and ends with bytes of 0's. In between, here are the bytes that can appear at the boundary line:

hex	*binary*	*decimal*
FF	**11111111**	**255**
FE	**11111110**	**254**
FC	**11111100**	**252**
F8	**11111000**	**248**
F0	**11110000**	**240**
E0	**11100000**	**224**
C0	**11000000**	**192**
80	**10000000**	**128**
00	**00000000**	**0**

Broadcast an echo request to all the hosts on the local network

In the IP address of each host in our local network, the first 26 bits identify the network. That leaves 6 bits to identify each host within the network. Our network can therefore hold at most $2^6 - 2 = 62$ different hosts.

If you give the broadcast address to **`ping -s`**, it will broadcast ICMP packets carrying the echo request code **`ICMP_ECHO`**. Only the superuser is allowed to broadcast, so the setuid bit of **`ping`** is turned on:

```
1$ cd /usr/sbin
2$ ls -l ping
-r-sr-xr-x   1 root     bin        47788 Apr  6  2002 ping
```

`ping` broadcasts an echo request code with the sequence number 0. (The sequence numbers are in the **`icmp_seq`** field of the ICMP header.) It outputs a line whenever it receives an ICMP packet carrying the **`ICMP_ECHOREPLY`** code. These replies carry the same sequence number. After one second, **`ping`** broadcasts another echo request, this time with the sequence number 1. This continues until you kill the **`ping`** with a control-c.

`ms` means millisecond: one thousandth of a second.

```
3$ /usr/sbin/ping -s 128.122.253.191 | more    Press q to quit.
PING 128.122.253.191: 56 data bytes
64 bytes from i5.nyu.edu (128.122.253.152): icmp_seq=0. time=0. ms
64 bytes from HPC2.ES.ITS.NYU.EDU (128.122.253.140): icmp_seq=0. time=3. ms
64 bytes from AS17.ES.ITS.NYU.EDU (128.122.253.163): icmp_seq=0. time=12. ms
etc.
64 bytes from i5.nyu.edu (128.122.253.152): icmp_seq=1. time=0. ms
64 bytes from HPC2.ES.ITS.NYU.EDU (128.122.253.140): icmp_seq=1. time=1. ms
64 bytes from AS17.ES.ITS.NYU.EDU (128.122.253.163): icmp_seq=1. time=3. ms
etc.
64 bytes from i5.nyu.edu (128.122.253.152): icmp_seq=2. time=0. ms
64 bytes from AS11.ES.ITS.NYU.EDU (128.122.253.172): icmp_seq=2. time=1. ms
64 bytes from HPC2.ES.ITS.NYU.EDU (128.122.253.140): icmp_seq=2. time=3. ms
etc.
control-c
```

Where does ping get the hostnames from?

`ping` outputs the name of our host **`i5.nyu.edu`** in lowercase, and the other hosts in uppercase. This is a hint that **`ping`** gets its information from two different sources. The **`/etc/nsswitch.conf`** file (p. 271) tells any program that needs to convert an IP address to a hostname to consult a file on our host first. The file it consults is **`/etc/hosts`**. The DNS (the Domain Name System) is the last resort:

```
1$ awk ’$1 == "hosts:"’ /etc/nsswitch.conf
hosts:      files dns

2$ man -s 4 nsswitch.conf
```

`i5.nyu.edu` is written in lowercase in the **`/etc/hosts`** file:

```
3$ awk ’$2 == "i5.nyu.edu"’ /etc/hosts
128.122.253.152  i5.nyu.edu   i5 loghost

4$ man -s 4 hosts
```

But the other hosts are not listed in the **`/etc/hosts`** file, so **`ping`** falls back on DNS for them. DNS coughs up uppercase hostnames.

Cut off the ping

To cut off the output of **`ping`** as soon as we receive a packet whose sequence number is not zero,

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

```
#!/bin/ksh
#Ping every host on our network once.  Then stop.

if [[ $# -ne 1 ]]
then
    echo $0: requires network name 1>&2
    exit 1
fi

/usr/sbin/ping -s $1 |
awk ’NR >= 2 {
    if ($0 ~ /icmp_seq=0\./) {
        print
    } else {
        exit 0
    }
}’

exit 0
```

```
1$ cutoff 128.122.253.191 | cat -n | head -5
 1  64 bytes from i5.nyu.edu (128.122.253.152): icmp_seq=0. time=0. ms
 2  64 bytes from IONWEB.FAS.NYU.EDU (128.122.253.147): icmp_seq=0. time=3. ms
 3  64 bytes from AS11.ES.ITS.NYU.EDU (128.122.253.172): icmp_seq=0. time=13. ms
 4  64 bytes from AS17.ES.ITS.NYU.EDU (128.122.253.163): icmp_seq=0. time=16. ms
 5  64 bytes from HPC2.ES.ITS.NYU.EDU (128.122.253.140): icmp_seq=0. time=18. ms

2$ cutoff 128.122.253.191 | cat -n | tail -5
24  64 bytes from WWITSGW-VLAN-13.NET.NYU.EDU (128.122.253.129): icmp_seq=0. time=66. m
25  64 bytes from ITP.NYU.EDU (128.122.253.189): icmp_seq=0. time=69. ms
26  64 bytes from AS5.ES.ITS.NYU.EDU (128.122.253.148): icmp_seq=0. time=71. ms
27  64 bytes from AS3.ES.ITS.NYU.EDU (128.122.253.138): icmp_seq=0. time=74. ms
28  64 bytes from GOOGLE.NYU.EDU (128.122.253.155): icmp_seq=0. time=76. ms
```

Loop through all the hosts on the local network

We’ve seen that the function **`inet_pton`** (“presentation to numeric’’) in line 25 takes a string containing a dotted IPv4 address such as **`"128.122.253.152"`** and converts it to a 32-bit number whose four bytes are in the “big-endian’’ order used by the Internet (highest byte first). The function **`inet_ntop`** in line 57 does the opposite conversion: it converts a 32-bit number back to a dotted string.

We’ve also seen that the function **`ntohl`** in line 31 takes an Internet-order number (big-endian) and returns the same number with the four bytes in the local order. The function **`htonl`** in line 54 does the opposite conversion: “host to network long’’.

If the second argument is 26, line 40 of the C program and line 26 of the Perl program will let the number of host bits be 6. The **`-1`** in line 43 of C and line 29 of Perl is thirty-two **`1`**’s:

```
11111111 11111111 11111111 11111111
```

When line 43 of C (25 of Perl) shifts it 6 places to the left, the result stored in **`$netmask`** is

```
11111111 11111111 11111111 11000000
```

The expression **`~$netmask`** in line 51 of C (line 35 of Perl) is the photographic negative:

```
00000000 00000000 00000000 00111111
```

Line 47 of C program (line 32 of Perl) perform

```
  10000000 01111010 11111101 10011000   IP address of the given host
& 11111111 11111111 11111111 11000000   $netmask
  -----------------------------------
  10000000 01111010 11111101 10000000   address of the network
```

Line 51 of C (line 35 of Perl) perform

```
  10000000 01111010 11111101 10011000   IP address of the given host
| 00000000 00000000 00000000 00111111   ~$netmask
  -----------------------------------
  10000000 01111010 11111101 10111111   broadcast address of the network
```

The function **`gethostbyaddr`** in line 55 takes (the address of) a 32-bit IP address and returns a pointer to a structure whose **`h_name`** field in line 63 of C (line 39 of Perl) is the name of he host whose IP address was supplied.

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

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <netdb.h>
 4 #include <arpa/inet.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     in_addr_t ip;
 9     in_addr_t i;
10     in_addr_t netmask;
11     in_addr_t network;
12     in_addr_t broadcast;
13
14     int netbits;         /* how many network bits */
15     int hostbits;        /* how many host bits */
16     struct hostent *p;
17     char buffer[INET_ADDRSTRLEN];
```

```

    if (argc != 3) {
        fprintf(stderr, "%s: requires 2 arguments: IP netbits\n",
            argv[0]);
        return 1;
    }

    if (inet_pton(AF_INET, argv[1], &ip) != 1) {
        fprintf(stderr, "%s: first argument %s must be a dotted IPv4 address\n",
            argv[0], argv[1]);
        return 2;
    }

    i = ntohl(ip);

    netbits = strtol(argv[2], NULL, 10);   /* string to long */
    if (netbits <= 0 || netbits > 30) {
        fprintf(stderr, "%s: second argument %s must be number of network bits\n",
            argv[0], argv[2]);
        return 3;
    }

    hostbits = 32 - netbits;

    /* The netmask is netbits 1’s, followed by hostbits 0’s. */
    netmask = -1 << hostbits;

    /* The network address is the IP address, with all the host bits turned
    off. */
    network = i & netmask;

    /* The broadcast address is the IP address, with all the host bits
    turned on. */
    broadcast = i | ~netmask;

    for (i = network + 1; i < broadcast; ++i) {
        ip = htonl(i);
        p = gethostbyaddr((char *)&ip, sizeof ip, AF_INET);
        if (p != NULL) {
            if (inet_ntop(AF_INET, &ip, buffer, sizeof buffer) !=
                buffer) {
                fprintf(stderr, "%s: inet_ntop failed\n",
                    argv[0]);
                return 4;
            }
            printf("%s %s\n", buffer, p->h_name);
        }
    }

    return EXIT_SUCCESS;
}
```

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

4$ localhosts 128.122.253.152 26 | head -5 | cat -n
     1  128.122.253.129 WWITSGW-VLAN-13.NET.NYU.EDU
     2  128.122.253.131 NYU-DA3400-1-253-128.NS.ITS.NYU.EDU
     3  128.122.253.135 DEVXFILES.NYU.EDU
     4  128.122.253.136 BLOGS.NYU.EDU
     5  128.122.253.137 AS2.ES.ITS.NYU.EDU
```

Note that **`i5.nyu.edu`** comes out in lowercase:

```
5$ localhosts 128.122.253.152 26 | cat -n | awk ’15 <= NR && NR <= 17’
    15  128.122.253.150 HPC1.ES.ITS.NYU.EDU
    16  128.122.253.152 i5.nyu.edu
    17  128.122.253.155 GOOGLE.NYU.EDU

6$ localhosts 128.122.253.152 26 | cat -n | tail -5
    37  128.122.253.185 ITS.NYU.EDU
    38  128.122.253.186 IONDATA.TSOA.NYU.EDU
    39  128.122.253.187 ION1.TSOA.NYU.EDU
    40  128.122.253.188 ION2.TSOA.NYU.EDU
    41  128.122.253.189 ITP.NYU.EDU
```

The same program, in Perl

We’ve seen that the function **`inet_aton`** (“alphanumeric to numeric”) in line 21 can take a dotted IP address such as **`"128.122.253.152"`**. It can also take a hostname such as **`i5.nyu.edu`** and return the corresponding IP address as a 32-bit number whose four bytes are in the order (“big-endian”) used by the Internet. The function **`inet_ntoa`** (“numeric to alphanumeric”) in line 41 does the opposite conversion: it takes a 32-bit number whose four bytes are in Internet order and returns a string such as **`"128.122.253.152"`**.

We’ve seen that the **`unpack`** in line 24 takes a 32-bit number whose bytes are in Internet order and returns the same number with the four bytes in local order. The **`pack`** in line 38 does the opposite conversion: it takes a local-order number and returns the same number with the four bytes in Internet order.

The **`AF_INET`** in line 39 means “IP version 4”; **`AF_INET6`** would mean “IP version 6”. **`AF_INET`** is not a variable: it has no leading **`$`**. It’s a subroutine with no arguments, declared in

```
1$ cat -n /usr/perl5/5.6.1/lib/sun4-solaris-64int/Socket.pm |
sed -n 345p
   345  sub AF_INET      ();
```

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

```
1 #!/bin/perl
2 #Output the IP address and fully qualified domain name of every host
3 #on the same network as the given host.
4 #The first argument is the hostname or IP address of a given host,
5 #e.g., i5.nyu.edu or 128.122.253.152
6 #The second argument is the number of network bits in the netmask, e.g., 26
7
8 use Socket;
9
```

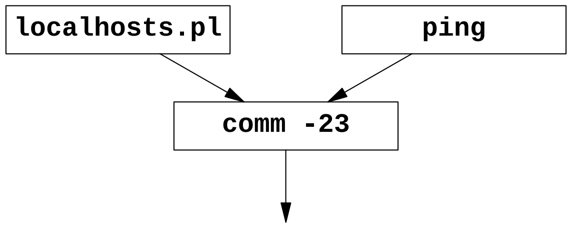
```
if (@ARGV != 2) {
        die "$0: requires 2 arguments";
}

$hostname = $ARGV[0];
$netbits  = $ARGV[1];   #number of network bits in the bitmask

if ($netbits <= 0 || $netbits > 30) {
    die "$0: netbits $netbits must be in range 1 to 30 inclusive";
}

$ip = inet_aton($hostname)
        or die "$0: argument $hostname must be a hostname or dotted IP address";

$i = unpack('N', $ip);

$hostbits = 32 - $netbits;

#The netmask is $netbits 1's, followed by $hostbits 0's.
$netmask = -1 << $hostbits;

#The network address is the IP address, with all the host bits turned off.
$network = $i & $netmask;

#The broadcast address is the IP address, with all the host bits turned on.
$broadcast = $i | ~$netmask;

for ($i = $network + 1; $i < $broadcast; ++$i) {
    $ip = pack('N', $i);
    $name = gethostbyaddr($ip, AF_INET);
    if (defined $name) {
        print inet_ntoa($ip), " $name\n";
    }
}

exit 0;
```

▼ **Homework 1.1: find all your local hosts**

Run **ifconfig** to find the IP address and netmask of your host. Then run **localhosts.c** or **localhosts.pl** to find the IP address and fully qualified domain name of every host on your local network. Hand in the output; write down how many network bits are in the netmask.

▲

Which local hosts are not answering a ping?



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

```
#!/bin/ksh
#Output the fully qualified domain name of every host on the
#local network that does not respond to a broadcast ping.

cd /tmp

~mm64/public_html/x52.9547/src/localhosts.pl 128.122.253.152 26 |
awk ’{print $2}’ |
tr ’[a-z]’ ’[A-Z]’ |
sort > local.$$

~mm64/public_html/x52.9547/src/cutoff 128.122.253.191 |
awk ’{print $4}’ |
tr ’[a-z]’ ’[A-Z]’ |
sort > ping.$$

comm -23 local.$$ ping.$$
rm local.$$ ping.$$
exit 0
```

```
1$ cd ~mm64/public_html/x52.9547/src
2$ noresponse | cat -n | tail -5
    11  IONDATA.TSOA.NYU.EDU
    12  MODIYA.NYU.EDU
    13  NEWI2.NYU.EDU
    14  NYU-DA3400-1-253-128.NS.ITS.NYU.EDU
    15  POLARIS.ITS.NYU.EDU
```

▼ Homework 1.2: display all the NYU hostnames

NYU has many LANs in addition to our LAN **128.122.253.128/26**. IN fact, NYU owns all the IPv4 addresses from **128.122.0.0** to **128.122.255.255** inclusive.

Write a program to display the IP address and fully qualified domain name of every NYU IP address that has a fully qualified domain name, from **128.122.0.0** to **128.122.255.255**. Run the program in the background because it takes too long:

```
1$ allnyu > allnyu.out 2>&1 &

128.122.0.0 NYU-NET
128.122.255.255 NYU-BROADCAST
```

▲

IP version 6 addresses

See RFC 3513 for IPv6 addresses. (RFC’s come from the IETF: Internet Engineering Task Force.) The Belgian National Research Network has hosts with IPv6 addresses as well as IPv4 addresses. Also try **www.ipv6.org**, **ipv6.linux-tech.com**.

By default, **nslookup** will display only IP version 4 addresses. They are called “type **A**”, for “address”.

```
    1$ /usr/sbin/nslookup patah.belnet.be
    Server:  NYUNSB.NYU.EDU
    Address:  128.122.253.37

    Non-authoritative answer:
    Name:    patah.belnet.be
    Address:  193.190.198.30
```

To see IP version 6 addresses ask for type **AAAA**, which are four times as long as IP version 4.

```
    2$ /usr/sbin/nslookup -type=AAAA patah.belnet.be
    Server:  NYUNSB.NYU.EDU
    Address:  128.122.253.37

    Non-authoritative answer:
    Name:    patah.belnet.be
    Address:  2001:6a8:3c80:0:a00:20ff:fea2:8dbc
```

```
3$ /usr/sbin/nslookup -type=PTR 2001:6a8:3c80:0:a00:20ff:fea2:8dbc | \
    awk 'NR == 5 {print $NF}'
patah.belnet.be
```

You have to type the address backward for dig.

```
4$ /usr/sbin/dig -t any \
c.b.d.8.2.a.e.f.f.f.0.2.0.0.a.0.0.0.0.0.0.8.c.3.8.a.6.0.1.0.0.2.ip6.arpa | \
awk '/^;; ANSWER SECTION:$/,/^$/' | \
fold -s -80                             fold at last space before column 80
;; ANSWER SECTION:
c.b.d.8.2.a.e.f.f.f.0.2.0.0.a.0.0.0.0.0.0.8.c.3.8.a.6.0.1.0.0.2.ip6.arpa.
2d23h52m42s IN PTR  patah.belnet.be.
```

If the first three network bits are **001**, there are 64 host bits. (This is called an “aggregatable unicast address”; we’ll talk about aggregation when we do CIDR.) If the host bits are derived from the 48-bit Ethernet address of the host, they are derived by inserting the 16 bits **0xFFFE** after the 24-bit Organizationally Unique Identifier (OIU) at the start of the Ethernet address, and flipping bit 17 of the OIU. This is the “universal/local” bit; 1 is universal, 0 is local. See RFC 3513, pp. 20−21.

For example, the host bits of **patah.belnet.be** are derived from the Ethernet address **08:00:20:a2:8d:bc**. The OIU of this address is **08:00:20**. Looking this OUI up at

```
    http://standards.ieee.org/regauth/oui/index.html
```

reveals that it was manufactured by Sun. Flipping bit 17 from **0** to **1** changes

```
    08:00:20   00001000 00000000 00100000 to
    0a:00:20   00001010 00000000 00100000
```

An “IPv4-mapped IPv6 address” is a 128-bit IPv6 address. But data sent to it will be carried by IPv4 datagrams. This lets an IPv6 host talk to an IPv4 host. An IPv4-mapped IPv6 address is 80 zeroes, followed by 16 ones, followed by the 32-bit IPv4 address. For example, the IPv4-mapped IPv6 address of i5.nyu.edu is **::ffff:807a:fd98**, ususally written as **::ffff:128.122.253.152**.

If you have a host that can receive IPv6, but which is accessible only via an IPv4 router, give the host an “IPv4-compatible IPv6 address”. For example, the IPv4-compatible IPv6 address of i5.nyu.edu would be **::807a:fd98**, ususally written as **::128.122.253.152**. An IPv6 datagram sent to an IPv4-compatible address will be encapsulated in one or more IPv4 datagrams before it gets to the router. This is called *tunneling.*

unspecified address: I don’t know who I am

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
  0000      0000      0000      0000      0000      0000      0000      0000
```

loopback address

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
  0000      0000      0000      0000      0000      0000      0000      0001
```

64 network bits **patah.belnet.be** *64 host bits (interface ID)*

```
0010000000000001000001101010100000111100100000000000000000000000|0000101000000000001000001111111111111110101000101000110110111100
  2001      06a8      3c80      0000      0a00      20ff      fea2      8dbc
```

IPv4-mapped IPv6 address for **i5.nyu.edu**

```
000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111|10000000011110101111110110011000
  0000      0000      0000      0000      0000      ffff      807a      fd98
```

IPv4-compatible IPv6 address for **i5.nyu.edu**

```
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000|10000000011110101111110110011000
  0000      0000      0000      0000      0000      0000      807a      fd98
```

Get an IPv4-mapped IPv6 address

The usual way to get an IPv4-mapped IPv6 address is to set

```
1$ echo $RES_OPTIONS                    resolver options
2$ export RES_OPTIONS=inet6
3$ echo $RES_OPTIONS
```

and call the function **gethostbyname2**. But we don’t have **gethostbyname2s**, and **resolv.conf**(4) says this has no effect on **gethostbyname**. Instead, we have to call **getipnodebyname**. The **AI_DEFAULT** in line 19 gets us, among other things, IPv4-mapped IPv6 addresses.

—On the Web at
http://i5.nyu.edu/˜mm64/x52.9547/src/getipnodebyname.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <netdb.h>   /* for gethostbyname */
 4
 5 extern int errno;    /* for inet_ntop */
 6
 7 int main(int argc, char **argv)
 8 {
 9     struct hostent *entry;
10     char **p;
11     char buffer[INET6_ADDRSTRLEN];
12     int error;
13
14     if (argc != 2) {
15         fprintf(stderr, "%s: need hostname\n", argv[0]);
16         return 1;
17     }
18
19     entry = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &error);
```

```
    if (entry == NULL) {
        fprintf(stderr, "%s: error %d from getipnodebyname\n",
            argv[0], error);
        return 2;
    }

    switch (entry->h_addrtype) {
    case AF_INET:
    case AF_INET6:
        for (p = entry->h_addr_list; *p != NULL; ++p) {
            if (inet_ntop(entry->h_addrtype, *p,
                buffer, sizeof buffer) == NULL) {

                perror(argv[0]);
                return 3;
            }
            printf("%s\n", buffer);
        }
        break;

    default:
        fprintf(stderr, "%s: unknown address type %d\n",
            argv[0], entry->h_addrtype);
        return 4;
    }

    return EXIT_SUCCESS;
}
```

```
4$ cd ~mm64/public_html/x52.9547/src
5$ gcc -o ~/bin/getipnodebyname getipnodebyname.c -lsocket -lnsl
```

i5.nyu.edu has no IPv6 address, so we get an IPv4-mapped IPv6 address instead. One group of multiple consecutive zeroes can be written as a double colon.

```
6$ getipnodebyname i5.nyu.edu
::ffff:128.122.253.152
```

Textbook

TCP/IP Network Administration, Third Edition by Craig Hunt; O’Reilly, 2002; ISBN 0-596-00297-1; $44.95.

```
http://www.oreilly.com/catalog/tcp3/
http://www.craighunt.com/
```

It’s $44.95 in aisle 9 downstairs at the NYU Main Bookstore (not the NYU Computer Bookstore), 18 Washington Place, (212) 998-4667, `http://www.bookc.nyu.edu/main/`.

Or you can get the *Networking CD Bookshelf, Version 2.0;* O’Reilly, 2002; ISBN 0-596-00334-x; $119.95.

```
http://www.oreilly.com/catalog/netcd2/
```

Computer bookstores in Manhattan

Computer Bookworks (212) 385-1616
78 Reade Street (near City Hall), between Broadway and Church Street
bookman3@mindspring.com

Barnes and Noble (212) 807-0099
http://www.barnesandnoble.com/

Contact information

Home page for this course:	**http://i5.nyu.edu/~mm64/x52.9547/**
Mark Meretzky’s email address:	**mark.meretzky@nyu.edu**
Mark Meretzky’s home page:	**http://i5.nyu.edu/~mm64/**

The system administrator’s address is **comment@i5.nyu.edu**. For the NYU computer help desk, send email to **its.clientservices@nyu.edu**, or call (212) 998-3333, or visit the ITS Client Services Center at **http://www.nyu.edu/its/helpdesk.html**. For problems with computer accounts and passwords, send email to the accounts office at **its.clientservices@nyu.edu** or call (212) 998-3333, or visit **http://www.nyu.edu/its/accounts/stuaccts.html**. For information about grades, incompletes, and NYU courses, including courses which I will teach next semester, call the School of Continuing and Professional Studies at (212) 998-7190. To contact me after the course is over, please send me email—don’t phone.

Homework, exams, grades

Your grade will depend on the homework you hand in to me on paper between the start of the course and 6:00 p.m. on the last class of the course. There will be no midterm or final. You get only one chance to hand in each problem. Hand in only the ones I assign in class, not all the problems in the Handouts.

Each assignment will be due one class after it is assigned. I can’t predict when I will assign each assignment, since it depends on how fast the class goes. The class web page will list when each assignment is due. If you have handed in little or no work during the semester, it will not help your grade if you hand in all or most of it late at the end of the semester.

I will give you the answer to every assignment on the day it is due. You therefore get no credit for homework that I receive after that date. If you will be absent on the due date, hand it in early or mail it to me so that I receive it early:

Mark Meretzky
care of Joanne Davis
NYU School of Continuing and Professional Studies
Technology Division Administration
10 Astor Place, room 505D (between Broadway and Lafayette Street; closed after 6:00 p.m.)
New York, NY 10003−6935

Put your real name, course number, and section number on the homework. Do not email me your homework. If you miss a class, send someone to tape it or take notes, and to drop and pick up your homework.

I will return each assignment to you one class after you give it to me, except for homework which you give me on or after the last class. I will give you back that homework only if you give me a self-addressed stamped envelope.

I will not give a grade to each individual homework, but I will correct every mistake you make. If there are no corrections, you did the homework perfectly. The only grade you will receive will be the one you get for the entire course. There is no way to predict this grade before the end of the course, since it partially depends on how well everyone else in the class does. You may also be penalized for gross absenteeism.

Collaboration

To collaborate with one or two other people, you may collectively hand in one copy of every assignment with the names of the two or three authors. You must stay with the same partner(s) throughout the semester, and you will all receive the same grade. In the real world you will program with other people, so I encourage you to do so now.

You must do all your own work with no help from anyone except your partner(s), if any.

You will fail the course if I receive two copies of the same work from people who are not partners, or work on which you were helped by a person who is not your partner. After you're caught, it is too late to make the other person your partner. You will fail the course if you hand in copies of my answers, or anybody else's answers.

The end of the course

I will not tell you your grade. I always mail the grades to NYU immediately after the last class of the course, or the day I receive the garding sheet from NYU, whichever comes last. I don't know what NYU will do with the grades or how long they will take to make them available to you.

`david.finney@nyu.edu`, Dean of SCPS, says ''Incomplete grades should be given only in rare circumstances where a student has been able to complete *nearly* all of the course assignments by the end of the semester.'' Some students request an Incomplete just to extend their computer account.

To extend your i5.nyu.edu account if you have requested an Incomplete, fill out form ITS 775 at

`http://www.nyu.edu/its/accounts/forms/request.extension.pdf`

and bring it to me by the last day of class. After I sign it, take it to the address at the top of the form. Do not leave form ITS 775 with me.

To complete your incomplete, mail the ''To complete an Incomplete'' project in the class web page to me at the above address. Also include xeroxes of all the homework I returned to you during the semester, showing my comments. Include your full name, social security number or NYU ID number, email address, the course and section number, and the year and semester when you took the course. Do not email me your late homework.

Computer labs at NYU:
`http://www.nyu.edu/its/labs/`

(212) 998-3409	Room LC-8 Tisch Hall, two flights down
PC's	40 West 4th Street at Greene Street
printers:	none accessible via the i5.nyu.edu **`lpr`** command
(212) 998-3457	14 Washington Place, one flight down
PC's	between Greene and Mercer Streets
printer:	**`wppc-hp9050-1`**
(212) 998-3421	Education Building, second floor
Macs	35 West 4th Street at Greene Street
printer:	**`ed-hp8150dn-1`**
(212) 998-3504	North Dorm, two flights down
PC's and Macs	75 Third Avenue (southeast corner of Third Avenue & 12th Street)
printer:	**`nd-hp9050dn-1`**

To get your plastic, magnetic NYU ID card, see

`http://www.nyu.edu/nyucard/`

Your i5.nyu.edu account

Our computer is Sun 250 server running Solaris 9 (SunOS 5.9). Its Internet hostname is **i5.nyu.edu**, its IP version 4 address is **128.122.253.152**, and its Ethernet address is **8:0:20:c9:a0:9**.

Your i5.nyu.edu login name is listed at

```
http://i5.nyu.edu/˜mm64/common/students.html
```

It is the same as your NYU NetID used by your NYU DIAL or NYU Home account. It will be two or three *lowercase* letters (your initials) followed by one or more digits. In these Handouts, we'll assume your login name is **abc1234**.

Your i5.nyu.edu secret password is the same as your central NYU single sign-on password used by your NYU DIAL or NYU Home account. In these Handouts, we'll assume your password is **Bacall8?**.

Before using your login name and password for the first time, register at **http://start.nyu.edu/**. First time i5.nyu.edu users must leave the password field blank as they have not yet set their password. They will then be prompted to enter their social security number and birth date.

To change your password to a more colorful one, e.g., **bogart!** or **Bacall8?**, go back to **http://start.nyu.edu/**.

The "secure shell" ssh

NYU will be your access provider if you don't already have one. Go to one of the computer labs and get the ITS NYU-NET CD-ROM containing the communications software and installation documentation. See **http://www.nyu.edu/its/faq/**. Your NetID is the same as your login name.

To log into **i5.nyu.edu**, you have to run the a program that speaks the "secure shell protocol". One example is a program named **PuTTY** or **putty.exe**. If you don't already have it, get it from

```
http://www.chiark.greenend.org.uk/˜sgtatham/putty/
```

(1) On Windows, run **putty.exe**. A window named **PuTTY Configuration** will appear. Type **i5.nyu.edu** as the host name, and select the radio button for the protocol **SSH**. The port number should be 22. Then press **Open**. Dismiss any **PuTTy Security Alert** window that may appear.

```
login as: abc1234
abc1234@i5.nyu.edu's password: Bacall8?
```

(2) From any other Unix host, you can get to **i5.nyu.edu** by running the program **ssh**. If you don't already have it, here is is NYU's page about where to get it:

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

On Mac OSX, for example, lauch the **Terminal** application to get a Unix shell window. Pull down the **Font** menu and select a pleasant font. Then give the command

```
ssh abc1234@i5.nyu.edu
```

(3) On other Macs, launch the application **MacSSH Telnet**. A window named **Open Connection...** will appear. Type **i5.nyu.edu** as the host name, and check the checkbox for **Secure Shell**. Then press **Connect**.

If it doesn't work, try again but say **128.122.253.152** instead of **i5.nyu.edu**.

After logging in

When you are finished logging in, you will see the prompt. To verify that you are really logged in, run simple programs such as

```
1$ date                                Press RETURN after each command line.
2$ cal 12 2005                         Need space before each command line argument.
```

Log out

```
1$ exit                                or logout on other systems
```

If the terminal window is still open, pull down the **File** menu and select **Exit** or **Quit** to close it.

What can go wrong: a guide to the special keys

(1) If you accidentally type **control-z**, it will say **Stopped**. Type

```
1$ fg                     Bring the most recently stopped program back into the foreground.
```

to start things up again. If it says **You have stopped jobs** when you try to log out, type **fg** to give your stopped job a chance to finish. Repeat if necessary.

(2) Press **backspace**, **delete**, or ← to erase the last character typed. If your ← or backspace key doesn’t work, type

```
2$ stty erase ←
```

(with a space before the ←) and press **RETURN**. As a last resort, see if **control-h** will backspace.

(3) To kill a long program, type **control-c** on a terminal or a PC. You may have to type it several times.

(4) **Control-s** will freeze the screen; unfreeze it with **control-q**. Similarly, **Hold Screen** on a terminal or **Scroll Lock** on a PC will freeze the screen. Unfreeze it with another **Hold Screen** or **Scroll Lock**.

(5) Never press **Caps Lock** in Unix: almost everything we type will be lowercase. Don’t confuse
(5a) the lowercase letter **l**, the uppercase letter **I**, and the digit **1**
(5b) the lowercase letter **o**, the uppercase letter **O**, and the digit **0**
(5c) the diagonal slash **/** and the backslash \
(5d) the single quote **’**, the double quote **"**, and the back quote **‘**
(5e) the dash **-**, the underscore **_**, and the tilde **˜**
(5f) the left parentheses **(**, the left curly brace **{**, and the left square bracket **[**
(5g) the right parentheses **)**, the right curly brace **}**, and the right square bracket **]**
(5h) the vertical bar (pipe symbol) **|**, the colon **:**, and the exclamation point **!**
(5i) any uppercase letter and the corresponding lowercase letter.

□