Fall 2005 Handout 2

Ethernet frames

The longest possible Ethernet frame holds only 1500 bytes of data because transceiver memory was expensive in 1978. There's also a 14-byte header and a 4-byte trailer, for a total of 1518 bytes.

The shortest possible Ethernet frame is 64 bytes, even if it holds no data. The frame must be long enough so that the time to transmit the frame is longer than the total round-trip time between any two points on the Ethernet. This ensures that the frame will still be in the process of being transmitted if word of a collision comes back to the sender.

The IEEE 802.3 Ethernet specification says that the maximum distance between two computers on Ethernet is 2500 meters = 2.5×10^3 meters, so the maximum round trip is 5×10^3 meters. The speed of light in a vacuum (c) is

186,282.4 miles per second =
299,792.5 kilometers per second =
299,792,500 meters per second =
2.997925×10^8 meters per second.

Let's round it to 3×10^8 meters per second. **units** always gives us a pair of reciprocals:

```
1$ units
you have: inch
you want: centimeter
     * 2.540000e+00
     / 3.937008e-01
you have: inch2                 square inches
you want: centimeter2           square centimeters
     * 6.451600e+00
     / 1.550003e-01
you have: c                     speed of light
you want: mile / second
     * 1.862824e+05
     / 5.368194e-06
you have: c
you want: meter / second
     * 2.997925e+08
     / 3.335641e-09
control-d
2$
```

The speed of electricity in a wire is about two-thirds the speed of light in a vacuum: 2×10^8 meters per second. Therefore a round trip of 5×10^3 meters should take

$$\frac{5 \times 10^3 \text{ meters}}{2 \times 10^8 \text{ meters per second}} = \frac{5}{2} \times 10^{-5} \text{ seconds}$$

In fact, the round trip can take up to twice as long (up to 5×10^{-5} seconds) because the Ethernet cable could have up to four repeaters, one every 500 meters, which slow down the signal. A microsecond (μsec, with the lowercase Greek letter mu) is a millionth of a second (10^{-6} seconds), so 5×10^{-5} seconds = 50×10^{-6} seconds = 50 μsec.

All rights reserved ©2005 Mark Meretzky

A 10-Mbps Ethernet transmits 10 megabits per second = 10 million bits per second = 10^7 bits per second. We would therefore need to transmit the following number of bytes to occupy the time needed for a round trip to the farthest computer.

$$10^7 \text{ bits per second} \times 5 \times 10^{-5} \text{ seconds} = 5 \times 10^2 \text{ bits} = \frac{5}{8} \times 10^2 \text{ bytes} = 62\tfrac{1}{2} \text{ bytes}$$

For safety, it’s rounded up to 64.

▼ Homework 2.1: Gigabit Ethernet

Gigabit Ethernet allows the maximum distance between two computers to be only 200 meters, so the maximum round trip is 400 meters. As above, let’s assume that there is hardware that cause the trip to take up to twice as long as we would theoretically expect. Gigabit Ethernet transmits a billion bits per second = 10^9 bits per second. What would the shortest possible Gigabit Ethernet frame length have to be to ensure that the frame will still be in the process of being transmitted if word of a collision comes back to the sender? For safety, the minimum frame length was rounded up to 512, so expect your answer to be slightly less than 512. The extra bytes are padded on by the hardware (as “carrier extension”); the software never sees them.

Ethernet addresses

For Ethernet, see Charles E. Spurgeon’s

```
http://www.ethermanage.com/ethernet/
http://www.oreilly.com/catalog/enettdg/
```

An Ethernet address is an example of a MAC address (“Medium Access Control”, p. 136).

An Ethernet address is 48 bits or six octets:

```
1$ cat -n /usr/include/sys/ethernet.h | sed -n '19p;22,27p'
    19  #define ETHERADDRL   (6)      /* ethernet address length in octets */
    22  /*
    23   * Ethernet address - 6 octets
    24   */
    25  struct  ether_addr {
    26          uchar_t  ether_addr_octet[ETHERADDRL];
    27  };

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

The “Address Resolution Protocol” program **arp** shows the Ethernet address of a host. The address is written byte by byte in hex, often lowercase, with colons or dashes between the bytes. A leading zero of a byte is often omitted:

```
3$ /usr/sbin/arp i5.nyu.edu
i5.nyu.edu (128.122.253.152) at 8:0:20:c9:a0:9 permanent published
```

Here are the 48 bits of our Ethernet address. Bit 46 is zero to show that this address was *not* assigned to the card by the local administrator.

```
0000 1000 0000 0000 0010 0000 1100 1001 1010 0000 0000 1001
  0    8    0    0    2    0    C    9    A    0    0    9
```

 All rights reserved ©2005 Mark Meretzky

OUI: Organizationally Unique Identifier

The first 24 bits of an Ethernet address are the OUI (“Organizationally Unique Identifier”) that tells you who manufactured the Ethernet card. For example, the OUI of our host’s Ethernet card is **08:00:20**. See Spurgeon’s Ethernet book, pp. 376–377.

http://standards.ieee.org/regauth/oui/index.html	*Official. Type 6 hex digits, no colons.*
http://www.cavebear.com/CaveBear/Ethernet/vendor.html	*Unofficial, has more OUI’s.*

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

```
 1 #!/bin/ksh
 2 #Output the manufacturer who uses this organizationally unique identifier.
 3
 4 if [[ $# -ne 1 ]]
 5 then
 6     echo $0: argument must be Organizationally Unique Identifier 1>&2
 7     exit 1
 8 fi
 9
10 lynx -dump http://www.cavebear.com/CaveBear/Ethernet/vendor.html |
11 grep -i $1
```

```
1$ oui 080020
080020  Sun
```

▼ Homework 2.2: who manufactured the Ethernet interface?

Look up the OUI of the Ethernet address of a few hosts.

The Ethernet header

Each Ethernet frame begins with a 6 + 6 + 2 = 14 byte Ethernet header (lines 33–35). Each frame ends with a four-byte Frame Check Sequence (FCS) containing a checksum (line 20).

```
1$ cat -n /usr/include/sys/ethernet.h | sed -n 19,36p
    19  #define ETHERADDRL   (6)      /* ethernet address length in octets */
    20  #define ETHERFCSL    (4)      /* ethernet FCS length in octets */
    21
    22  /*
    23   * Ethernet address - 6 octets
    24   */
    25  struct  ether_addr {
    26      uchar_t  ether_addr_octet[ETHERADDRL];
    27  };
    28
    29  /*
    30   * Structure of a 10Mb/s Ethernet header.
    31   */
    32  struct  ether_header {
    33      struct  ether_addr   ether_dhost;
    34      struct  ether_addr   ether_shost;
    35      ushort_t    ether_type;
    36  };
```

 All rights reserved ©2005 Mark Meretzky

A Unix system also has macros for these numbers. Here are three of the possible values for the **ether_type** field in the above line 35:

```
2$ cat -n /usr/include/sys/ethernet.h | sed -n '41,42p;44p'
    41  #define ETHERTYPE_IP   (0x0800) /* IP protocol */
    42  #define ETHERTYPE_ARP  (0x0806) /* Addr. resolution protocol */
    44  #define ETHERTYPE_IPV6 (0x86dd) /* IPv6 */
```

An Ethernet frame can therefore carry an IP version 4 datagram, an ARP packet, or an IP version 6 datagram. The code numbers are all up in the thousands because any number ≤ decimal 1500 would be interpreted as the length of the frame rather than as the type of the payload. See Spurgeon's Ethernet book, p. 45.

```
3$ bc
obase = 16      You type this.
1500            You type this.
5DC             bc types this.
control-d
4$
```

All the possible values for the **ether_type** field are listed at

http://www.iana.org/assignments/ethernet-numbers

The ARP cache

A host remembers the Ethernet addresses of the other hosts that it has talked to in the last five minutes. You can see these addresses with the "Address Resolution Protocol" program **arp**. See pp. 43–44, 444–447.

S means "static": the superuser put this Ethernet address into the ARP cache and it stays there. The other addresses expire after five minutes. **P** means "publish": our host will respond to other hosts asking the question "what is the IP address of the machine whose Ethernet address is **08:00:20:c9:a0:09**?". **M** is for multicasting. Handout 1, p. 22, explains why **i5.nyu.edu** is the only lowercase name.

```
1$ /usr/sbin/arp -a | head | cat -n             -a for "all"
 1
 2  Net to Media Table: IPv4
 3  Device   IP Address               Mask      Flags   Phys Addr
 4  ------ -------------------- --------------- ----- ---------------
 5  ge0    WWITSGW-VLAN-13.NET.NYU.EDU 255.255.255.255       00:d0:04:a4:10:0a
 6  ge0    10.88.103.5          255.255.255.255       00:0d:60:f6:b3:fe
 7  ge0    10.88.102.5          255.255.255.255       00:0d:60:f6:c4:6e
 8  ge0    10.89.0.1            255.255.255.255       00:0d:60:6f:32:63
 9  ge0    IONADMIN.FAS.NYU.EDU 255.255.255.255       00:06:5b:f3:f8:df
10  ge0    IONDATA.FAS.NYU.EDU  255.255.255.255       00:06:5b:f3:f2:b7
```

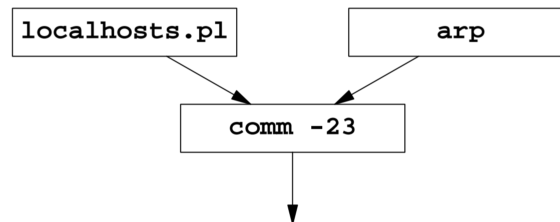
 All rights reserved ©2005 Mark Meretzky

```
2$ /usr/sbin/arp -an | head | cat -n               -n  for "numeric"
 1
 2  Net to Media Table: IPv4
 3  Device   IP Address               Mask      Flags   Phys Addr
 4  ------ -------------------- --------------- ----- ---------------
 5  ge0    128.122.253.129      255.255.255.255       00:d0:04:a4:10:0a
 6  ge0    10.88.103.5          255.255.255.255       00:0d:60:f6:b3:fe
 7  ge0    10.88.102.5          255.255.255.255       00:0d:60:f6:c4:6e
 8  ge0    10.89.0.1            255.255.255.255       00:0d:60:6f:32:63
 9  ge0    128.122.253.146      255.255.255.255       00:06:5b:f3:f8:df
10  ge0    128.122.253.145      255.255.255.255       00:06:5b:f3:f2:b7
```

Add a host to the ARP cache

Only the superuser can use the **-s** option of **arp** to add a new entry to the ARP cache. But there is a way a non-superuser can modify the cache temporarily.

First, let's find every host on the local network that is *not* in the ARP cache by writing a shellscript to merge the output of **arp** with that of the **localhosts.pl** in Handout 1, pp. 26–27.



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

```
 1 #!/bin/ksh
 2 #Output the fully qualified domain name of every host on the local network
 3 #whose ethernet address is not currently in the ARP cache.
 4
 5 cd /tmp
 6
 7 ˜mm64/public_html/x52.9547/src/localhosts.pl 128.122.253.152 26 |
 8 awk '{print $2}' |
 9 tr '[a-z]' '[A-Z]' |
10 sort > $$.all
11
12 /usr/sbin/arp -a |
13 awk 'NR >= 5 {print $2}' |
14 tr '[a-z]' '[A-Z]' |
15 sort > $$.incache
16
17 comm -23 $$.all $$.incache   #hosts in $$.all but not in $$.incache
18 rm $$.all $$.incache
19 exit 0
```

 All rights reserved ©2005 Mark Meretzky

```
1$ ~mm64/public_html/x52.9547/src/notcached | cat -n | tail -5
    30  MODIYA.NYU.EDU
    31  NEWI2.NYU.EDU
    32  NYU-DA3400-1-253-128.NS.ITS.NYU.EDU
    33  OLDUTEST.ES.ITS.NYU.EDU
    34  POLARIS.ITS.NYU.EDU
```

ping one of these hosts and see if it then appears in the **arp** cache:

```
2$ /usr/sbin/ping hostname
3$ /usr/sbin/arp -a | more
```

To put the Ethernet address of *every* host into the ARP cache,

```
4$ /usr/sbin/arp -a | more
5$ /usr/sbin/ping 128.122.253.191      our network's broadcast address; Handout 1, p. 23
6$ /usr/sbin/arp -a | more
```

▼ Homework 2.3: how long will the host stay in the arp cache?

Without the **'**single quotes**'**, the question mark would be a wildcard. The shell would change the question mark into the names of all the files in the current directory with single-character names. See pp. 110–111.

```
1$ /usr/sbin/ndd /dev/arp '?' | more                    "network device driver"
?                                   (read only)
arp_cache_report                    (read only)
arp_debug                           (read and write)
arp_cleanup_interval                (read and write)
arp_publish_interval                (read and write)
arp_publish_count                   (read and write)
```

The output is in milliseconds. 1000 milliseconds is 1 second, so 300000 milliseconds is five minutes.

```
2$ /usr/sbin/ndd /dev/arp arp_cleanup_interval
300000
```

/dev/arp is actually a (symbolic link to a) character device driver:

```
3$ cd /dev
4$ ls -l arp
lrwxrwxrwx   1 root     other         27 Aug 18  2003 arp -> ../devices/pseudo/arp@0:arp
```

```
5$ cd ../devices/pseudo
6$ ls -l arp@0:arp
crw-rw-rw-   1 root     sys       44,  0 Aug 18  2003 arp@0:arp

7$ awk '$2 == 44' /etc/name_to_major
arp 44

8$ cd /kernel/drv
9$ ls -li arp                    The device driver has 2 hard links. See its inode number.
       281 -rwxr-xr-x   2 root     sys        52136 Apr 24  2003 arp

10$ find /kernel -inum 281 -print 2> /dev/null
/kernel/drv/arp
/kernel/strmod/arp
```

 All rights reserved ©2005 Mark Meretzky

```
11$ ls -li /kernel/drv/arp /kernel/strmod/arp              STREAMS module
281 -rwxr-xr-x   2 root     sys        52136 Apr 24  2003 /kernel/drv/arp
281 -rwxr-xr-x   2 root     sys        52136 Apr 24  2003 /kernel/strmod/arp

    12$ awk '$1 ~ /^name=/' /kernel/drv/arp.conf        See driver.conf(4).
    name="arp" parent="pseudo" instance=0;
```

The ARP header: Address Resolution Protocol

An Ethernet address is a "hardware address"; an IPv4 address is a "protocol address". Line 52 is a possible value for the **ar_hrd** field in line 51. Lines 57–60 are four possible values for the **ar_op** field in line 56.

```
1$ cat -n /usr/include/netinet/arp.h | sed -n '50,60p;73p'
    50  struct  arphdr {
    51      ushort_t ar_hrd; /* format of hardware address */
    52  #define ARPHRD_ETHER     1   /* ethernet hardware address */
    53      ushort_t ar_pro; /* format of protocol address */
    54      uchar_t  ar_hln;     /* length of hardware address */
    55      uchar_t  ar_pln;     /* length of protocol address */
    56      ushort_t ar_op;      /* one of: */
    57  #define ARPOP_REQUEST    1   /* request to resolve address */
    58  #define ARPOP_REPLY  2   /* response to previous request */
    59  #define REVARP_REQUEST   3   /* Reverse ARP request */
    60  #define REVARP_REPLY 4   /* Reverse ARP reply */
    73  };
```

The structure in line 83 is the above structure.

```
2$ cat -n /usr/include/netinet/arp.h | sed -n 82,88p
    82  struct  ether_arp {
    83      struct  arphdr ea_hdr;       /* fixed-size header */
    84      struct ether_addr arp_sha;   /* sender hardware address */
    85      uchar_t arp_spa[4];      /* sender protocol address */
    86      struct ether_addr arp_tha;   /* target hardware address */
    87      uchar_t arp_tpa[4];      /* target protocol address */
    88  };
```

The IP version 4 header: Internet Protocol

The IP version 4 header is in the diagrams on pp. 14 and 679–680. On our machine, the macro **_BIT_FIELDS_LTOH** ("low to high") in line 45 is undefined. This puts the version number in line 49 into the high-order nibble of the first byte of the header.

All rights reserved ©2005 Mark Meretzky

```
1$ cat -n /usr/include/netinet/ip.h | sed -n '37,62p' | more
    37  /*
    38   * Structure of an internet header, naked of options.
    39   *
    40   * We declare ip_len and ip_off to be short, rather than ushort_t
    41   * pragmatically since otherwise unsigned comparisons can result
    42   * against negative integers quite easily, and fail in subtle ways.
    43   */
    44  struct ip {
    45  #ifdef _BIT_FIELDS_LTOH
    46      uchar_t ip_hl:4,     /* header length */
    47          ip_v:4;          /* version */
    48  #else
    49      uchar_t ip_v:4,          /* version */
    50          ip_hl:4;     /* header length */
    51  #endif
    52      uchar_t ip_tos;          /* type of service */
    53      short   ip_len;          /* total length */
    54      ushort_t ip_id;          /* identification */
    55      short   ip_off;          /* fragment offset field */
    56  #define IP_DF 0x4000         /* dont fragment flag */
    57  #define IP_MF 0x2000         /* more fragments flag */
    58      uchar_t ip_ttl;          /* time to live */
    59      uchar_t ip_p;            /* protocol */
    60      ushort_t ip_sum;     /* checksum */
    61      struct  in_addr ip_src, ip_dst;  /* source and dest address */
    62  };
```

The "protocol" field in the above line 59 plays the rôle of the **ether_type** field of an Ethernet frame. It tells what the IP datagram is carrying by specifying which operating system functions it should be delivered do. The possible values for this field are listed at

http://www.iana.org/assignments/protocol-numbers

Here are macros for four possible values for the protocol field.

```
2$ awk '$1 ~ /^(icmp|tcp|udp|ospf)$/' /etc/protocols
icmp    1   ICMP    # internet control message protocol
tcp     6   TCP     # transmission control protocol
udp     17  UDP     # user datagram protocol
ospf    89  OSPFIGP     # Open Shortest Path First

3$ man -s 4 protocols
```

An IP datagram can therefore carry a TCP segment, a UDP packet, an ICMP packet, or an OSPF packet.

The ICMP header: Internet Control Message Protocol

The ICMP header is in the diagram on pp. 683–684:

 All rights reserved ©2005 Mark Meretzky

```
1$ cat -n /usr/include/netinet/ip_icmp.h | sed -n '54,67p;81p;85p;108p'
    54  /*
    55   * Structure of an icmp header.
    56   */
    57  struct icmp {
    58      uchar_t  icmp_type;       /* type of message, see below */
    59      uchar_t  icmp_code;       /* type sub code */
    60      uint16_t icmp_cksum;      /* ones complement cksum of struct */
    61      union {
    62          uchar_t ih_pptr;      /* ICMP_PARAMPROB */
    63          struct in_addr ih_gwaddr;     /* ICMP_REDIRECT */
    64          struct ih_idseq {
    65              uint16_t icd_id;
    66              uint16_t icd_seq;
    67          } ih_idseq;
    81      } icmp_hun;
    85 #define icmp_seq icmp_hun.ih_idseq.icd_seq
   108  };
```

The possible values for the `icmp_type` field in the above line 58 are at

`http://www.iana.org/assignments/icmp-parameters`

Here are macros for four of the possible values for `icmp_type`.

```
2$ cat -n /usr/include/netinet/ip_icmp.h |
sed -n '128,129p;134p;146p;152p;157,159p'
   128  #define ICMP_ECHOREPLY       0        /* echo reply */
   129  #define ICMP_UNREACH     3        /* dest unreachable, codes: */
   134  #define      ICMP_UNREACH_NEEDFRAG        4    /* IP_DF caused drop */
   146  #define ICMP_SOURCEQUENCH    4        /* packet lost, slow down */
   152  #define ICMP_ECHO        8        /* echo service */
   157  #define ICMP_TIMXCEED        11       /* time exceeded, code: */
   158  #define      ICMP_TIMXCEED_INTRANS    0        /* ttl==0 in transit */
   159  #define      ICMP_TIMXCEED_REASS  1        /* ttl==0 in reass */
```

If the value of the `icmp_type` in the above line 58 is the `ICMP_UNREACH` in line 129, then the `ICMP_UNREACH_NEEDFRAG` in line 134 is a possible value for the `icmp_code` field in line 59. If the value of `icmp_type` is the `ICMP_TIMXCEED` in line 157, then the line 158–159 are possible values for the `icmp_code` field.

To prevent Linux from responding to ICMP echo requests, including those to `localhost`,

```
3$ echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

`echo 0` to go back to the default.

Four dialogs

Dialog 1: An ICMP dialog with a preliminary exchange in ARP

Here are two hosts on the same local network. The netmask is `255.255.255.0`: 24 network bits, followed by 8 host bits. I typed the following `snoop` and `ping` commands on `phoenix.wolfnet.com`, where I'm the superuser.

 All rights reserved ©2005 Mark Meretzky

fully qualified domain name	*IP address*	*Ethernet address*
phoenix.wolfnet.com	**192.168.0.5**	**00:d0:09:ff:57:05**
smoke.wolfnet.com	**192.168.0.4**	**00:02:55:56:e0:c0**

We'll eavesdrop on the network traffic with a "packet sniffer". Solaris machines such as **i5.nyu.edu** and **phoenix.wolfnet.com** have **snoop**; see pp. 471–478. The manual "page" for **snoop** is really nine pages long. Print it at at

http://i5.nyu.edu/~mm64/man/

Linux has Van L. Jacobson's **tcpdump**; see **http://www.tcpdump.com/**. Other versions of Unix have the annoying **nettl**. For PC 's, see **http://www.ethereal.com/**.

Only the superuser can run a packet sniffer or give you permission to do so. See **http://www.sudo.ws/**.

By default, **snoop** will write to the standard output. The option **-o** will make it write to a file instead.

```
1$ snoop -o arp_icmp.snoop arp or host smoke.wolfnet.com &
[1] 20680                       The shell displays the PID number of a background process.

2$ ping 192.168.0.4             The PID of this ping is 20681 (carried in packets 3 and 4).
192.168.0.4 is alive

3$ kill 20680                   Kill the snoop which was running in the background.
```

I then moves the resulting file **arp_icmp.snoop** to the **~mm64/public_html/x52.9547/src/snoop** directory of i5.nyu.edu , where you can examine it.

Packet 1

By default, **snoop** will read from a network interface; only the superuser can do this. The option **-i** will make it read from a file instead; a non-superuser can do this. **-ta** is absolute (as opposed to relative) time; lowercase **-v** is the highest level of verbosity.

The first two packets are the ARP dialog triggered by any attempt to send information to a host whose Ethernet address is not in the ARP cache of our host. When we gave the above **ping** command, the Ethernet address of the host **smoke.wolfnet.com** (**192.168.0.4**) was not in the ARP cache of our host **phoenix.wolfnet.com** (**192.168.0.5**).

The source in line 6 is the Ethernet address of my machine **phoenix.wolfnet.com**. The destination in line 5 is the Ethernet broadcast address: 48 bytes of all ones.

The code number in line 7 says that this Ethernet frame is carrying an ARP packet. It is the macro **ETHERTYPE_ARP** in lines 35 and 40 of the header file **ethernet.h** in Handout 2, pp. 3–4.

RFC 826 defines the format of the ARP packet in lines 9–20. The code number in line 11 says that this ARP packet carries questions or answers about converting to or from Ethernet addresses. It is the macro **ARPHDR_ETHER** in lines 51–52 of the header file **arp.h** in Handout 2, p. 6. The code number in line 12 says that this ARP packet carries questions or answers about converting to or from IP version 4 addresses. It is the macro **ETHERTYPE_IP** in line 39 of **ethernet.h**.

The code number in line 15 says that we're requesting help from anyone who wants to respond. It is the **ARPOP_REQUEST** in lines 56 and 57 of **arp.h**. Let's hope that a host on the local network will be able to answer our question in line 18.

Line 4 says that the Ethernet frame is 42 bytes. The first 6 + 6 + 2 = 14 of these are the three fields of the Ethernet header in lines 5–7. This leaves room for 42 −14 = 28 bytes of cargo. The nine fields of the ARP packet in lines 11–19 occupy all 28 bytes: 2 + 2 + 1 + 1 + 2 + 6 + 6 + 4 + 4 = 28.

 All rights reserved ©2005 Mark Meretzky

```
1$ cd ~mm64/public_html/x52.9547/src/snoop
2$ /usr/sbin/snoop -i arp_icmp.snoop -ta -p 1,1 -v | cat -n
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 1 arrived at 17:14:45.05
     4  ETHER:  Packet size = 42 bytes
     5  ETHER:  Destination = ff:ff:ff:ff:ff:ff, (broadcast)
     6  ETHER:  Source      = 0:d0:9:ff:57:5,
     7  ETHER:  Ethertype = 0806 (ARP)
     8  ETHER:
     9  ARP:  ----- ARP/RARP Frame -----
    10  ARP:
    11  ARP:  Hardware type = 1
    12  ARP:  Protocol type = 0800 (IP)
    13  ARP:  Length of hardware address = 6 bytes
    14  ARP:  Length of protocol address = 4 bytes
    15  ARP:  Opcode 1 (ARP Request)
    16  ARP:  Sender's hardware address = 0:d0:9:ff:57:5
    17  ARP:  Sender's protocol address = 192.168.0.5, 192.168.0.5
    18  ARP:  Target hardware address = ?
    19  ARP:  Target protocol address = 192.168.0.4, 192.168.0.4
    20  ARP:
```

Packet 2

The code number in line 15 is the macro **ARPOP_REPLY** in lines 56 and 58 of **arp.h** in Handout 2, p. 7. Line 16 is the reply to the question in packet 1: the Ethernet address of **smoke.wolfnet.com**.

Once again, the ARP packet occupies 28 bytes. Adding the 14-byte Ethernet header, 14 + 28 = 42 bytes. But line 4 shows that the Ethernet frame was padded out to 64 bytes, the minimum frame length necessary for Ethernet to detect collisions. (The CD in the celebrated CSMA/CD is ‘‘Collision Detection’’.) See pp. 24, 50 in the Spurgeon Ethernet book. Why didn't we see the above Packet 1 padded out to 64 bytes?

 All rights reserved ©2005 Mark Meretzky

```
1$ snoop -i arp_icmp.snoop -p 2,2 -ta -v | cat -n
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 2 arrived at 17:14:45.05
     4  ETHER:  Packet size = 64 bytes
     5  ETHER:  Destination = 0:d0:9:ff:57:5,
     6  ETHER:  Source      = 0:2:55:56:e0:c0,
     7  ETHER:  Ethertype = 0806 (ARP)
     8  ETHER:
     9  ARP:  ----- ARP/RARP Frame -----
    10  ARP:
    11  ARP:  Hardware type = 1
    12  ARP:  Protocol type = 0800 (IP)
    13  ARP:  Length of hardware address = 6 bytes
    14  ARP:  Length of protocol address = 4 bytes
    15  ARP:  Opcode 2 (ARP Reply)
    16  ARP:  Sender's hardware address = 0:2:55:56:e0:c0
    17  ARP:  Sender's protocol address = 192.168.0.4, 192.168.0.4
    18  ARP:  Target hardware address = 0:d0:9:ff:57:5
    19  ARP:  Target protocol address = 192.168.0.5, 192.168.0.5
    20  ARP:
```

The RARP protocol lets a host broadcast the opposite question: here's my Ethernet address—could someone please tell me my IP address? It has been superseded by BOOTP, which has been superseded by DHCP.

Packet 3

At this point **phoenix.wolfnet.com** knows that the Ethernet address of **smoke.wolfnet.com** is **00:02:55:56:e0:c0**. The **ping** on **phoenix** can therefore send an echo request. The code number in line 7 is the macro **ETHERTYPE_IP** in lines 35 and 41 of the header file **ethernet.h** in Handout 2, pp. 3–4. The analogous number in line 27 is the code number for ICMP in line 59 of the header file **ip.h** and in the **/etc/protocols** file in Handout 2, p. 8. The code number in line 35 is the macro **ICMP_ECHO** in lines 58 and 152 of the header file **ip_icmp.h** in Handout 2, p. 9.

The Ethernet frame occupies 98 bytes (line 4). The first 14 bytes are always the Ethernet header (= 6 + 6 + 2), leaving 84 bytes for cargo. This agrees with line 20.

The cargo of the Ethernet frame is an 84-byte IP datagram. Line 12 says that the first 20 bytes are the IP header (= 1 + 1 + 2 + 2 + 2 + 1 + 1 + 2 + 4 + 4) leaving 64 bytes for cargo. This agrees with the size of the ICMP messages in Handout 2, p. 7.

The bits for ‘‘Explicit Congestion Notification’’ in lines 18 and 19 are in RFC 3168.

Our 84-byte IP datagram had no trouble fitting into a single Ethernet frame. But a datagram larger than 1500 bytes would have to be broken up and carried in two or more Ethernet frames. This is called *fragmentation.* The identification number in line 21 would be used to reassemble the fragments: all the fragments of a datagram share the same identification number. The offset in line 25 would show the location of this fragment within the datagram. Line 24 tells if there are additional fragments after this one.

 All rights reserved ©2005 Mark Meretzky

```
1$ cd ~mm64/public_html/x52.9547/src/snoop
2$ snoop -i arp_icmp.snoop -p 3,3 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 3 arrived at 17:14:45.05
     4  ETHER:  Packet size = 98 bytes
     5  ETHER:  Destination = 0:2:55:56:e0:c0,
     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 = 84 bytes
    21  IP:   Identification = 2107
    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 = 1 (ICMP)
    28  IP:   Header checksum = f213
    29  IP:   Source address = 192.168.0.5, 192.168.0.5
    30  IP:   Destination address = 192.168.0.4, 192.168.0.4
    31  IP:   No options
    32  IP:
    33  ICMP:  ----- ICMP Header -----
    34  ICMP:
    35  ICMP:  Type = 8 (Echo request)
    36  ICMP:  Code = 0 (ID: 20681 Sequence number: 0)
    37  ICMP:  Checksum = 69f3
    38  ICMP:
```

To compute the IP header checksum **0xF213** in the above line 28, we first display the 20-byte IP header in hex by chopping off the 14-byte Ethernet header in front of it with the **-x** option:

```
3$ snoop -i arp_icmp.snoop -p 3,3 -x 14,20 | tail +3
       0: 4500 0054 083b 4000 ff01 f213 c0a8 0005    E..T.;@.........
      16: c0a8 0004                                  ....
```

Add the ten two-byte words (except the sixth, **f213**, which is the checksum itself):

 All rights reserved ©2005 Mark Meretzky

```
4$ bc
obase = 16                                  Must specify output base before input base.
ibase = 16
4500+0054+083B+4000+FF01+0000+C0A8+0005+C0A8+0004
30DE9
control-d
5$
```

The bottom 16 bits of the sum (**0DE9**) plus the top 16 bits of the sum (**0003**) is **0DEC**. The "one's complement" (i.e., bitwise not, photographic negative) of **0DEC** is the IP header checksum **F213** in the above line 28.

```
  0DE9
+ 0003
------
  0DEC == 0000 1101 1110 1100
  F213 == 1111 0010 0001 0011
```

On the destination host (in this case, **smoke**), an ICMP packet is received by the operating system as a whole, not by any particular program. Similarly, the reply packet will be received by our operating system. Our operating system may be running programs, including many copies of **ping**. How will out operating system know which **ping** it should deliver the reply to? Usually, we would need TCP or UDP for this, but a trick lets us do it with ICMP: the Unix process ID number (PID) of the **ping** program is stored in the ICMP identifier field in line 36 of the above packet and the following reply packet.

To compute the ICMP checksum **0x69F3** in the above line 37, we first display the 64-byte ICMP message in hex by chopping off the 14-byte Ethernet header and 20-byte IP header:

```
6$ snoop -i arp_icmp.snoop -p 3,3 -x 34,64 | tail +3
       0: 0800 69f3 50c9 0000 c526 d53e b7da 0000    ..i.P....&.>....
      16: 0809 0a0b 0c0d 0e0f 1011 1213 1415 1617    ................
      32: 1819 1a1b 1c1d 1e1f 2021 2223 2425 2627    ........ !"#$%&'
      48: 2829 2a2b 2c2d 2e2f 3031 3233 3435 3637    ()*+,-./01234567
```

Add the 32 two-byte words (except the second, **69f3**, which contains the checksum itself):

```
7$ bc
obase = 16
ibase = 16
0800+0000+50C9+0000+C526+D53E+B7DA+0000+\
0809+0A0B+0C0D+0E0F+1011+1213+1415+1617+\
1819+1A1B+1C1D+1E1F+2021+2223+2425+2627+\
2829+2A2B+2C2D+2E2F+3031+3233+3435+3637
59607
control-d
8$
```

The bottom 16 bits of the sum (**9607**) plus the top 16 bits of the sum (**0005**) is **960C**. The "one's complement" (i.e., bitwise not, photographic negative) of **960C** is the IP header checksum **69F3** in the above line 37.

```
  9607
+ 0005
------
  960C == 1001 0110 0000 1100
  69F3 == 0110 1001 1111 0011
```

 All rights reserved ©2005 Mark Meretzky

Packet 4

The code number in line 35 is the macro **ICMP_ECHOREPLY** in lines 58 and 128 of the header file **ip_icmp.h** in Handout 2, p. 9. Once again, the ICMP identifier field in line 36 is the PID of **ping**.

The IP datagrams emitted by **phoenix.wolfnet.com** have a time to live of 255, but those emitted by **smoke.wolfnet.com** have a time to live of only 64. See line 26 of packets 3 and 4.

In this and all the following examples, every incoming Ethernet frame is exactly 4 bytes bigger than it needs to be to hold its cargo; see lines 4 and 20. Why is that? If it's because of the four-byte Frame Check Sequence (FCS) in Handout 2, p. 3, line 20, why don't we see the FCS in the output of **snoop**?

```
1$ snoop -i arp_icmp.snoop -p 4,4 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 4 arrived at 17:14:45.05
     4  ETHER:  Packet size = 102 bytes
     5  ETHER:  Destination = 0:d0:9:ff:57:5,
     6  ETHER:  Source      = 0:2:55:56:e0:c0,
     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 = 84 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 = 64 seconds/hops
    27  IP:   Protocol = 1 (ICMP)
    28  IP:   Header checksum = f94f
    29  IP:   Source address = 192.168.0.4, 192.168.0.4
    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 = 0 (Echo reply)
    36  ICMP:  Code = 0 (ID: 20681 Sequence number: 0)
    37  ICMP:  Checksum = 71f3
    38  ICMP:
```

▼ Homework 2.4: adjust the verbosity

The first command outputs only one line per Ethernet frame, describing only the highest level packet in the frame. **C** is "call" (outgoing); **R** is "reply" (incoming). The second outputs one line for each packet in the Ethernet frame. For example, it would output three lines for the above packet 4: the Ethernet frame,

 All rights reserved ©2005 Mark Meretzky

the IP datagram inside it, and the ICMP packet inside *it*. The third is what you saw above.

```
1$ cd ~mm64/public_html/x52.9547/src/snoop
2$ snoop -i arp_icmp.snoop                       terse
  1   0.00000  192.168.0.5 -> (broadcast)  ARP C Who is 192.168.0.4, 192.168.0.4 ?
  2   0.00016  192.168.0.4 -> 192.168.0.5  ARP R 192.168.0.4, 192.168.0.4 is 0:2:55:56:
  3   0.00028  192.168.0.5 -> 192.168.0.4  ICMP Echo request (ID: 20681 Sequence number
  4   0.00015  192.168.0.4 -> 192.168.0.5  ICMP Echo reply (ID: 20681 Sequence number:

3$ snoop -i arp_icmp.snoop -V | more       verbose: uppercase -V
________________________________
  1   0.00000  192.168.0.5 -> (broadcast)  ETHER Type=0806 (ARP), size = 42 bytes
  1   0.00000  192.168.0.5 -> (broadcast)  ARP C Who is 192.168.0.4, 192.168.0.4 ?
________________________________
  2   0.00016  192.168.0.4 -> 192.168.0.5  ETHER Type=0806 (ARP), size = 64 bytes
  2   0.00016  192.168.0.4 -> 192.168.0.5  ARP R 192.168.0.4, 192.168.0.4 is 0:2:55:56:
________________________________
  3   0.00028  192.168.0.5 -> 192.168.0.4  ETHER Type=0800 (IP), size = 98 bytes
  3   0.00028  192.168.0.5 -> 192.168.0.4  IP  D=192.168.0.4 S=192.168.0.5 LEN=84, ID=2
  3   0.00028  192.168.0.5 -> 192.168.0.4  ICMP Echo request (ID: 20681 Sequence number
________________________________
  4   0.00015  192.168.0.4 -> 192.168.0.5  ETHER Type=0800 (IP), size = 102 bytes
  4   0.00015  192.168.0.4 -> 192.168.0.5  IP  D=192.168.0.5 S=192.168.0.4 LEN=84, ID=0
  4   0.00015  192.168.0.4 -> 192.168.0.5  ICMP Echo reply (ID: 20681 Sequence number:

4$ snoop -i arp_icmp.snoop -v | more  more verbose: lowercase -v
```

▲

The UDP header

The UDP header is in the diagram on p. 18. The “destination port” field in line 29 plays the rôle of the **ether_type** field of an Ethernet frame and the **ip_p** field of an IP datagram. It tells what the UDP packet is carrying by specifying the program to which it should be delivered. Ditto for the “destination port” field in line 33 of the TCP header below.

```
1$ cat -n /usr/include/netinet/udp.h | sed -n 27,32p
    27  struct udphdr {
    28      in_port_t   uh_sport;       /* source port */
    29      in_port_t   uh_dport;       /* destination port */
    30      int16_t     uh_ulen;    /* udp length */
    31      uint16_t uh_sum;        /* udp checksum */
    32  };

2$ grep 'typedef.*in_port_t;' /usr/include/netinet/in.h
typedef uint16_t in_port_t;

3$ grep 'typedef.*uint16_t;' /usr/include/sys/int_types.h
typedef unsigned short uint16_t;
```

Dialog 2: A two-packet dialog carried by UDP

Ethernet frames, IP datagrams, and ICMP packets are not addressed to any specific program on the receiving machine. They are therefore received and processed by the operating system on that machine. To send data to a specific program, you must put it into a UDP packet or TCP segment. Each UDP packet and TCP segment carries a port number.

 All rights reserved ©2005 Mark Meretzky

The specific program we will talk to is the **daytime** server. It has the shortest RFC:

```
1$ lynx -dump ftp://ftp.rfc-editor.org/in-notes/rfc867.txt | more
```

The file **/etc/services** holds the TCP and UDP port number of each server:

```
2$ awk '$1 == "daytime"' /etc/services
daytime      13/tcp
daytime      13/udp
```

The documentation for this file is at

```
3$ man -s 4 services
```

Print it at

http://i5.nyu.edu/~mm64/man/

The port numbers are also at

http://www.iana.org/assignments/port-numbers

There is no Unix program that will send and receive the UDP packet I want, so I wrote my own in C and Perl. The file **/etc/nsswitch.conf** tells the function **getservbyname** in C lines 21–22 and Perl line 7 to look up the server **daytime** in the file **/etc/services**:

```
4$ awk '$1 == "services:"' /etc/nsswitch.conf
services:   files
```

C line 57 and Perl line 7 therefore put the value 13 into the structure field **sin_port** and the variable **$port**.

In C line 24 and Perl line 13, the address family **AF_INET** means "IP version 4"; **AF_INET6** would have meant "IP version 6". **SOCK_DGRAM** means "UDP"; **SOCK_STREAM** would have meant "TCP". This program doesn't need the third argument of **socket** in C line 49 and Perl line 13, so we write a zero.

The empty string **""** in C line 60 and **''** in Perl line 14 is the empty contents of the UDP packet. The zero means no flags for out of band data, etc. See **send**(3socket).

The **recv** in C lines 88–89 and Perl line 27 might wait forever for a UDP packet that never arrives. C lines 80–85 and Perl lines 24–25 therefore first ask the operating system for a wakeup call (the "alarm signal") if the **recv** is still waiting after 15 seconds.

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

```
 1 /*
 2 Send a UDP datagram to the daytime server on another host.
 3 Display the UDP datagram that comes back.
 4 */
 5
 6 #include <stdio.h>
 7 #include <stdlib.h>
 8 #include <errno.h>
 9 #include <signal.h>
10
11 #include <netdb.h>
12 #include <sys/types.h>
13 #include <sys/socket.h>
14 #include <netinet/in.h>
15 #include <arpa/inet.h>
16
```

 All rights reserved ©2005 Mark Meretzky

```
17 void handle_sigalrm(int s);
18
19 int main(int argc, char **argv)
20 {
21     const struct servent *const service = getservbyname("daytime", "udp");
22
23     const char name[] = "aixmita1.urz.uni-heidelberg.de";
24     const sa_family_t family = AF_INET;   /* IP version 4 */
25     const struct hostent *host;
26
27     int s;        /* file descriptor for socket */
28     struct sockaddr_in address;
29     socklen_t length = sizeof address;
30
31     char buffer[100];
32     char dotted[100];
33     ssize_t n;    /* number of bytes received from socket */
34
35     if (service == NULL) {
36         fprintf(stderr, "%s: couldn't find udp service daytime\n",
37             argv[0]);
38         return 1;
39     }
40
41     host = gethostbyname(name);
42     if (host == NULL) {
43         fprintf(stderr,
44             "%s: couldn't get IP address for %s, h_errno == %d\n",
45             argv[0], name, h_errno);
46         return 2;
47     }
48
49     s = socket(family, SOCK_DGRAM, 0);
50     if (s < 0) {
51         perror(argv[0]);
52         return 3;
53     }
54
55     bzero(&address, sizeof address);
56     address.sin_family = family;
57     address.sin_port = service->s_port;
58     address.sin_addr.s_addr = *(in_addr_t *)host->h_addr_list[0];
59
60     if (sendto(s, "", 0, 0, (const struct sockaddr *)&address,
61         sizeof address) != 0) {
62         perror(argv[0]);
63         return 4;
64     }
65
66     if (getsockname(s, (struct sockaddr *)&address, &length) != 0) {
67         perror(argv[0]);
68         return 5;
69     }
70
```

 All rights reserved ©2005 Mark Meretzky

```
 71     if (inet_ntop(family, &address.sin_addr, buffer, sizeof buffer)
 72         == NULL) {
 73         perror(argv[0]);
 74         return 6;
 75     }
 76
 77     printf("I sent an empty datagram from my port %d.\n",
 78         ntohs(address.sin_port));
 79
 80     /* Remain in recvfrom for no more than 15 seconds. */
 81     if (signal(SIGALRM, handle_sigalrm) == SIG_ERR) {
 82         perror(argv[0]);
 83         return 7;
 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(argv[0]);
 93         return 8;
 94     }
 95
 96     if (inet_ntop(family, &address.sin_addr, dotted, sizeof dotted)
 97         == NULL) {
 98         perror(argv[0]);
 99         return 9;
100     }
101
102     printf("I then received a datagram containing the following %d bytes\n"
103         "from port %d of %s", n, ntohs(address.sin_port), dotted);
104
105     host = gethostbyaddr((const char *)&address.sin_addr,
106         sizeof address.sin_addr, family);
107
108     if (host != NULL) {
109         printf(" (%s)", host->h_name);
110     }
111     printf("\n%*s\n", n, buffer);
112     return EXIT_SUCCESS;
113 }
114
115 void handle_sigalrm(int s)
116 {
117     printf("The datagram I was waiting for didn't arrive in 15 seconds.\n");
118     exit(10);
119 }
```

```
5$ gcc -o ~/bin/udp udp.c -lsocket -lnsl
6$ ls -l ~/bin/udp
7$ udp
```

 All rights reserved ©2005 Mark Meretzky

—On the Web at
http://i5.nyu.edu/˜mm64/x52.9547/src/snoop/udp.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
 5 use Socket;
 6
 7 $port = getservbyname('daytime', 'udp')
 8     or die "$0: couldn't find udp service daytime";
 9
10 $name = 'aixmita1.urz.uni-heidelberg.de';
11 $ip = inet_aton($name) or die "$0: couldn't get IP address for \"$name\"";
12
13 socket(S, AF_INET, SOCK_DGRAM, 0) or die "$0: $!";
14 $n = send(S, '', 0, sockaddr_in($port, $ip));
15 defined $n or die "$0: $!";
16 if ($n != 0) {
17     die "$0: sent $n bytes instead of 0 bytes\n";
18 }
19
20 $address = getsockname(S) or die "$0: $!";   #Can't do this before the send.
21 ($port, $ip) = sockaddr_in($address);
22 print "I sent an empty datagram from my port $port.\n";
23
24 $SIG{ALRM} = 'handle_sigalrm';
25 alarm(15);
26
27 $address = recv(S, $buffer, 100, 0) or die "$0: $!";
28 ($port, $ip) = sockaddr_in($address);
29
30 print "I then received a datagram containing the following ", length($buffer),
31     " bytes\nfrom port $port of ", inet_ntoa($ip);
32
33 $name = gethostbyaddr($ip, AF_INET);
34 if (defined $name) {
35     print " ($name)";
36 }
37
38 print "\n$buffer\n";
39 exit 0;
40
41 sub handle_sigalrm {
42     die "$0: The datagram I was waiting for didn't arrive in 15 seconds.\n";
43 }
```

I ran **snoop** and my **udp.pl** on the host **phoenix.wolfnet.com** (**192.168.0.5**). **aixmita1** has the digit one; **heidelberg** has a lowercase L.

```
8$ snoop -o udp.snoop host aixmita1.urz.uni-heidelberg.de &
[1] 12345
```
The shell displays the PID number of a background process.

The count of 26 bytes includes a terminating carriage return and line feed.

 All rights reserved ©2005 Mark Meretzky

```
9$ udp.pl
I sent an empty datagram from my port 35583.
I then received a datagram containing the following 26 bytes
from port 13 of 129.206.218.160 (aixmita1.urz.uni-heidelberg.de)
Thu May 29 19:25:47 2003

10$ date
Thu May 29 13:25:47 EDT 2003          Germany is 6 hours ahead of us.

11$ kill 12345
```

Packet 1

The host that speaks first is the *client* (**phoenix.wolfnet.com**; **192.168.0.1**); the other host is the *server* (**aixmita1.urz.uni-heidelberg.de** (**129.206.218.160**).

The Ethernet destination address in line 5 is not the Ethernet address of **aixmita1**. **aixmita1** may not even have an Ethernet address—its LAN might be a Token Ring or FDDI. The address in line 5 is merely the address of the first gateway along the route to Germany.

The code number in line 7 is the macro **ETHERTYPE_IP** in lines 35 and 39 of the header file **ethernet.h** in Handout 2, pp. 3–4. The number in line 27 is the code number for UDP in line 59 of the header file **ip.h** and in the **/etc/protocols** file in Handout 2, p. 8.

The outgoing Ethernet frame is 42 bytes (line 4). Apparently, we're seeing it before it is padded to the minimum Ethernet frame size. The first 14 bytes are the Ethernet header, leaving 42 – 14 = 28 bytes for the cargo (line 20).

The cargo of the Ethernet frame is a 28-byte IP datagram. The first 20 bytes are the IP header (line 12), leaving 28 – 20 = 8 bytes for the cargo.

The cargo of the IP datagram is an 8-byte UDP message. The first 8 bytes are the UDP header, leaving room for no cargo. The file **udp.h** in Handout 2, p. 16 lists the four two-byte fields of the UDP header.

The source port number in line 35 was chosen arbitrarily by UDP. It will be used only once, as the destination of packet 2, and is called an ‘‘ephemeral port number’’.

 All rights reserved ©2005 Mark Meretzky

```
1$ cd ~mm64/public_html/x52.9547/src/snoop
2$ snoop -i udp.snoop -p 1,1 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 1 arrived at 13:25:49.12
     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 = 63333
    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 = 674e
    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 = 35583
    36  UDP:  Destination port = 13 (DAYTIME)
    37  UDP:  Length = 8
    38  UDP:  Checksum = 57B5
    39  UDP:
    40  DAYTIME:  ----- DAYTIME:   -----
    41  DAYTIME:
    42  DAYTIME:  ""
    43  DAYTIME:
```

To compute the UDP checksum **0x57B5** in the above line 38, we first display the entire UDP packet in hex by chopping off the 14-byte Ethernet header and 20-byte IP header:

```
3$ snoop -i udp.snoop -p 1,1 -x 34 | tail +3
       0: 8aff 000d 0008 57b5                            ......W.
```

The UDP packet consists of an eight-byte header with no cargo. Add the four two-byte words (except the fourth, **57b8**, which is the checksum itself), plus a “pseudoheader” consisting of the source

 All rights reserved ©2005 Mark Meretzky

and destination IP addresses, the protocol number, and the UDP packet length:

```
4$ bc
obase = 16
ibase = 16
8AFF+000D+0008+0000+\
C0A8+0005+\          Source address: 192.168.0.5
81CE+DAA0+\          Destination address: 129.206.218.160
0011+\               UDP is protocol number 17 in /etc/protocols
0008                 length of UDP packet
2A848
control-d
```

The bottom 16 bits of the sum (**A848**) plus the top 16 bits of the sum (**0002**) is **A84A**. The "one's complement" (i.e., bitwise not, photographic negative) of **A84A** is the IP header checksum **57B5** in the above line 38.

```
  A848
+ 0002
------
  A84A == 1010 1000 0100 1010
  57B5 == 0101 0111 1011 0101
```

Packet 2

The cargo of the IP datagram is an 34-byte UDP message (line 37). The first 8 bytes are the UDP header, leaving 26 bytes for cargo. These are the 26 characters in line 42, including a carriage return (**'\r'**) and newline (**'\n'**) but no terminating **'\0'** character.

 All rights reserved ©2005 Mark Meretzky

```
1$ snoop -i udp.snoop -p 2,2 -ta -v | cat -n | more
     1  ETHER:  ----- Ether Header -----
     2  ETHER:
     3  ETHER:  Packet 2 arrived at 13:25:49.23
     4  ETHER:  Packet size = 72 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 = 54 bytes
    21  IP:   Identification = 50659
    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 = 16 seconds/hops
    27  IP:   Protocol = 17 (UDP)
    28  IP:   Header checksum = c7b7
    29  IP:   Source address = 129.206.218.160, aixmita1.urz.uni-heidelberg.de
    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 = 13
    36  UDP:  Destination port = 35583
    37  UDP:  Length = 34
    38  UDP:  Checksum = 28DB
    39  UDP:
    40  DAYTIME:  ----- DAYTIME:   -----
    41  DAYTIME:
    42  DAYTIME:  "Thu May 29 19:25:47 2003\r\n"
    43  DAYTIME:
```

The “time to live” in line 26 of packet 2 from Germany has counted down to 16 by the time the packet reaches us.

 All rights reserved ©2005 Mark Meretzky

```
2$ traceroute aixmita1.urz.uni-heidelberg.de
traceroute to aixmita1.urz.uni-heidelberg.de (129.206.218.160),
30 hops max, 40 byte packets
 1  WWHGWA-FDDI-10-0.NYU.NET (128.122.253.129)  0.706 ms  0.459 ms  0.444 ms
 2  WPGWA-FDDI-9-0.NYU.NET (128.122.253.89)  0.726 ms  1.127 ms  0.780 ms
 3  NYUGWA-GE-4-2.NYU.NET (128.122.253.9)  0.585 ms  0.598 ms  0.782 ms
 4  EXTGWB-GE-3-0-0.NYU.NET (192.76.177.68)  0.797 ms  0.751 ms  0.653 ms
 5  nyc-m20-nyu.nysernet.net (199.109.5.46)  1.407 ms  1.613 ms  2.618 ms
 6  abilene-nycm-nyc-m20.nysernet.net (199.109.5.2)  1.568 ms  1.606 ms  2.046 ms
 7  198.32.11.62 (198.32.11.62)  1.944 ms  1.762 ms  1.972 ms
 8  ny.uk1.uk.geant.net (62.40.96.170)  72.802 ms  70.540 ms  69.786 ms
 9  uk.fr1.fr.geant.net (62.40.96.89)  78.042 ms  86.584 ms  76.796 ms
10  fr.de1.de.geant.net (62.40.96.49)  92.992 ms  92.552 ms  92.485 ms
11  62.40.103.34 (62.40.103.34)  94.138 ms  93.763 ms  93.695 ms
12  cr-frankfurt1-po8-2.g-win.dfn.de (188.1.80.37)  94.585 ms  93.182 ms  93.839 ms
13  cr-stuttgart1-po4-0.g-win.dfn.de (188.1.18.78)  96.392 ms  96.526 ms  97.416 ms
14  ar-stuttgart2-ge0-0-0.g-win.dfn.de (188.1.76.4)  97.914 ms  96.749 ms  96.130 ms
15  Stuttgart2.belwue.de (188.1.38.54)  97.158 ms  97.168 ms  96.421 ms
16  Heidelberg1.BelWue.DE (129.143.1.2)  98.893 ms  99.277 ms  98.748 ms
17  129.143.79.115 (129.143.79.115)  99.658 ms  99.435 ms  99.614 ms
18  aixmita1.urz.uni-heidelberg.de (129.206.218.160)  99.499 ms  99.882 ms  100.062 ms
```

The TCP header: Transmission Control Protocol

> “And you’ll write to me regularly. . . . And they must all be numbered, every one of them, so that I shall know at once if I’ve missed one; remember!”
>
> —*Edith Wharton, Old New York: False Dawn, Part 1, Chapter II*

The TCP header is in the diagrams on pp. 19 and 682:

 All rights reserved ©2005 Mark Meretzky

```
1$ cat -n /usr/include/netinet/tcp.h | sed -n 27,55p | more
    27  /*
    28   * TCP header.
    29   * Per RFC 793, September, 1981.
    30   */
    31  struct tcphdr {
    32      uint16_t th_sport;    /* source port */
    33      uint16_t th_dport;    /* destination port */
    34      tcp_seq     th_seq;      /* sequence number */
    35      tcp_seq     th_ack;      /* acknowledgement number */
    36  #ifdef _BIT_FIELDS_LTOH
    37      uint_t  th_x2:4,     /* (unused) */
    38          th_off:4;        /* data offset */
    39  #else
    40      uint_t  th_off:4,        /* data offset */
    41          th_x2:4;    /* (unused) */
    42  #endif
    43      uchar_t  th_flags;
    44  #define  TH_FIN   0x01
    45  #define  TH_SYN   0x02
    46  #define  TH_RST   0x04
    47  #define  TH_PUSH  0x08
    48  #define  TH_ACK   0x10
    49  #define  TH_URG   0x20
    50  #define  TH_ECE   0x40
    51  #define  TH_CWR   0x80
    52      uint16_t th_win;      /* window */
    53      uint16_t th_sum;      /* checksum */
    54      uint16_t th_urp;      /* urgent pointer */
    55  };
```

Dialog 3: A nine-packet dialog carried by TCP

The above conversation in UDP took only two packets; the same conversation in TCP will occupy nine. TCP requires more bandwidth, but is more reliable because it sends a confirmation for every packet.

Two programs communicating via TCP keep track of which packets they've sent and received by keeping track of what TCP "state" they're in. The **netstat** program will display the state of each program; the 13 possible states are listed in **netstat**(1M) pp. 2–3.

The arguments **-f inet** tell **netstat** to display only conversations that are carried by IP; **-P tcp** tell **netstat** to display only conversations that are carried by TCP. Most of these programs are in the **LISTEN** state. A program in this state is a server that is waiting to accept a client. If you don't want to see all these dormant servers, don't say **-a**.

The numeric columns are the sizes of the send window and send queue, and the receive window and receive queue. The window is the currently unused part of the buffer. The program **lsof -U** would show the PID number of each process, but only the superuser can do this.

awk will show us the first program in each TCP state. In most languages, an array subscript must be an integer, but in **awk** a subscript can be a string. The strings we will use will be **$NF**, the last word on each line of input. The value of each array element is initially zero, and will be incremented each time its subscript is seen at the end of an input line. We output the entire input line if its last word is making its first appearance:

 All rights reserved ©2005 Mark Meretzky

```
1$ /bin/netstat -a -f inet -P tcp | \
awk '2 <= NR && NR <= 4 || NR > 4 && ++count[$NF] == 1'
TCP: IPv4
   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
      *.*                  *.*                0      0 49152      0 IDLE
      *.sunrpc             *.*                0      0 49152      0 LISTEN
i5.ssh               h-67-101-157-125.nycmny83.dynamic.covad.net.4648 64307      0 4942
i5.smtp              RECON.ES.ITS.NYU.EDU.55033  6912      0 49232      0 TIME_WAIT
```

To send UDP packets, I had to write the Perl program in the second dialog. To send TCP packets, I merely ran the **telnet** client on the host **phoenix.wolfnet.com** (**192.168.0.5**). The second argument of **telnet** is the port number on the host that we want to talk to.

```
3$ snoop -o tcp.snoop host aixmita1.urz.uni-heidelberg.de &
[1] 12345
```
The shell displays the PID number of a background process.

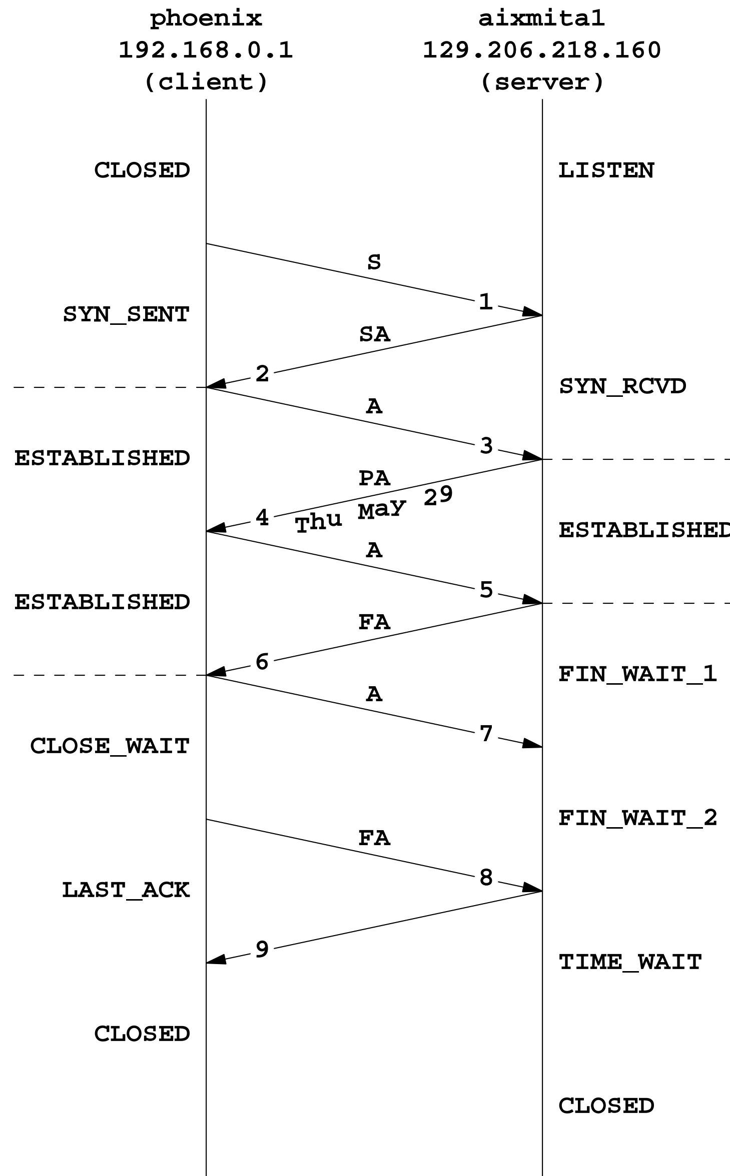
```
4$ telnet aixmita1.urz.uni-heidelberg.de 13
Trying 129.206.218.160...
Connected to aixmita1.urz.uni-heidelberg.de.
Escape character is '^]'.
Thu May 29 20:18:41 2003
Connection closed by foreign host.

5$ date
Thu May 29 14:18:43 2003
```
Germany is 6 hours ahead of us.

```
6$ kill 12345
```

Once again, the host that speaks first is the client (**phoenix.wolfnet.com**; **192.168.0.1**); the other host is the server (**aixmita1.urz.uni-heidelberg.de**; **129.206.218.160**). The first three TCP segments are called the "three-way handshake"; see pp 19–20. Either host can initiate the end of the conversation. In our dialog the server does, with the **FIN** in packet 6. The last four packets could be called the "four-way handshake". The users' data is transmitted while both parties are in the **ESTABLISHED** state.

 All rights reserved ©2005 Mark Meretzky



Segment 1

Initially, the client is in the **CLOSED** state and the server is in the **LISTEN** state. As it emits packet 1 (with the **SYN** in line 47), the client moves from the **CLOSED** state to the **SYN_SENT** state. Line 57 is a timestamp (RFC 1323). Line 49 gives a window size of 32850 = 32K + 82; see p. 21. Line 60 permits selective acknowledgement of bytes.

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

 All rights reserved ©2005 Mark Meretzky

```
 1    ETHER:  ----- Ether Header -----
 2    ETHER:
 3    ETHER:  Packet 1 arrived at 14:18:42.92
 4    ETHER:  Packet size = 78 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 = 64 bytes
21    IP:   Identification = 23525
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 = 64 seconds/hops
27    IP:   Protocol = 6 (TCP)
28    IP:   Header checksum = c1b6
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    TCP:  ----- TCP Header -----
34    TCP:
35    TCP:  Source port = 56727
36    TCP:  Destination port = 13 (DAYTIME)
37    TCP:  Sequence number = 2268097426
38    TCP:  Acknowledgement number = 0
39    TCP:  Data offset = 44 bytes
40    TCP:  Flags = 0x02
41    TCP:        0... .... = No ECN congestion window reduced
42    TCP:        .0.. .... = No ECN echo
43    TCP:        ..0. .... = No urgent pointer
44    TCP:        ...0 .... = No acknowledgement
45    TCP:        .... 0... = No push
46    TCP:        .... .0.. = No reset
47    TCP:        .... ..1. = Syn
48    TCP:        .... ...0 = No Fin
49    TCP:  Window = 32850
50    TCP:  Checksum = 0x9586
51    TCP:  Urgent pointer = 0
52    TCP:  Options: (24 bytes)
53    TCP:    - No operation
54    TCP:    - Window scale = 1
55    TCP:    - No operation
56    TCP:    - No operation
57    TCP:    - TS Val = 265888458, TS Echo = 0
58    TCP:    - No operation
59    TCP:    - No operation
60    TCP:    - SACK permitted option
61    TCP:    - Maximum segment size = 1460 bytes
62    TCP:
63    DAYTIME:  ----- DAYTIME:   -----
64    DAYTIME:
65    DAYTIME:  ""
66    DAYTIME:
```

To compute the TCP checksum `0x9586` in the above line 50, we first display the entire TCP segment in hex by chopping off the 14-byte Ethernet header and 20-byte IP header:

 All rights reserved ©2005 Mark Meretzky

```
3$ snoop -i udp.snoop -p 1,1 -x 34 | tail +3
       0: dd97 000d 8730 6b92 0000 0000 b002 8052    .....0k........R
      16: 9586 0000 0103 0301 0101 080a 0fd9 22ca    ..............".
      32: 0000 0000 0101 0402 0204 05b4              ............
```

The TCP segment consists of a 44-byte header (line 39) with no cargo. The header is the usual 20 bytes, plus 24 bytes of options (line 52). Add the 22 two-byte words (except the ninth, **9586**, which is the checksum itself), plus a "pseudoheader" consisting of the source and destination IP addresses, the protocol number, and the TCP segment length:

```
4$ bc
obase = 16
ibase = 16
DD97+000D+8730+6B92+0000+0000+B002+8052+\
0000+0000+0103+0301+0101+080A+0FD9+22CA+\
0000+0000+0101+0402+0204+05B4+\
C0A8+0005+\           Source address: 192.168.0.5
81CE+DAA0+\           Destination address: 129.206.218.160
0006+\                TCP is protocol number 6 in the above line 27; see /etc/protocols
002C                  length of TCP segment: 44 bytes in the above line 39
56A74
control-d
```

The bottom 16 bits of the sum (**6A74**) plus the top 16 bits of the sum (**0005**) is **6A79**. The "one's complement" (i.e., bitwise not, photographic negative) of **6A79** is the IP header checksum **9586** in the above line 50.

```
  6A74
+ 0005
------
  6A79 == 0110 1010 0111 1001
  9586 == 1001 0101 1000 0110
```

Segment 2

As it receives segment 1 and emits segment 2 (with the **SYN** in line 47) the server moves from the **LISTEN** state to the **SYN_RCVD** state. This and all subsequent segments have the **ACK** in line 44.

Every TCP segment coming back to us has a window size of 64620 = 64K – 916; see line 49.

We sent out packet 1 with a time-to-live of 64; packet 2 arrives with its time-to-live reduced to 47 (line 26).

```
1$ snoop -i tcp.snoop -p 2,2 -ta -v | cat -n | more
```

 All rights reserved ©2005 Mark Meretzky

```
 1     ETHER:  ----- Ether Header -----
 2     ETHER:
 3     ETHER:  Packet 2 arrived at 14:18:43.04
 4     ETHER:  Packet size = 78 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 = 60 bytes
21     IP:   Identification = 21567
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 = 47 seconds/hops
27     IP:   Protocol = 6 (TCP)
28     IP:   Header checksum = 1a61
29     IP:   Source address = 129.206.218.160, aixmita1.urz.uni-heidelberg.de
30     IP:   Destination address = 192.168.0.5, 192.168.0.5
31     IP:   No options
32     IP:
33     TCP:  ----- TCP Header -----
34     TCP:
35     TCP:  Source port = 13
36     TCP:  Destination port = 56727
37     TCP:  Sequence number = 4290149869
38     TCP:  Acknowledgement number = 2268097427
39     TCP:  Data offset = 40 bytes
40     TCP:  Flags = 0x12
41     TCP:        0... .... = No ECN congestion window reduced
42     TCP:        .0.. .... = No ECN echo
43     TCP:        ..0. .... = No urgent pointer
44     TCP:        ...1 .... = Acknowledgement
45     TCP:        .... 0... = No push
46     TCP:        .... .0.. = No reset
47     TCP:        .... ..1. = Syn
48     TCP:        .... ...0 = No Fin
49     TCP:  Window = 64620
50     TCP:  Checksum = 0x2e53
51     TCP:  Urgent pointer = 0
52     TCP:  Options: (20 bytes)
53     TCP:    - Maximum segment size = 1448 bytes
54     TCP:    - No operation
55     TCP:    - Window scale = 0
56     TCP:    - No operation
57     TCP:    - No operation
58     TCP:    - TS Val = 1054360479, TS Echo = 265888458
59     TCP:
60     DAYTIME:  ----- DAYTIME:   -----
61     DAYTIME:
62     DAYTIME:  ""
63     DAYTIME:
```

Segment 3

As it receives segment 2 and emits segment 3, the client moves from the **`SYN_SENT`** state to the **`ESTABLISHED`** state. As it receives segment 3, the server moves from the **`SYN_RCVD`** state to the **`ESTABLISHED`** state.

 All rights reserved ©2005 Mark Meretzky

```
1$ snoop -i tcp.snoop -p 3,3 -ta -v | cat -n | more

     1	ETHER:  ----- Ether Header -----
     2	ETHER:
     3	ETHER:  Packet 3 arrived at 14:18:43.04
     4	ETHER:  Packet size = 66 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 = 52 bytes
    21	IP:   Identification = 23526
    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 = 64 seconds/hops
    27	IP:   Protocol = 6 (TCP)
    28	IP:   Header checksum = c1c1
    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	TCP:  ----- TCP Header -----
    34	TCP:
    35	TCP:  Source port = 56727
    36	TCP:  Destination port = 13 (DAYTIME)
    37	TCP:  Sequence number = 2268097427
    38	TCP:  Acknowledgement number = 4290149870
    39	TCP:  Data offset = 32 bytes
    40	TCP:  Flags = 0x10
    41	TCP:        0... .... = No ECN congestion window reduced
    42	TCP:        .0.. .... = No ECN echo
    43	TCP:        ..0. .... = No urgent pointer
    44	TCP:        ...1 .... = Acknowledgement
    45	TCP:        .... 0... = No push
    46	TCP:        .... .0.. = No reset
    47	TCP:        .... ..0. = No Syn
    48	TCP:        .... ...0 = No Fin
    49	TCP:  Window = 33028
    50	TCP:  Checksum = 0xd567
    51	TCP:  Urgent pointer = 0
    52	TCP:  Options: (12 bytes)
    53	TCP:    - No operation
    54	TCP:    - No operation
    55	TCP:    - TS Val = 265888470, TS Echo = 1054360479
    56	TCP:
    57	DAYTIME:  ----- DAYTIME:   -----
    58	DAYTIME:
    59	DAYTIME:  ""
    60	DAYTIME:
```

Segment 4

Segment 4 is the only one that carries data that the user sees (line 59). The 26 bytes of data end with a carriage return (**'\r'**) and newline (**'\n'**), but no terminating **'\0'** character. Together with the 32-byte TCP header (line 39), this accounts for the size of 58 bytes for the TCP segment. Together with the 20-byte IP header (line 12), this accounts for the size of 78 bytes for the IP datagram (line 20). Together

 All rights reserved ©2005 Mark Meretzky

with the 14-byte Ethernet header, this should give us a size of 92 bytes for the Ethernet frame (line 4).

This is the first time we've seen the ''push'' flag in line 45 turned on. It described in section 2.8 of RFC 793: ''When a receiving TCP sees the PUSH flag, it must not wait for more data from the sending TCP before passing the data to the receiving process.'' If, for example, the receiving process is a Perl program reading from a socket,

```
1 $x = <S>;
```

the Perl program will receive all the data in this segment.

```
1$ snoop -i tcp.snoop -p 4,4 -ta -v | cat -n | more
```

 All rights reserved ©2005 Mark Meretzky

```
 1     ETHER:  ----- Ether Header -----
 2     ETHER:
 3     ETHER:  Packet 4 arrived at 14:18:43.16
 4     ETHER:  Packet size = 96 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 = 78 bytes
21     IP:   Identification = 21571
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 = 47 seconds/hops
27     IP:   Protocol = 6 (TCP)
28     IP:   Header checksum = 1a4b
29     IP:   Source address = 129.206.218.160, aixmita1.urz.uni-heidelberg.de
30     IP:   Destination address = 192.168.0.5, 192.168.0.5
31     IP:   No options
32     IP:
33     TCP:  ----- TCP Header -----
34     TCP:
35     TCP:  Source port = 13
36     TCP:  Destination port = 56727
37     TCP:  Sequence number = 4290149870
38     TCP:  Acknowledgement number = 2268097427
39     TCP:  Data offset = 32 bytes
40     TCP:  Flags = 0x18
41     TCP:        0... .... = No ECN congestion window reduced
42     TCP:        .0.. .... = No ECN echo
43     TCP:        ..0. .... = No urgent pointer
44     TCP:        ...1 .... = Acknowledgement
45     TCP:        .... 1... = Push
46     TCP:        .... .0.. = No reset
47     TCP:        .... ..0. = No Syn
48     TCP:        .... ...0 = No Fin
49     TCP:  Window = 64620
50     TCP:  Checksum = 0x3b33
51     TCP:  Urgent pointer = 0
52     TCP:  Options: (12 bytes)
53     TCP:    - No operation
54     TCP:    - No operation
55     TCP:    - TS Val = 1054360479, TS Echo = 265888470
56     TCP:
57     DAYTIME:  ----- DAYTIME:   -----
58     DAYTIME:
59     DAYTIME:  "Thu May 29 20:18:41 2003\r\n"
60     DAYTIME:
```

Segment 5

The client remains in the **ESTABLISHED** state when it emits segment 5, and the server remains in the **ESTABLISHED** state when it receives segment 5. The only purpose of segment 5 is to acknowledge segment 4 (line 44). Note that the acknowledgement number in line 38 is 26 greater than the acknowledgement number in line 38 of segment 3.

```
1$ snoop -i tcp.snoop -p 5,5 -ta -v | cat -n | more
```

 All rights reserved ©2005 Mark Meretzky

```
 1    ETHER:  ----- Ether Header -----
 2    ETHER:
 3    ETHER:  Packet 5 arrived at 14:18:43.16
 4    ETHER:  Packet size = 66 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 = 52 bytes
21    IP:   Identification = 23527
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 = 64 seconds/hops
27    IP:   Protocol = 6 (TCP)
28    IP:   Header checksum = c1c0
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    TCP:  ----- TCP Header -----
34    TCP:
35    TCP:  Source port = 56727
36    TCP:  Destination port = 13 (DAYTIME)
37    TCP:  Sequence number = 2268097427
38    TCP:  Acknowledgement number = 4290149896
39    TCP:  Data offset = 32 bytes
40    TCP:  Flags = 0x10
41    TCP:        0... .... = No ECN congestion window reduced
42    TCP:        .0.. .... = No ECN echo
43    TCP:        ..0. .... = No urgent pointer
44    TCP:        ...1 .... = Acknowledgement
45    TCP:        .... 0... = No push
46    TCP:        .... .0.. = No reset
47    TCP:        .... ..0. = No Syn
48    TCP:        .... ...0 = No Fin
49    TCP:  Window = 33028
50    TCP:  Checksum = 0xd540
51    TCP:  Urgent pointer = 0
52    TCP:  Options: (12 bytes)
53    TCP:    - No operation
54    TCP:    - No operation
55    TCP:    - TS Val = 265888483, TS Echo = 1054360479
56    TCP:
57    DAYTIME:  ----- DAYTIME:   -----
58    DAYTIME:
59    DAYTIME:  ""
60    DAYTIME:
```

Segment 6

As it emits segment 6 (with the **FIN** in line 48) the server moves from the **ESTABLISHED** state to the **FIN_WAIT_1** state.

```
1$ snoop -i tcp.snoop -p 6,6 -ta -v | cat -n | more
```

 All rights reserved ©2005 Mark Meretzky

```
 1     ETHER:  ----- Ether Header -----
 2     ETHER:
 3     ETHER:  Packet 6 arrived at 14:18:43.16
 4     ETHER:  Packet size = 70 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 = 52 bytes
21     IP:   Identification = 21572
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 = 47 seconds/hops
27     IP:   Protocol = 6 (TCP)
28     IP:   Header checksum = 1a64
29     IP:   Source address = 129.206.218.160, aixmita1.urz.uni-heidelberg.de
30     IP:   Destination address = 192.168.0.5, 192.168.0.5
31     IP:   No options
32     IP:
33     TCP:  ----- TCP Header -----
34     TCP:
35     TCP:  Source port = 13
36     TCP:  Destination port = 56727
37     TCP:  Sequence number = 4290149896
38     TCP:  Acknowledgement number = 2268097427
39     TCP:  Data offset = 32 bytes
40     TCP:  Flags = 0x11
41     TCP:        0... .... = No ECN congestion window reduced
42     TCP:        .0.. .... = No ECN echo
43     TCP:        ..0. .... = No urgent pointer
44     TCP:        ...1 .... = Acknowledgement
45     TCP:        .... 0... = No push
46     TCP:        .... .0.. = No reset
47     TCP:        .... ..0. = No Syn
48     TCP:        .... ...1 = Fin
49     TCP:  Window = 64620
50     TCP:  Checksum = 0x59e4
51     TCP:  Urgent pointer = 0
52     TCP:  Options: (12 bytes)
53     TCP:    - No operation
54     TCP:    - No operation
55     TCP:    - TS Val = 1054360479, TS Echo = 265888470
56     TCP:
57     DAYTIME:  ----- DAYTIME:   -----
58     DAYTIME:
59     DAYTIME:  ""
60     DAYTIME:
```

Segment 7

As it receives segment 6 (**`FIN`**) and emits segment 7, the client moves from the **`ESTABLISHED`** state to the **`CLOSE_WAIT`** state. As it receives segment 7, the server moves from the **`FIN_WAIT_1`** state to the **`FIN_WAIT_2`** state.

```
1$ snoop -i tcp.snoop -p 7,7 -ta -v | cat -n | more
```

 All rights reserved ©2005 Mark Meretzky

```
 1     ETHER:  ----- Ether Header -----
 2     ETHER:
 3     ETHER:  Packet 7 arrived at 14:18:43.16
 4     ETHER:  Packet size = 66 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 = 52 bytes
21     IP:   Identification = 23528
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 = 64 seconds/hops
27     IP:   Protocol = 6 (TCP)
28     IP:   Header checksum = c1bf
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     TCP:  ----- TCP Header -----
34     TCP:
35     TCP:  Source port = 56727
36     TCP:  Destination port = 13 (DAYTIME)
37     TCP:  Sequence number = 2268097427
38     TCP:  Acknowledgement number = 4290149897
39     TCP:  Data offset = 32 bytes
40     TCP:  Flags = 0x10
41     TCP:        0... .... = No ECN congestion window reduced
42     TCP:        .0.. .... = No ECN echo
43     TCP:        ..0. .... = No urgent pointer
44     TCP:        ...1 .... = Acknowledgement
45     TCP:        .... 0... = No push
46     TCP:        .... .0.. = No reset
47     TCP:        .... ..0. = No Syn
48     TCP:        .... ...0 = No Fin
49     TCP:  Window = 33028
50     TCP:  Checksum = 0xd53f
51     TCP:  Urgent pointer = 0
52     TCP:  Options: (12 bytes)
53     TCP:    - No operation
54     TCP:    - No operation
55     TCP:    - TS Val = 265888483, TS Echo = 1054360479
56     TCP:
57     DAYTIME:  ----- DAYTIME:   -----
58     DAYTIME:
59     DAYTIME:  ""
60     DAYTIME:
```

Segment 8

As it emits segment 8 (with the **FIN** in line 48) the client moves from the **CLOSE_WAIT** state to the **LAST_ACK** state.

```
1$ snoop -i tcp.snoop -p 8,8 -ta -v | cat -n | more
```

 All rights reserved ©2005 Mark Meretzky

```
 1    ETHER:  ----- Ether Header -----
 2    ETHER:
 3    ETHER:  Packet 8 arrived at 14:18:43.16
 4    ETHER:  Packet size = 66 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 = 52 bytes
21    IP:   Identification = 23529
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 = 64 seconds/hops
27    IP:   Protocol = 6 (TCP)
28    IP:   Header checksum = c1be
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    TCP:  ----- TCP Header -----
34    TCP:
35    TCP:  Source port = 56727
36    TCP:  Destination port = 13 (DAYTIME)
37    TCP:  Sequence number = 2268097427
38    TCP:  Acknowledgement number = 4290149897
39    TCP:  Data offset = 32 bytes
40    TCP:  Flags = 0x11
41    TCP:        0... .... = No ECN congestion window reduced
42    TCP:        .0.. .... = No ECN echo
43    TCP:        ..0. .... = No urgent pointer
44    TCP:        ...1 .... = Acknowledgement
45    TCP:        .... 0... = No push
46    TCP:        .... .0.. = No reset
47    TCP:        .... ..0. = No Syn
48    TCP:        .... ...1 = Fin
49    TCP:  Window = 33028
50    TCP:  Checksum = 0xd53e
51    TCP:  Urgent pointer = 0
52    TCP:  Options: (12 bytes)
53    TCP:    - No operation
54    TCP:    - No operation
55    TCP:    - TS Val = 265888483, TS Echo = 1054360479
56    TCP:
57    DAYTIME:  ----- DAYTIME:   -----
58    DAYTIME:
59    DAYTIME:  ""
60    DAYTIME:
```

Segment 9

As it receives segment 8 and emits segment 9, the server moves from the **`FIN_WAIT_2`** state to the **`TIME_WAIT`** state. As it receives segment 9 (**`ACK`**), the client moves from the **`LAST_ACK`** state to the **`CLOSED`** state. After a minute or two, the server then changes to the **`CLOSED`** state.

```
1$ snoop -i tcp.snoop -p 9,9 -ta -v | cat -n | more
```

 All rights reserved ©2005 Mark Meretzky

```
 1    ETHER:  ----- Ether Header -----
 2    ETHER:
 3    ETHER:  Packet 9 arrived at 14:18:43.29
 4    ETHER:  Packet size = 70 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 = 52 bytes
21    IP:   Identification = 21573
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 = 47 seconds/hops
27    IP:   Protocol = 6 (TCP)
28    IP:   Header checksum = 1a63
29    IP:   Source address = 129.206.218.160, aixmita1.urz.uni-heidelberg.de
30    IP:   Destination address = 192.168.0.5, 192.168.0.5
31    IP:   No options
32    IP:
33    TCP:  ----- TCP Header -----
34    TCP:
35    TCP:  Source port = 13
36    TCP:  Destination port = 56727
37    TCP:  Sequence number = 4290149897
38    TCP:  Acknowledgement number = 2268097428
39    TCP:  Data offset = 32 bytes
40    TCP:  Flags = 0x10
41    TCP:        0... .... = No ECN congestion window reduced
42    TCP:        .0.. .... = No ECN echo
43    TCP:        ..0. .... = No urgent pointer
44    TCP:        ...1 .... = Acknowledgement
45    TCP:        .... 0... = No push
46    TCP:        .... .0.. = No reset
47    TCP:        .... ..0. = No Syn
48    TCP:        .... ...0 = No Fin
49    TCP:  Window = 64620
50    TCP:  Checksum = 0x59d6
51    TCP:  Urgent pointer = 0
52    TCP:  Options: (12 bytes)
53    TCP:    - No operation
54    TCP:    - No operation
55    TCP:    - TS Val = 1054360479, TS Echo = 265888483
56    TCP:
57    DAYTIME:  ----- DAYTIME:   -----
58    DAYTIME:
59    DAYTIME:  ""
60    DAYTIME:
```

A summary of the above dialog

Here is the least verbose rendering of the above dialog. Without the **-p** option, **snoop** shows us all nine packets. **C** is “call” (outgoing); **R** is “reply” (incoming).

 All rights reserved ©2005 Mark Meretzky

```
1$ cd ~mm64/public_html/x52.9547/src/snoop
2$ snoop -i tcp.snoop -ta | more
1 14:18:42.92447  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=56727
2 14:18:43.04544 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=56727
3 14:18:43.04570  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=56727
4 14:18:43.16891 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=56727 Th
5 14:18:43.16913  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=56727
6 14:18:43.16945 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=56727
7 14:18:43.16953  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=56727
8 14:18:43.16987  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=56727
9 14:18:43.29860 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=56727
```

With minus uppercase **V**, we get four long lines of output for each segment, folded every 80 bytes with the **-b -w 80**. They summarize the Ethernet frame, the IP datagram, the TCP segment, and the **daytime** cargo of the TCP segment. Before each group of four lines is a horizontal line of underscores.

```
3$ snoop -i tcp.snoop -ta -V | fold -b -w 80 | more

________________________________
1 14:18:42.92447  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de ETHER Type=0800
(IP), size = 78 bytes
1 14:18:42.92447  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de IP  D=129.206.21
8.160 S=192.168.0.5 LEN=64, ID=23525, TOS=0x0, TTL=64
1 14:18:42.92447  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de TCP D=13 S=56727
 Syn Seq=2268097426 Len=0 Win=32850 Options=<nop,wscale 1,nop,nop,tstamp 2658884
58 0,nop,nop,sackOK,mss 1460>
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  ETHER Type=0800
(IP), size = 78 bytes
2 14:18:43.04544 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  IP  D=192.168.0.
5 S=129.206.218.160 LEN=60, ID=21567, TOS=0x0, TTL=47
2 14:18:43.04544 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  TCP D=56727 S=13
 Syn Ack=2268097427 Seq=4290149869 Len=0 Win=64620 Options=<mss 1448,nop,wscale
0,nop,nop,tstamp 1054360479 265888458>
2 14:18:43.04544 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727

________________________________
3 14:18:43.04570  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de ETHER Type=0800
(IP), size = 66 bytes
3 14:18:43.04570  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de IP  D=129.206.21
8.160 S=192.168.0.5 LEN=52, ID=23526, TOS=0x0, TTL=64
3 14:18:43.04570  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de TCP D=13 S=56727
 Ack=4290149870 Seq=2268097427 Len=0 Win=33028 Options=<nop,nop,tstamp 265888470
 1054360479>
3 14:18:43.04570  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
```

 All rights reserved ©2005 Mark Meretzky

```
4 14:18:43.16891 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  ETHER Type=0800
(IP), size = 96 bytes
4 14:18:43.16891 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  IP  D=192.168.0.
5 S=129.206.218.160 LEN=78, ID=21571, TOS=0x0, TTL=47
4 14:18:43.16891 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  TCP D=56727 S=13
 Push Ack=2268097427 Seq=4290149870 Len=26 Win=64620 Options=<nop,nop,tstamp 105
4360479 265888470>
4 14:18:43.16891 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727 Thu May 29 20:18:41
```

```
5 14:18:43.16913  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de ETHER Type=0800
(IP), size = 66 bytes
5 14:18:43.16913  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de IP  D=129.206.21
8.160 S=192.168.0.5 LEN=52, ID=23527, TOS=0x0, TTL=64
5 14:18:43.16913  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de TCP D=13 S=56727
 Ack=4290149896 Seq=2268097427 Len=0 Win=33028 Options=<nop,nop,tstamp 265888483
 1054360479>
5 14:18:43.16913  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
```

```
6 14:18:43.16945 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  ETHER Type=0800
(IP), size = 70 bytes
6 14:18:43.16945 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  IP  D=192.168.0.
5 S=129.206.218.160 LEN=52, ID=21572, TOS=0x0, TTL=47
6 14:18:43.16945 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  TCP D=56727 S=13
 Fin Ack=2268097427 Seq=4290149896 Len=0 Win=64620 Options=<nop,nop,tstamp 10543
60479 265888470>
6 14:18:43.16945 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727
```

```
7 14:18:43.16953  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de ETHER Type=0800
(IP), size = 66 bytes
7 14:18:43.16953  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de IP  D=129.206.21
8.160 S=192.168.0.5 LEN=52, ID=23528, TOS=0x0, TTL=64
7 14:18:43.16953  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de TCP D=13 S=56727
 Ack=4290149897 Seq=2268097427 Len=0 Win=33028 Options=<nop,nop,tstamp 265888483
 1054360479>
7 14:18:43.16953  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
```


All rights reserved ©2005 Mark Meretzky

```
8 14:18:43.16987  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de ETHER Type=0800
(IP), size = 66 bytes
8 14:18:43.16987  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de IP  D=129.206.21
8.160 S=192.168.0.5 LEN=52, ID=23529, TOS=0x0, TTL=64
8 14:18:43.16987  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de TCP D=13 S=56727
 Fin Ack=4290149897 Seq=2268097427 Len=0 Win=33028 Options=<nop,nop,tstamp 26588
8483 1054360479>
8 14:18:43.16987  192.168.0.5 -> aixmita1.urz.uni-heidelberg.de DAYTIME C port=5
6727
```

```
9 14:18:43.29860 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  ETHER Type=0800
(IP), size = 70 bytes
9 14:18:43.29860 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  IP  D=192.168.0.
5 S=129.206.218.160 LEN=52, ID=21573, TOS=0x0, TTL=47
9 14:18:43.29860 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  TCP D=56727 S=13
 Ack=2268097428 Seq=4290149897 Len=0 Win=64620 Options=<nop,nop,tstamp 105436047
9 265888483>
9 14:18:43.29860 aixmita1.urz.uni-heidelberg.de -> 192.168.0.5  DAYTIME R port=5
6727
```

grep the TCP sequence numbers

A segment carrying the **SYN** or **FIN** bit counts as one byte, even though it contains no data.

```
1$ cd ~mm64/public_html/x52.9547/src/snoop/
2$ snoop -i tcp.snoop -v | grep 22680 | cat -n | more
     1  TCP:  Sequence number = 2268097426
     2  TCP:  Acknowledgement number = 2268097427
     3  TCP:  Sequence number = 2268097427
     4  TCP:  Acknowledgement number = 2268097427
     5  TCP:  Sequence number = 2268097427
     6  TCP:  Acknowledgement number = 2268097427
     7  TCP:  Sequence number = 2268097427
     8  TCP:  Sequence number = 2268097427
     9  TCP:  Acknowledgement number = 2268097428
```

The numbers jump by 26 between segments 4 and 5 because segment 4 contains 26 bytes of data. (We have to add the segment numbers with **awk**, not with **cat -n**, because segment 1 contained no acknowledgement number in its line 38.)

```
3$ cd ~mm64/public_html/x52.9547/src/snoop/
4$ snoop -i tcp.snoop -v | grep 42901 | awk '{print NR + 1, $0}' | more
2 TCP:  Sequence number = 4290149869
3 TCP:  Acknowledgement number = 4290149870
4 TCP:  Sequence number = 4290149870
5 TCP:  Acknowledgement number = 4290149896
6 TCP:  Sequence number = 4290149896
7 TCP:  Acknowledgement number = 4290149897
8 TCP:  Acknowledgement number = 4290149897
9 TCP:  Sequence number = 4290149897
```

 All rights reserved ©2005 Mark Meretzky

grep the TCP timestamp options

The TCP timestamp option is in RFC 1323. It gives the current time from some arbitrary starting point in hundredths of a second. For example, our packet 3 was sent out .12 seconds after packet 1, and the timestamp of packet 3 is 12 greater than the timestamp of packet 1.

```
1$ cd ~mm64/public_html/x52.9547/src/snoop/
2$ snoop -i tcp.snoop -v | egrep 'arrived at|TS Val' | more
ETHER:  Packet 1 arrived at 14:18:42.92
TCP:    - TS Val = 265888458, TS Echo = 0
ETHER:  Packet 2 arrived at 14:18:43.04
TCP:    - TS Val = 1054360479, TS Echo = 265888458
ETHER:  Packet 3 arrived at 14:18:43.04
TCP:    - TS Val = 265888470, TS Echo = 1054360479
ETHER:  Packet 4 arrived at 14:18:43.16
TCP:    - TS Val = 1054360479, TS Echo = 265888470
ETHER:  Packet 5 arrived at 14:18:43.16
TCP:    - TS Val = 265888483, TS Echo = 1054360479
ETHER:  Packet 6 arrived at 14:18:43.16
TCP:    - TS Val = 1054360479, TS Echo = 265888470
ETHER:  Packet 7 arrived at 14:18:43.16
TCP:    - TS Val = 265888483, TS Echo = 1054360479
ETHER:  Packet 8 arrived at 14:18:43.16
TCP:    - TS Val = 265888483, TS Echo = 1054360479
ETHER:  Packet 9 arrived at 14:18:43.29
TCP:    - TS Val = 1054360479, TS Echo = 265888483
```

▼ Homework 2.5: Dialog 4: the TCP reset flag

The header of a TCP segment has six flags (eight, nowadays). See Handout 2, p. 26, lines 43–51 of **tcp.h**; textbook, pp. 682–683. A TCP segment with all six flags turned on is called a *Christmas tree.*

On the host **phoenix.wolfnet.com**, I said

```
1$ snoop -o rst.snoop host smoke.wolfnet.com tcp port 13 &
[12345]

2$ telnet smoke.wolfnet.com 13
Trying 192.168.0.4...
telnet: Unable to connect to remote host: Connection refused

3$ kill 12345
```

Verify that I sent out one TCP segment carrying the **SYNC** bit, that I received one TCP segment carrying the **RST** (reset) bit, and that nothing else happened.

```
4$ cd ~mm64/public_html/x52.9547/src/snoop
5$ snoop -i rst.snoop -v | grep -i reset
TCP:        .... .0.. = No reset
TCP:        .... .1.. = Reset
```