“Standard” Unix Processes/IPC

- Through the filesystem: file descriptors, read()/write(), pipes

“Standard” Unix Processes/IPC

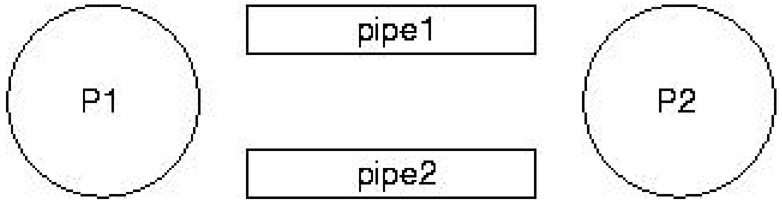
- Through the filesystem: file descriptors, read()/write(), pipes

- Implies: ipc construct shared thru normal process hierarchy inheritance rules; pipes are nameless (in most Unix dialects)

“Standard” Unix Processes/IPC

- Through the filesystem: file descriptors, read()/write(), pipes

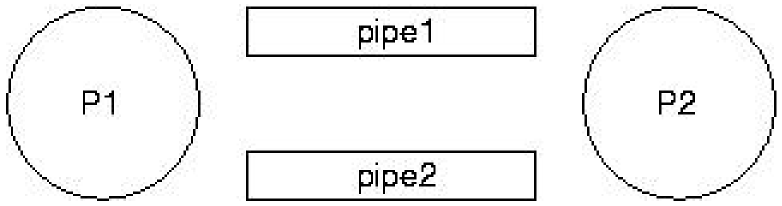
- Implies: ipc construct shared thru normal process hierarchy inheritance rules; pipes are nameless (in most Unix dialects)

- 

“Standard” Unix Processes/IPC

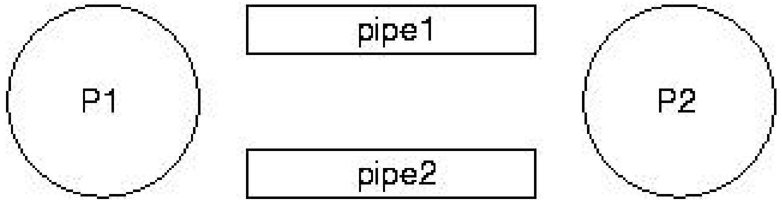
- Through the filesystem: file descriptors, read()/write(), pipes

- Implies: ipc construct shared thru normal process hierarchy inheritance rules; pipes are nameless (in most Unix dialects)

- 

- Totally reliable *byte stream* between producer and consumer.

“Standard” Unix Processes/IPC

- Through the filesystem: file descriptors, read()/write(), pipes
- Implies: ipc construct shared thru normal process hierarchy inheritance rules; pipes are nameless (in most Unix dialects)
- 

- Totally reliable *byte stream* between producer and consumer.
- Tie-in to conventional Unix semantics of process creation and termination.

We want:

- A real *generalization* of the conventional pipe construct to allow network-based i/o. This means that file descriptors and read()/write() are still going to work.

We want:

- A real *generalization* of the conventional pipe construct to allow network-based i/o. This means that file descriptors and read()/write() are still going to work.
- Need some extra features to take into account: (1) network protocol stacks, (2) network naming conventions, (3) requirements of protocol-specific message passing.

We want:

- A real *generalization* of the conventional pipe construct to allow network-based i/o. This means that file descriptors and read()/write() are still going to work.
- Need some extra features to take into account: (1) network protocol stacks, (2) network naming conventions, (3) requirements of protocol-specific message passing.
- The BSD solution: `socket()`. Most concise definition is simply that of a *communication endpoint*. Can be accessed through a file descriptor.

We want:

- A real *generalization* of the conventional pipe construct to allow network-based i/o. This means that file descriptors and read()/write() are still going to work.
- Need some extra features to take into account: (1) network protocol stacks, (2) network naming conventions, (3) requirements of protocol-specific message passing.
- The BSD solution: `socket()`. Most concise definition is simply that of a *communication endpoint*. Can be accessed through a file descriptor.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Communication *domain*:

Basically specifies a *protocol stack*. Some Unices implement a richer set of commo domains than others.

Communication *domain*:

Basically specifies a *protocol stack*. Some Unices implement a richer set of commo domains than others.

- `AF_UNIX`: the Unix IPC domain, local to single machine

Communication *domain*:

Basically specifies a *protocol stack*. Some Unices implement a richer set of commo domains than others.

- `AF_UNIX`: the Unix IPC domain, local to single machine
- `AF_INET`: the Internet domain, global in scope

Communication *domain*:

Basically specifies a *protocol stack*. Some Unices implement a richer set of commo domains than others.

- `AF_UNIX`: the Unix IPC domain, local to single machine
- `AF_INET`: the Internet domain, global in scope
- `AF_NS`: the Xerox NS protocol family

Communication *domain*:

Basically specifies a *protocol stack*. Some Unices implement a richer set of commo domains than others.

- `AF_UNIX`: the Unix IPC domain, local to single machine
- `AF_INET`: the Internet domain, global in scope
- `AF_NS`: the Xerox NS protocol family
- `AF_ISO`: the ISO protocol family

Communication *domain*:

Basically specifies a *protocol stack*. Some Unices implement a richer set of commo domains than others.

- `AF_UNIX`: the Unix IPC domain, local to single machine
- `AF_INET`: the Internet domain, global in scope
- `AF_NS`: the Xerox NS protocol family
- `AF_ISO`: the ISO protocol family

Once a domain is specified, know how to associate a name with the socket.

Communication *domain*:

Basically specifies a *protocol stack*. Some Unices implement a richer set of commo domains than others.

- `AF_UNIX`: the Unix IPC domain, local to single machine
- `AF_INET`: the Internet domain, global in scope
- `AF_NS`: the Xerox NS protocol family
- `AF_ISO`: the ISO protocol family

Once a domain is specified, know how to associate a name with the socket.

Once a domain is specified, know the semantics of supported IPC mechanisms

Unix Domain Sockets (in brief)

Start with the (less interesting) case of the `AF_UNIX` commo domain. Header file `<sys/un.h>` defines addresses.

Unix Domain Sockets (in brief)

Start with the (less interesting) case of the `AF_UNIX` commo domain. Header file `<sys/un.h>` defines addresses.

```
#define UNIX_PATH_MAX   108

struct sockaddr_un {
        unsigned short sun_family;      /* AF_UNIX */
        char sun_path[UNIX_PATH_MAX];   /* pathname */
};
```

Unix Domain Sockets (in brief)

Start with the (less interesting) case of the `AF_UNIX` commo domain. Header file `<sys/un.h>` defines addresses.

```
#define UNIX_PATH_MAX   108

struct sockaddr_un {
        unsigned short sun_family;      /* AF_UNIX */
        char sun_path[UNIX_PATH_MAX];   /* pathname */
};
```

For example:

```
curly% ls -l /dev/printer
srwxrwxrwx  1 root            0 Feb 26 08:49 /dev/printer
```

Unix Domain Sockets (in brief)

Start with the (less interesting) case of the `AF_UNIX` commo domain. Header file `<sys/un.h>` defines addresses.

```
#define UNIX_PATH_MAX   108

struct sockaddr_un {
        unsigned short sun_family;      /* AF_UNIX */
        char sun_path[UNIX_PATH_MAX];   /* pathname */
};
```

For example:

```
curly% ls -l /dev/printer
srwxrwxrwx  1 root            0 Feb 26 08:49 /dev/printer
```

`/dev/printer` is a Unix domain socket used by the printer spooler subsystem.

Two *type*s of sockets available in Unix commo domain:

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Unix may discard datagrams in times of buffer congestion! Connectionless.

Two *type*s of sockets available in Unix commo domain:

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Unix may discard datagrams in times of buffer congestion! Connectionless.
- `SOCK_STREAM` implements a *virtual circuit*; reliable FIFO point-to-point commo. Appears as a byte stream to applications. Actually, this is the way 4.1bsd+ Unix implements pipes. Point-to-point connection.

Two *type*s of sockets available in Unix commo domain:

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Unix may discard datagrams in times of buffer congestion! Connectionless.
- `SOCK_STREAM` implements a *virtual circuit*; reliable FIFO point-to-point commo. Appears as a byte stream to applications. Actually, this is the way 4.1bsd+ Unix implements pipes. Point-to-point connection.

Choose socket type in accordance with needs of application. Program in accordance with well-specified delivery semantics of chosen type.

Operations on sockets:

- Binding a name to a socket:

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Operations on sockets:

- Binding a name to a socket:

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- Sending datagram on a socket (*asynchronous*):

```
int  sendto(int s, const void *msg, int len, unsigned int flags,
            const struct sockaddr *to, int tolen);
```

Operations on sockets:

- Binding a name to a socket:

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- Sending datagram on a socket (*asynchronous*):

```
int  sendto(int s, const void *msg, int len, unsigned int flags,
            const struct sockaddr *to, int tolen);
```

- Receiving datagram from a socket (**synchronous, blocking**):

```
int  recvfrom(int s, void *buf, int len, unsigned int flags,
              struct sockaddr *from, int *fromlen);
```

Operations on sockets:

- Binding a name to a socket:

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- Sending datagram on a socket (*asynchronous*):

```
int  sendto(int s, const void *msg, int len, unsigned int flags,
            const struct sockaddr *to, int tolen);
```

- Receiving datagram from a socket (**synchronous, blocking**):

```
int  recvfrom(int s, void *buf, int len, unsigned int flags,
              struct sockaddr *from, int *fromlen);
```

The programs `unix_wdgram.c` and `unix_rdgram.c` illustrate the use of these constructs to build a simple client/server system based on Unix domain datagrams. (Warning ... error checking deleted from the code to make smaller slides.)

unix_rdgram.c: Server

```
#include <errno.h>
#include <strings.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>

main()
{
  short p_len;
  int socket_fd, cc, h_len, fsize, namelen;
  void printsun();
  struct sockaddr_un s_un, from;
  size_t addrlength;

  struct {
    char head;
    u_long body;
    char tail;
  } msg;

  socket_fd = socket (AF_UNIX, SOCK_DGRAM, 0);

  s_un.sun_family = AF_UNIX;
  strcpy(s_un.sun_path, "udgram");
  addrlength = sizeof(s_un.sun_family) + sizeof(s_un.sun_path); /* Note! */

  unlink("udgram");  /* Just in case ... */

  bind(socket_fd, (struct sockaddr *)&s_un, addrlength)

  for(;;) {
   fsize = sizeof(from);
   cc = recvfrom(socket_fd,&msg,sizeof(msg),0,(struct sockaddr *)&from,
                 &fsize);
   printsun( &from, "unix_rdgram: ", "Packet from:");
   printf("Got data ::%c%ld%c\n",msg.head,msg.body,msg.tail);fflush(stdout);
  }
}
```

```
void printsun(Sun, s1, s2)
struct sockaddr_un *Sun; char *s1, *s2;
{
 printf ("%s %s:\n", s1, s2);
 printf ("    family <%d> addr <%s>\n", Sun->sun_family, Sun->sun_path);
}
```

unix_wdgram.c: Client

```
#include <errno.h>
#include <strings.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

main()
{
  int socket_fd, cc;
  long getpid();
  struct sockaddr_un dest;

  struct {
    char head;
    u_long body;
    char tail;
  } msgbuf;

  socket_fd = socket (AF_UNIX, SOCK_DGRAM, 0);

  dest.sun_family = AF_UNIX;
  strcpy(dest.sun_path, "udgram");
  msgbuf.head = '<';
  msgbuf.body = (u_long) getpid();
  msgbuf.tail = '>';

  cc = sendto(socket_fd,&msgbuf,sizeof(msgbuf),0,
                   (struct sockaddr *)&dest,sizeof(dest));
}
```

Sockets and the Internet (IPv4)

`AF_INET` commo domain.

Sockets and the Internet (IPv4)

`AF_INET` commo domain.

Two *type*s of sockets available in Internet commo domain (like in Unix domain):

Sockets and the Internet (IPv4)

`AF_INET` commo domain.

Two *type*s of sockets available in Internet commo domain (like in Unix domain):

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Systems/routers may discard datagrams in times of buffer congestion! Connectionless. **UDP/IP**.

Sockets and the Internet (IPv4)

`AF_INET` commo domain.

Two *type*s of sockets available in Internet commo domain (like in Unix domain):

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Systems/routers may discard datagrams in times of buffer congestion! Connectionless. **UDP/IP**.
- `SOCK_STREAM` implements a *virtual circuit*; reliable FIFO point-to-point commo. Appears as a byte stream to applications. Pipe-like. Point-to-point connection. **TCP/IP**.

Sockets and the Internet (IPv4)

`AF_INET` commo domain.

Two *type*s of sockets available in Internet commo domain (like in Unix domain):

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Systems/routers may discard datagrams in times of buffer congestion! Connectionless. **UDP/IP**.
- `SOCK_STREAM` implements a *virtual circuit*; reliable FIFO point-to-point commo. Appears as a byte stream to applications. Pipe-like. Point-to-point connection. **TCP/IP**.

For example:

```
    sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
```

Sockets and the Internet (IPv4)

`AF_INET` commo domain.

Two *type*s of sockets available in Internet commo domain (like in Unix domain):

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Systems/routers may discard datagrams in times of buffer congestion! Connectionless. **UDP/IP**.
- `SOCK_STREAM` implements a *virtual circuit*; reliable FIFO point-to-point commo. Appears as a byte stream to applications. Pipe-like. Point-to-point connection. **TCP/IP**.

For example:

```
    sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
```

Analogous to an `open()` in that the “file decsriptor” which is returned serves as a handle for future i/o on the socket.

Sockets and the Internet (IPv4)

`AF_INET` commo domain.

Two *type*s of sockets available in Internet commo domain (like in Unix domain):

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Systems/routers may discard datagrams in times of buffer congestion! Connectionless. **UDP/IP**.
- `SOCK_STREAM` implements a *virtual circuit*; reliable FIFO point-to-point commo. Appears as a byte stream to applications. Pipe-like. Point-to-point connection. **TCP/IP**.

For example:

```
sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
```

Analogous to an `open()` in that the "file decsriptor" which is returned serves as a handle for future i/o on the socket.

In order to do network i/o through a socket fd, need a way to associate names with sockets.

Sockets and the Internet (IPv4)

`AF_INET` commo domain.

Two *types* of sockets available in Internet commo domain (like in Unix domain):

- `SOCK_DGRAM` provides *datagram* commo semantics; only best-effort delivery promised. Systems/routers may discard datagrams in times of buffer congestion! Connectionless. **UDP/IP**.
- `SOCK_STREAM` implements a *virtual circuit*; reliable FIFO point-to-point commo. Appears as a byte stream to applications. Pipe-like. Point-to-point connection. **TCP/IP**.

For example:

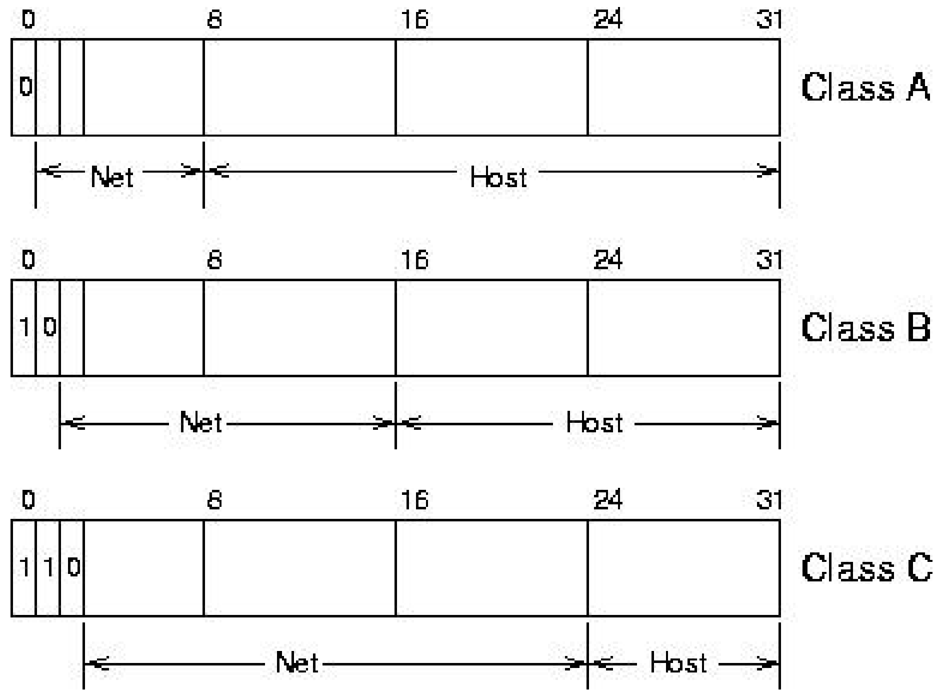
```
    sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
```

Analogous to an `open()` in that the "file decsriptor" which is returned serves as a handle for future i/o on the socket.

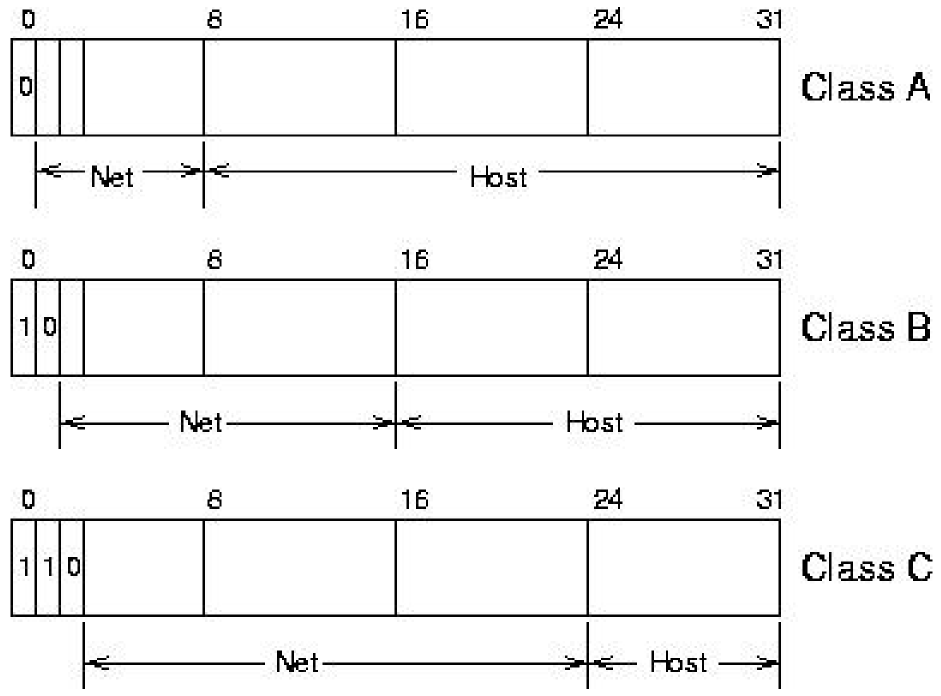
In order to do network i/o through a socket fd, need a way to associate names with sockets.

Header file `<netinet/in.h>` defines a 32-bit address for an Internet host. Actually identifies a specific network interface on a specific system on the Internet. 32-bit number.

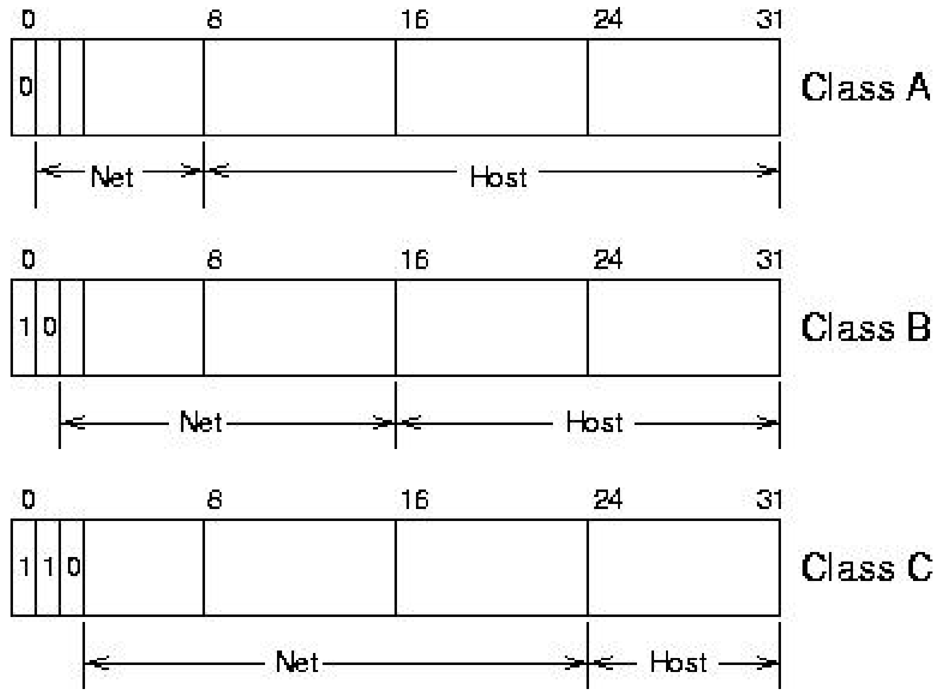
```
struct in_addr {
        __u32   s_addr;
};
```



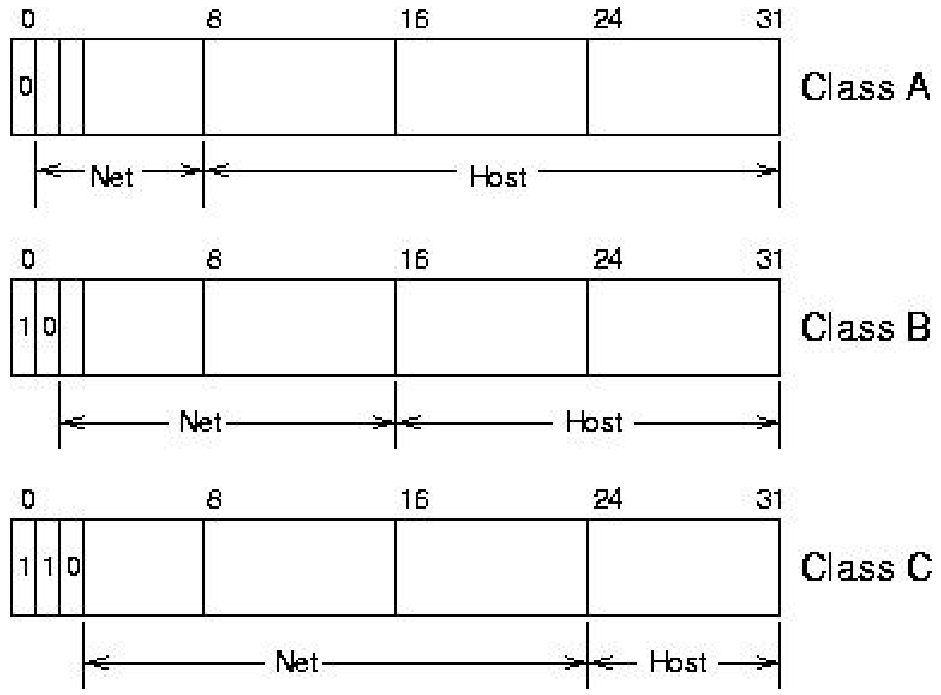
- Class D for multicast; Class E is reserved; Classless.



- Class D for multicast; Class E is reserved; Classless.
- Dotted decimal notation.



- Class D for multicast; Class E is reserved; Classless.
- Dotted decimal notation.
- Net name—host part all 0's; broadcast address—host part all 1's (root only);



- Class D for multicast; Class E is reserved; Classless.
- Dotted decimal notation.
- Net name—host part all 0's; broadcast address—host part all 1's (root only);
- Localhost—127.0.0.1

- In header file `<netinet/in.h>`:

```
#define __SOCK_SIZE__   16          /* sizeof(struct sockaddr)      */

struct sockaddr_in {
 short int          sin_family;     /* Address family               */
 unsigned short int sin_port;       /* Port number                  */
 struct in_addr     sin_addr;       /* Internet address             */

 /* Pad to size of ‘struct sockaddr’. */
 unsigned char      __pad[__SOCK_SIZE__ - sizeof(short int) -
                    sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

- In header file `<netinet/in.h>`:

```
#define __SOCK_SIZE__   16          /* sizeof(struct sockaddr)      */

struct sockaddr_in {
 short int          sin_family;     /* Address family               */
 unsigned short int sin_port;       /* Port number                  */
 struct in_addr     sin_addr;       /* Internet address             */

 /* Pad to size of ‘struct sockaddr’. */
 unsigned char      __pad[__SOCK_SIZE__ - sizeof(short int) -
                    sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

- Declare/allocate instance of struct sockaddr_in whenever you need to specify a full adress on the Internet.

- In header file <netinet/in.h>:

```
#define __SOCK_SIZE__   16          /* sizeof(struct sockaddr)      */

struct sockaddr_in {
 short int          sin_family;     /* Address family               */
 unsigned short int sin_port;       /* Port number                  */
 struct in_addr     sin_addr;       /* Internet address             */

 /* Pad to size of ‘struct sockaddr’. */
 unsigned char      __pad[__SOCK_SIZE__ - sizeof(short int) -
                    sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

- Declare/allocate instance of struct sockaddr_in whenever you need to specify a full adress on the Internet.
- A port is an Internet commo endpoint associated with an application. (host,port) defines an Internet address.

- In header file <netinet/in.h>:

```
#define __SOCK_SIZE__   16          /* sizeof(struct sockaddr)      */

struct sockaddr_in {
 short int          sin_family;     /* Address family               */
 unsigned short int sin_port;       /* Port number                  */
 struct in_addr     sin_addr;       /* Internet address             */

 /* Pad to size of `struct sockaddr'. */
 unsigned char      __pad[__SOCK_SIZE__ - sizeof(short int) -
                    sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

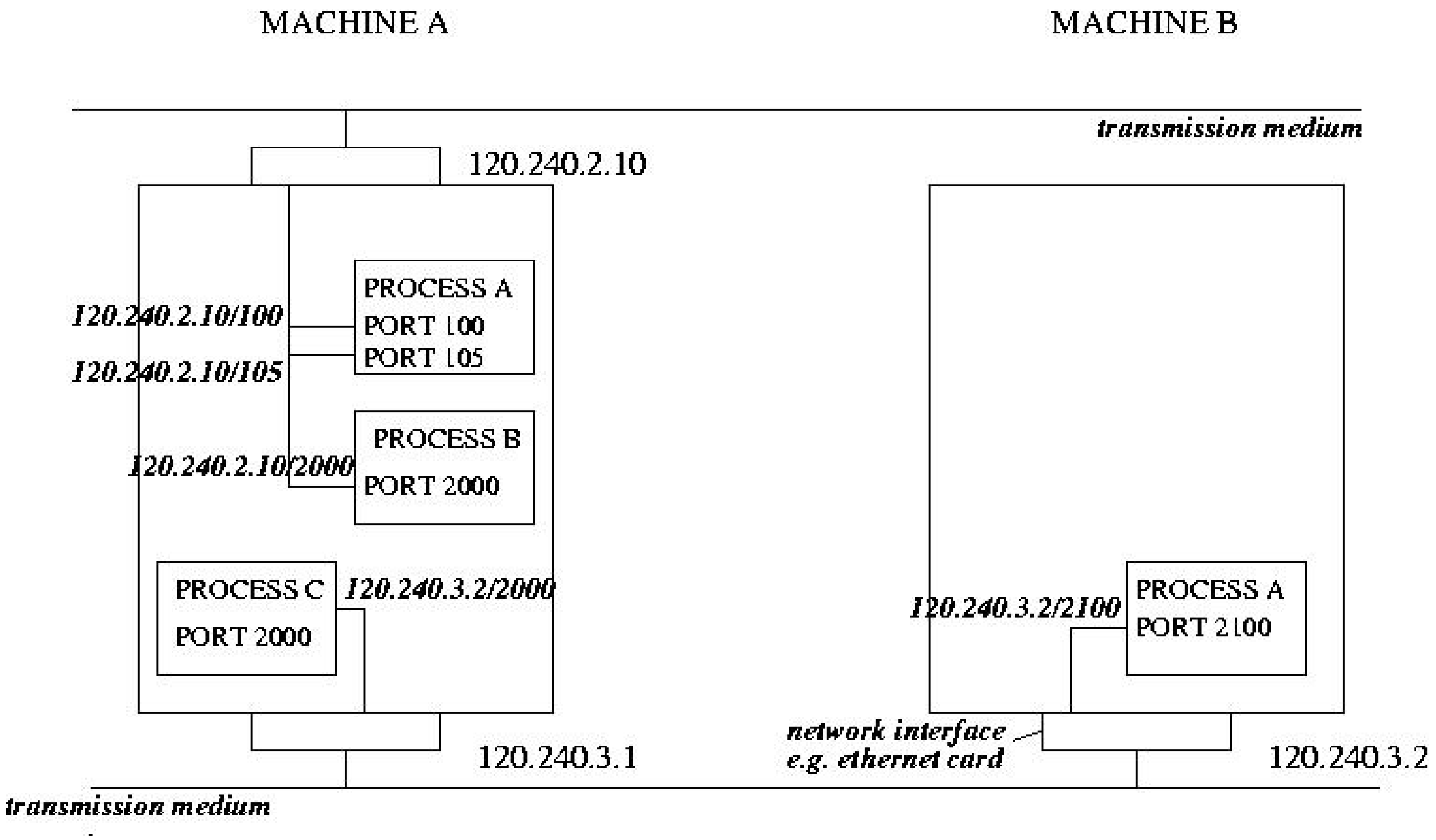
- Declare/allocate instance of struct sockaddr_in whenever you need to specify a full adress on the Internet.
- A port is an Internet commo endpoint associated with an application. (host,port) defines an Internet address.
- Ports in the range [0,1023] reserved for root; others available to ordinary users. (See RFC 1700, IANA *http://www.iana.com/numbers.html*)

- In header file `<netinet/in.h>`:

```
#define __SOCK_SIZE__   16          /* sizeof(struct sockaddr)      */

struct sockaddr_in {
 short int          sin_family;     /* Address family               */
 unsigned short int sin_port;       /* Port number                  */
 struct in_addr     sin_addr;       /* Internet address             */

 /* Pad to size of ‘struct sockaddr’. */
 unsigned char      __pad[__SOCK_SIZE__ - sizeof(short int) -
                    sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

- Declare/allocate instance of struct sockaddr_in whenever you need to specify a full adress on the Internet.
- A port is an Internet commo endpoint associated with an application. (host,port) defines an Internet address.
- Ports in the range [0,1023] reserved for root; others available to ordinary users. (See RFC 1700, IANA *http://www.iana.com/numbers.html*)

- ftp uses 20 & 21; telnet uses 23; finger uses 79; rlogin uses 513; talk uses 517 . . . see `/etc/services` ("well-known" ports).



MACHINE A
MACHINE B
transmission medium
120.240.2.10
PROCESS A
PORT 100
PORT 105
120.240.2.10/100
120.240.2.10/105
PROCESS B
PORT 2000
120.240.2.10/2000
PROCESS C
PORT 2000
120.240.3.2/2000
120.240.3.2/2100
PROCESS A
PORT 2100
network interface
e.g. ethernet card
120.240.3.1
120.240.3.2
transmission medium

Library function to map symbolic host name into IP address(es):

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name)

void herror(const char *s);
```

Library function to map symbolic host name into IP address(es):

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name)

void herror(const char *s);
```

Hostent data structure:

```
struct hostent {
  char    *h_name;        /* official name of host */
  char    **h_aliases;    /* alias list */
  int     h_addrtype;     /* host address type */
  int     h_length;       /* length of address */
  char    **h_addr_list;  /* list of addresses */
}
#define h_addr  h_addr_list[0]
```

Useful for printing IP addresses and turning “dotted decimal” strings into IP addresses (see UPM inet(3) for more info):

Useful for printing IP addresses and turning “dotted decimal” strings into IP addresses (see UPM inet(3) for more info):

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

char *inet_ntoa(struct in_addr in);

int inet_aton(const char *cp, struct in_addr *inp);
```

getaddrs.c: Get Host Info From Symbolic Name

getaddrs.c: Get Host Info From Symbolic Name

```
#include <netdb.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
main(argc,argv)
int argc; char **argv;

  struct hostent *entry; char **next; struct in_addr address, **addrptr;
  entry = gethostbyname(argv[1]);
  if (!entry) { herror("lookup error"); exit(1);}
  printf("Official name -> %s\n", entry->h_name);
  if (entry->h_aliases[0]) {
    printf("Aliases ->\n");
    for (next = entry->h_aliases; *next; next++)
      printf("     %s\n", *next);
  }
  printf("IP Addresses:\n");
  for (addrptr=(struct in_addr **) entry->h_addr_list; *addrptr; addrptr++)
    printf("     %s\n", inet_ntoa(**addrptr));
}
```

```
anthony.csl% ./getaddrs anthony
Official name -> anthony.csl.mtu.edu
Aliases:
     anthony.csl
     anthony
     cslab21
IP Addresses:
     141.219.150.190
```

```
anthony.csl% ./getaddrs anthony
Official name -> anthony.csl.mtu.edu
Aliases:
     anthony.csl
     anthony
     cslab21
IP Addresses:
     141.219.150.190
anthony.csl% ./getaddrs www.linux.org
Official name -> www.linux.org
IP Addresses:
     198.182.196.56
```

Inverse function (know IP adress, want symbolic name):

```
#include <netdb.h>

struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Inverse function (know IP adress, want symbolic name):

```
#include <netdb.h>

struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

gethost.c: Get Host Info From IP Address

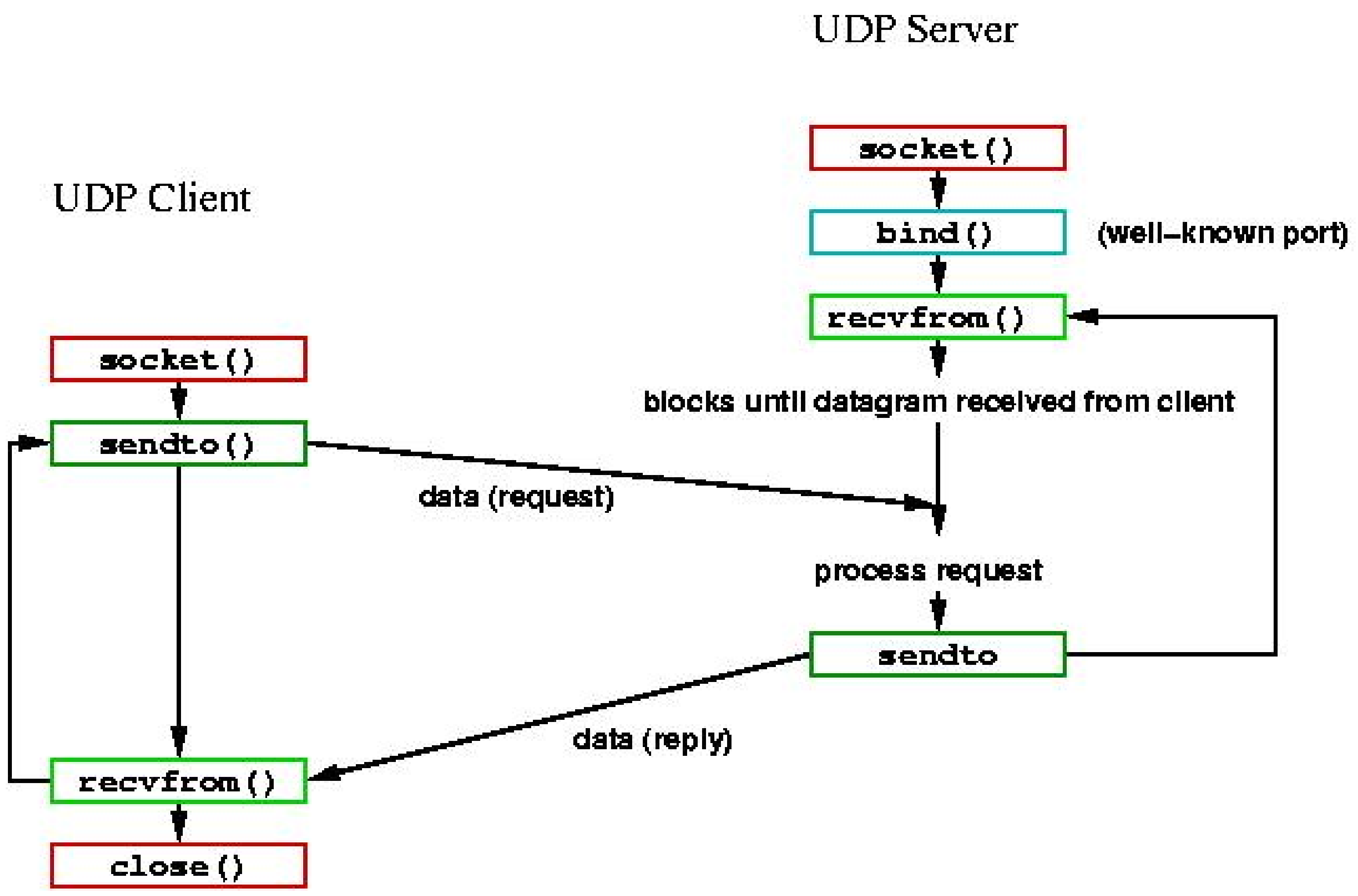
Inverse function (know IP adress, want symbolic name):

```
#include <netdb.h>

struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

gethost.c: Get Host Info From IP Address

```
#include <netdb.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
main(argc,argv)
int argc; char **argv;
{
  struct hostent *entry; struct in_addr address; char **next;
  inet_aton(argv[1], &address);
    entry = gethostbyaddr((char *)&address,sizeof(address),AF_INET);
  if (!entry) { herror("lookup error"); exit(1);}

  . . . like getaddrs.c . . .
```

```
anthony.csl% ./gethost 141.219.150.190
Official name -> anthony.csl.mtu.edu
Aliases:
     anthony.csl
     anthony
     cslab21
IP Addresses:
     141.219.150.190
```


UDP Server
UDP Client
socket()
bind()
(well-known port)
recvfrom()
blocks until datagram received from client
socket()
sendto()
data (request)
process request
sendto
data (reply)
recvfrom()
close()

recv_udp.c: UDP/IP Server

recv_udp.c: UDP/IP Server

```
main()
{
  short p_len;
  int socket_fd, cc, h_len, fsize, namelen;
  struct sockaddr_in  s_in, from;
  struct { char head; u_long  body; char tail;} msg;

  socket_fd = socket (AF_INET, SOCK_DGRAM, 0);

  bzero((char *) &s_in, sizeof(s_in));  /* They say you must do this    */
  s_in.sin_family = (short)AF_INET;
  s_in.sin_addr.s_addr = htonl(INADDR_ANY);    /* WILDCARD */
  s_in.sin_port = htons((u_short)0x3333);
  printsin( &s_in, "RECV_UDP", "Local socket is:"); fflush(stdout);
  bind(socket_fd, (struct sockaddr *)&s_in, sizeof(s_in));
  for(;;) {
    fsize = sizeof(from);
    cc = recvfrom(socket_fd,&msg,sizeof(msg),0,(struct sockaddr *)&from,&fsize);
    printsin( &from, "recv_udp: ", "Packet from:");
    printf("Got data ::%c%ld%c\n",msg.head,ntohl(msg.body),msg.tail); fflush(stdout);
  }
}
```

send_udp.c: UDP/IP Client

send_udp.c: UDP/IP Client

```
main(argc,argv)
int argc; char **argv;
{
  int socket_fd;
  struct sockaddr_in  dest;
  struct hostent *hostptr;

  struct { char head; u_long body; char tail; } msgbuf;

  socket_fd = socket (AF_INET, SOCK_DGRAM, 0);

  bzero((char *) &dest, sizeof(dest)); /* They say you must do this */
  hostptr = gethostbyname(argv[1]);
  dest.sin_family = (short) AF_INET;
  bcopy(hostptr->h_addr, (char *)&dest.sin_addr,hostptr->h_length);
  dest.sin_port = htons((u_short)0x3333);

  msgbuf.head = '<';
  msgbuf.body = htonl(getpid());    /* IMPORTANT! */
  msgbuf.tail = '>';

  sendto(socket_fd,&msgbuf,sizeof(msgbuf),0,(struct sockaddr *)&dest,
                  sizeof(dest));
}
```

Note striking similarities and simple differences between Unix datagram programs and Internet datagram programs. (Extends local IPC into networked IPC relatively seamlessly).

Note striking similarities and simple differences between Unix datagram programs and Internet datagram programs. (Extends local IPC into networked IPC relatively seamlessly).

Different socket creation parameters (trivial).

Note striking similarities and simple differences between Unix datagram programs and Internet datagram programs. (Extends local IPC into networked IPC relatively seamlessly).

Different socket creation parameters (trivial).

Different naming conventions (significant).

Note striking similarities and simple differences between Unix datagram programs and Internet datagram programs. (Extends local IPC into networked IPC relatively seamlessly).

Different socket creation parameters (trivial).

Different naming conventions (significant).

Of course, underlying implementation is completely different (but generally hidden from programmer).

Note striking similarities and simple differences between Unix datagram programs and Internet datagram programs. (Extends local IPC into networked IPC relatively seamlessly).

Different socket creation parameters (trivial).

Different naming conventions (significant).

Of course, underlying implementation is completely different (but generally hidden from programmer).

Impt. practical note: can always open up Internet ports on “localhost” (127.0.0.1) to test/develop network software. Implementation should be smart enough not to put packets on wire (move from output buffer to input buffer).

Want a much fancier application. Namely:

- Receiver will shut down cleanly if no datagram received after a 1 minute interval. Will also shut down cleanly if receives anything on stdin.

Want a much fancier application. Namely:

- Receiver will shut down cleanly if no datagram received after a 1 minute interval. Will also shut down cleanly if receives anything on stdin.
- Basic problem . . . hanging a read/recv from several descriptors at once.

Want a much fancier application. Namely:

- Receiver will shut down cleanly if no datagram received after a 1 minute interval. Will also shut down cleanly if receives anything on stdin.
- Basic problem . . . hanging a read/recv from several descriptors at once.
- Solution thru kernel call:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int  select(int  n, fd_set *readfds, fd_set *writefds,
              fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

fancy_recv_udp.c: Fancy UDP Server

fancy_recv_udp.c: Fancy UDP Server

```
main()
{
  int socket_fd, cc, h_len, fsize, namelen, hits;
  fd_set mask;
  struct timeval timeout;
  struct sockaddr_in  s_in, from;

  struct { char head; u_long body; char tail; } msg;

  socket_fd = socket (AF_INET, SOCK_DGRAM, 0);

  bzero((char *) &s_in, sizeof(s_in));  /* They say you must do this    */
  s_in.sin_family = (short) AF_INET;
  s_in.sin_addr.s_addr = htons(INADDR_ANY);    /* WILDCARD */
  s_in.sin_port = htonl((u_short)0x3333);
  bind(socket_fd, (struct sockaddr *)&s_in, sizeof(s_in));
  for(;;) {
    fsize = sizeof(from);
    FD_ZERO(&mask); FD_SET(0,&mask); FD_SET(socket_fd,&mask);
    timeout.tv_sec = 60; timeout.tv_usec = 0;

    if ((hits = select(socket_fd+1, &mask, (fd_set *)0, (fd_set *)0,
                       &timeout)) < 0) {
      perror("recv_udp:select"); exit(1);
    }


    if ( (hits==0) || ((hits>0) && (FD_ISSET(0,&mask))) ) {
      printf("Shutting down\n"); exit(0);
    }
    cc = recvfrom(socket_fd,&msg,sizeof(msg),0,
                          (struct sockaddr *)&from,&fsize);
    printsin(&from, "recv_udp: ", "Packet from:");
    printf("Got data ::%c%ld%c\n",msg.head,ntohl(msg.body),msg.tail);
    fflush(stdout);
  }
}
```

Internet Virtual Circuits (TCP/IP)

Want to extend pipe abstraction into the Internet. This means a *reliable* byte stream with no visible packets/messages.

Internet Virtual Circuits (TCP/IP)

Want to extend pipe abstraction into the Internet. This means a *reliable* byte stream with no visible packets/messages.

Implies point-to-point connection. In entire Internet, the tuple

$$((\text{host1},\text{port1}),\ (\text{host2},\text{port2}))$$

is unique. Note that point-to-point means process-to-process.

Internet Virtual Circuits (TCP/IP)

Want to extend pipe abstraction into the Internet. This means a *reliable* byte stream with no visible packets/messages.

Implies point-to-point connection. In entire Internet, the tuple

$$((\text{host1},\text{port1}),\ (\text{host2},\text{port2}))$$

is unique. Note that point-to-point means process-to-process.

Note that there is a non-trivial cost in setting up the connection, maintaining reliable transmission on the connection, and tearing down the connection.

Internet Virtual Circuits (TCP/IP)

Want to extend pipe abstraction into the Internet. This means a *reliable* byte stream with no visible packets/messages.

Implies point-to-point connection. In entire Internet, the tuple

$$((\mathrm{host1},\mathrm{port1}),\ (\mathrm{host2},\mathrm{port2}))$$

is unique. Note that point-to-point means process-to-process.

Note that there is a non-trivial cost in setting up the connection, maintaining reliable transmission on the connection, and tearing down the connection.

Client-server model implicit.

New kernel calls:

To mark socket as being capable of accepting TCP/IP connections, server does:

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

New kernel calls:

To mark socket as being capable of accepting TCP/IP connections, server does:

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

To accept a TCP/IP connection on a listening socket, server can then do:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, int *addrlen);
```

New kernel calls:

To mark socket as being capable of accepting TCP/IP connections, server does:

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

To accept a TCP/IP connection on a listening socket, server can then do:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, int *addrlen);
```

To initiate a connection to a server, client does:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int  addrlen );
```

New kernel calls:

To mark socket as being capable of accepting TCP/IP connections, server does:

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

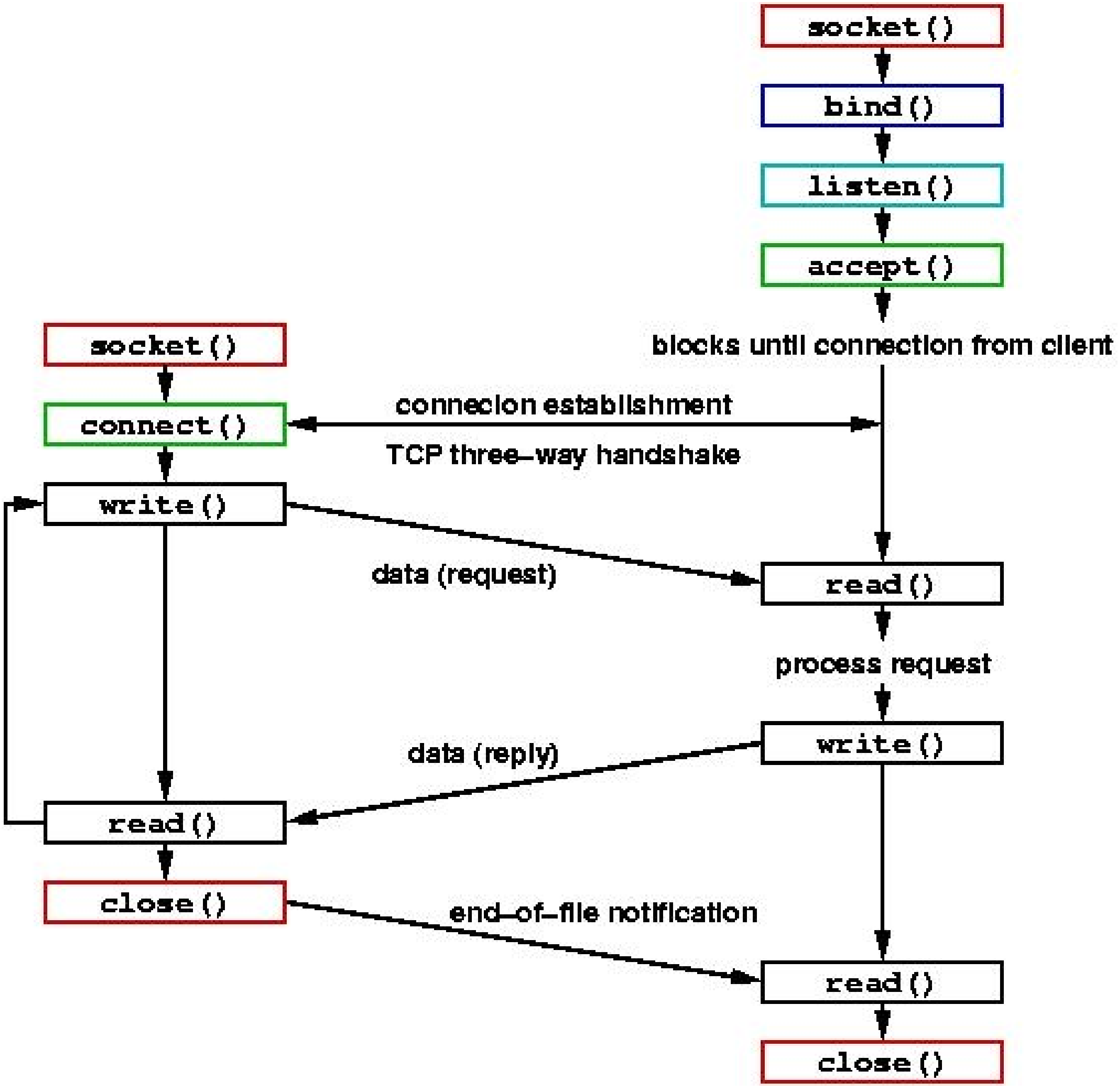
To accept a TCP/IP connection on a listening socket, server can then do:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, int *addrlen);
```

To initiate a connection to a server, client does:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int  addrlen );
```

Can then use `read` and `write` to pass data on the virtual circuit.


socket()
bind()
listen()
accept()
blocks until connection from client
socket()
connect()
connecion establishment
TCP three–way handshake
write()
data (request)
read()
process request
write()
data (reply)
read()
close()
end–of–file notification
read()
close()

inet_wstream.c TCP/IP Client

inet_wstream.c TCP/IP Client

```
char msg[] = { "false pearls before real swine" };
main(argc, argv)
int argc; char **argv;
{
  char *remhost; u_short remport;
  int sock, left, num, put;
  struct sockaddr_in remote;
  struct hostent *h;

  remhost = argv[1];  remport = atoi(argv[2]);

  sock = socket( AF_INET, SOCK_STREAM, 0 );

  bzero((char *) &remote, sizeof(remote));
  remote.sin_family = AF_INET;
  h = gethostbyname(remhost);
  bcopy((char *)h->h_addr, (char *)&remote.sin_addr, h->h_length);
  remote.sin_port = htons(remport);
  connect(sock, (struct sockaddr *)&remote, sizeof(remote));

/* can’t guarantee socket will accept all we try to write, cope */

  left = sizeof(msg); put=0;
  while (left > 0){
    if((num = write(sock, msg+put , left)) < 0) {
      perror("inet_wstream:write");
      exit(1);
    }
    else {
      left -= num;
      put += num;
    }
  }
}
```

inet_rstream.c TCP/IP Server

inet_rstream.c TCP/IP Server

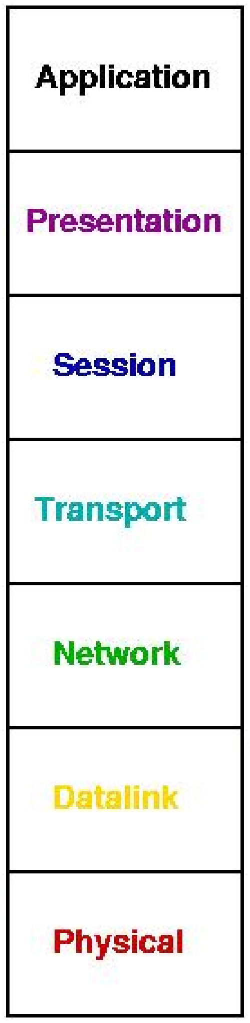
```
main()
{
  int listener, conn, length; char ch;
  struct sockaddr_in s1, s2;

  listener = socket( AF_INET, SOCK_STREAM, 0 );

  bzero((char *) &s1, sizeof(s1));
  s1.sin_family = AF_INET;
  s1.sin_addr.s_addr = htonl(INADDR_ANY);  /* Any of this host’s interfaces is OK. */
  s1.sin_port = htons(0);                  /* bind() will gimme unique port. */
  bind(listener, (struct sockaddr *)&s1, sizeof(s1));
  length = sizeof(s1);
  getsockname(listener, (struct sockaddr *)&s1, &length); /* Find out port number */
  printf("RSTREAM:: assigned port number %d\n", s1.sin_port);

  listen(listener,1);
  length = sizeof(s2);
  conn=accept(listener, (struct sockaddr *)&s2, &length);
  printsin(&s2,"RSTREAM::", "accepted connection from");

  printf("\n\nRSTREAM:: data from stream:\n");
  while ( read(conn, &ch, 1) == 1) putchar(ch);
  putchar(’\n’);
}
```

Typical TCP/IP Client/Server Server

Typical TCP/IP Client/Server Server

```
. . .
listener = socket(...);
. . .
bind(listener, ...);
listen(listener, ...);
...
while (1) {
    new = accept(listener, ...);
    if (fork() == 0) {
        close(listener);
        /* read lots of stuff from new */
    }
    close(new);
    while(wait3(status, WNOHANG, NULL)); /* Can also handle
                                            SIGCHLD */
}
```

OSI Reference Model



Application related protocols

e.g., FTP: handle file naming
convention differences

Services requested frequently

Concerned with syntax and semantics of information
transmission
e.g., data encoding, data compression

Allow users to establish session

Dialogue control
Token management
Synchronization

End–to–end layer

Ensure data from session layer arrives
in order, error free
Data broken into smaller units

Control operation of subnet

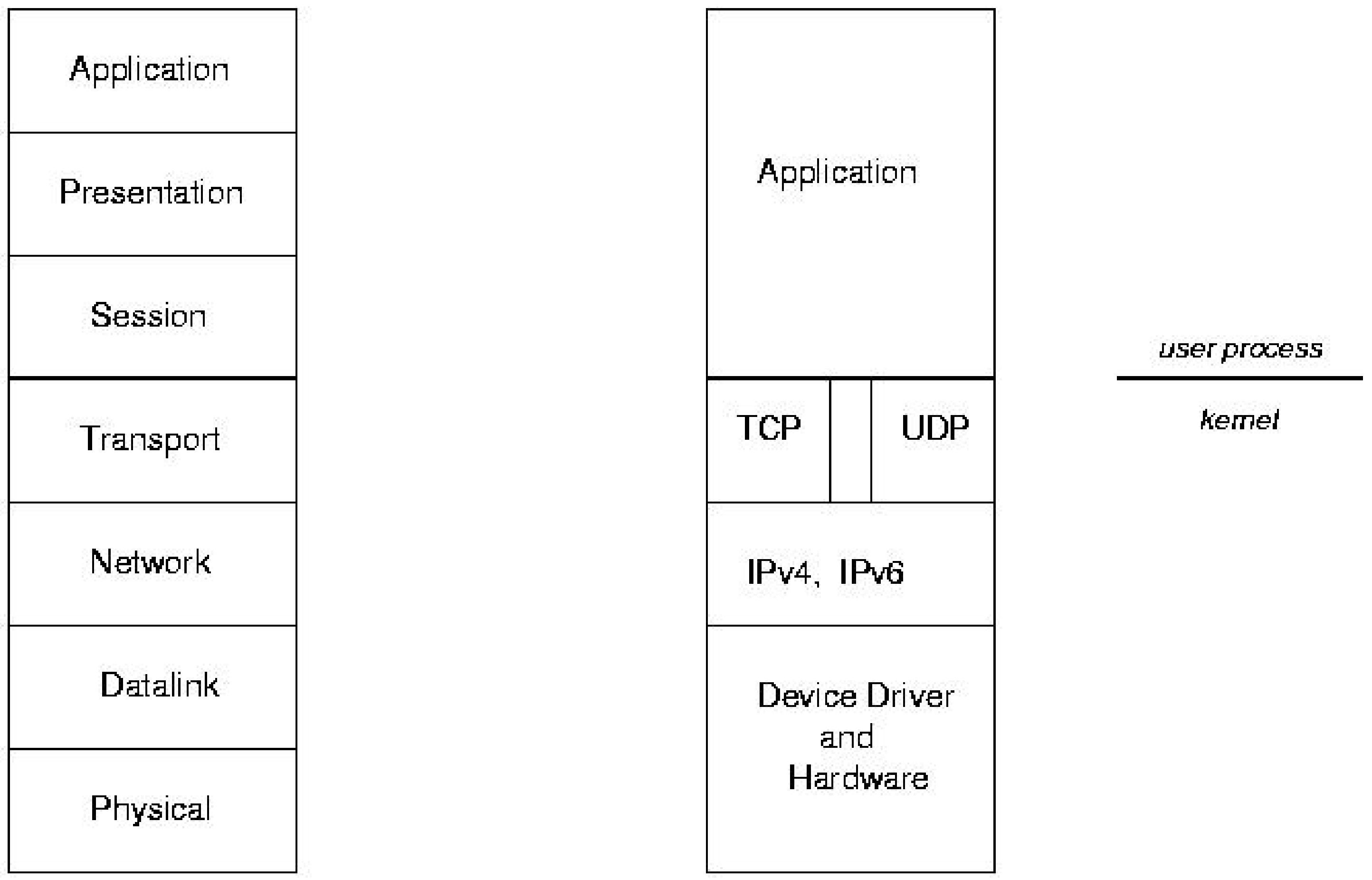
Routing
Congestion control
Accounting

Turn raw transmission into apparently error–free line

Data broken into frames
Create and recognize frame boundaries
Handle lost or duplicated frames

Transmission of raw bits over wire

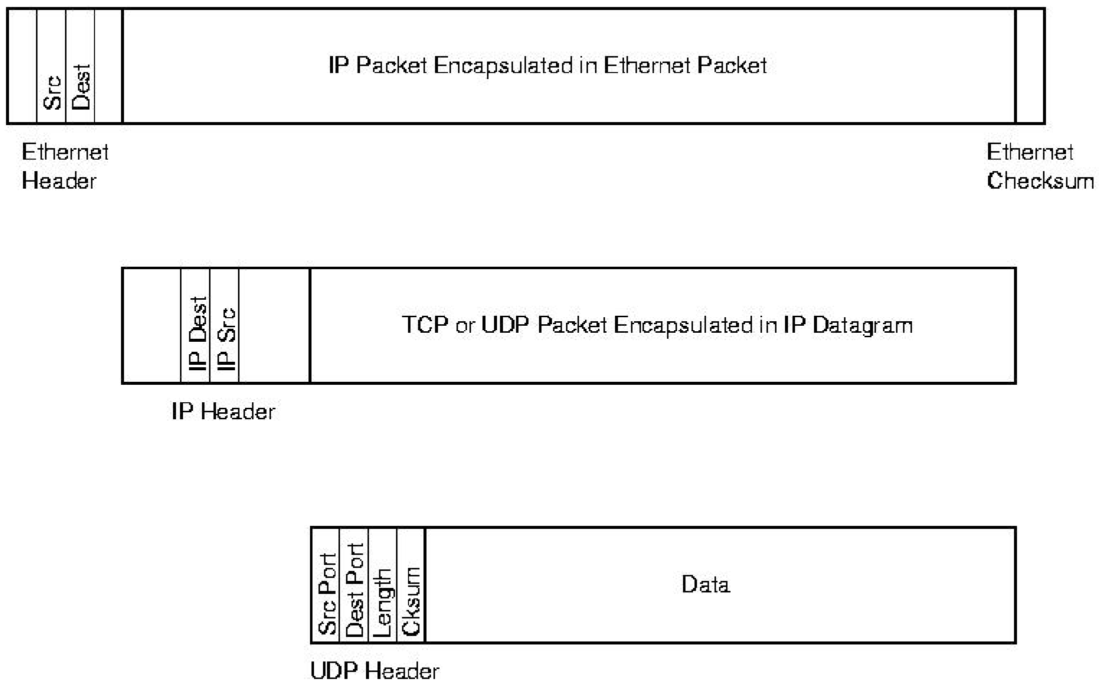
How many volts represent a 1, a 0?
Is transmission bidirectional?



OSI REFERENCE MODEL

INTERNET PROTOCOL SUITE

Packet Encapsulation



Ethernet

- IEEE Standard 802: CSMA/CD, token bus, token ring

Ethernet

- IEEE Standard 802: CSMA/CD, token bus, token ring
- IEEE Standard 802.3: CSMA/CD

Ethernet

- IEEE Standard 802: CSMA/CD, token bus, token ring
- IEEE Standard 802.3: CSMA/CD
  - ⋆ Ethernet is implementation of 802.3

Ethernet

- IEEE Standard 802: CSMA/CD, token bus, token ring
- IEEE Standard 802.3: CSMA/CD
  - ⋆ Ethernet is implementation of 802.3
- Typically 6 byte address: 00:62:97:B8:C9:4A

Ethernet

- IEEE Standard 802: CSMA/CD, token bus, token ring
- IEEE Standard 802.3: CSMA/CD
  - ⋆ Ethernet is implementation of 802.3
- Typically 6 byte address: 00:62:97:B8:C9:4A
- ARP: protocol for mapping IPv4 address the hardware address

Ethernet

- IEEE Standard 802: CSMA/CD, token bus, token ring
- IEEE Standard 802.3: CSMA/CD
  - ⋆ Ethernet is implementation of 802.3
- Typically 6 byte address: 00:62:97:B8:C9:4A
- ARP: protocol for mapping IPv4 address the hardware address
- RARP: protocol for mapping hardware address to IPv4 address

Ethernet

- IEEE Standard 802: CSMA/CD, token bus, token ring
- IEEE Standard 802.3: CSMA/CD
  - ⋆ Ethernet is implementation of 802.3
- Typically 6 byte address: 00:62:97:B8:C9:4A
- ARP: protocol for mapping IPv4 address the hardware address
- RARP: protocol for mapping hardware address to IPv4 address
- Ethernet address outermost
  Routing protocols determine correct IP address on a hop-by-hop basis
  ARP maps IP address to ethernet address to transmit message across next hop

Ethernet

- IEEE Standard 802: CSMA/CD, token bus, token ring
- IEEE Standard 802.3: CSMA/CD
  - ⋆ Ethernet is implementation of 802.3
- Typically 6 byte address: 00:62:97:B8:C9:4A
- ARP: protocol for mapping IPv4 address the hardware address
- RARP: protocol for mapping hardware address to IPv4 address
- Ethernet address outermost
  Routing protocols determine correct IP address on a hop-by-hop basis
  ARP maps IP address to ethernet address to transmit message across next hop

jmayo: arp -a

```
cec.mtu.edu (141.219.151.196) at 08:00:20:1D:16:13 [ether] on eth0
kurosawa.hu.mtu.edu (141.219.148.44) at 00:50:04:A1:D4:C2 [ether] on eth0
cslserver.csl.mtu.edu (141.219.150.71) at 08:00:20:77:32:D6 [ether] on eth0
cs.mtu.edu (141.219.150.12) at 08:00:20:21:A5:D3 [ether] on eth0
brtr11.tc.mtu.edu (141.219.148.1) at 00:00:EF:06:76:30 [ether] on
eth0
```

```
jmayo> ping rainbow.cs.mtu.edu
PING rainbow.cs.mtu.edu (141.219.150.6) from 141.219.150.16 : 56(84) bytes of data.
64 bytes from rainbow.cs.mtu.edu (141.219.150.6): icmp_seq=0 ttl=255 time=1.2 ms64 bytes from rainbo
--- rainbow.cs.mtu.edu ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.5/0.8/1.2 ms
```

```
jmayo> ping rainbow.cs.mtu.edu
PING rainbow.cs.mtu.edu (141.219.150.6) from 141.219.150.16 : 56(84) bytes of data.
64 bytes from rainbow.cs.mtu.edu (141.219.150.6): icmp_seq=0 ttl=255 time=1.2 ms64 bytes from rainbo
--- rainbow.cs.mtu.edu ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.5/0.8/1.2 ms
```

jmayo: arp -a

```
cec.mtu.edu (141.219.151.196) at 08:00:20:1D:16:13 [ether] on eth0
kurosawa.hu.mtu.edu (141.219.148.44) at 00:50:04:A1:D4:C2 [ether] on eth0
cslserver.csl.mtu.edu (141.219.150.71) at 08:00:20:77:32:D6 [ether] on eth0
cs.mtu.edu (141.219.150.12) at 08:00:20:21:A5:D3 [ether] on eth0
brtr11.tc.mtu.edu (141.219.148.1) at 00:00:EF:06:76:30 [ether] on eth0
rainbow.cs.mtu.edu (141.219.150.6) at 08:00:20:9C:2F:97 [ether] on eth0
```

jmayo: ping ftp.x.org

```
PING ftp.x.org (198.4.202.8) from 141.219.150.16 : 56(84) bytes of data.
64 bytes from ftp.x.org (198.4.202.8): icmp_seq=0 ttl=51 time=81.0 ms
64 bytes from ftp.x.org (198.4.202.8): icmp_seq=1 ttl=51 time=79.0 ms

--- ftp.x.org ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 79.0/80.0/81.0 ms
```

jmayo: ping ftp.x.org

```
PING ftp.x.org (198.4.202.8) from 141.219.150.16 : 56(84) bytes of data.
64 bytes from ftp.x.org (198.4.202.8): icmp_seq=0 ttl=51 time=81.0 ms
64 bytes from ftp.x.org (198.4.202.8): icmp_seq=1 ttl=51 time=79.0 ms

--- ftp.x.org ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 79.0/80.0/81.0 ms
```

jmayo: arp -a

```
cec.mtu.edu (141.219.151.196) at 08:00:20:1D:16:13 [ether] on eth0
kurosawa.hu.mtu.edu (141.219.148.44) at 00:50:04:A1:D4:C2 [ether] on eth0
cslserver.csl.mtu.edu (141.219.150.71) at 08:00:20:77:32:D6 [ether] on eth0
cs.mtu.edu (141.219.150.12) at 08:00:20:21:A5:D3 [ether] on eth0
brtr11.tc.mtu.edu (141.219.148.1) at 00:00:EF:06:76:30 [ether] on eth0
rainbow.cs.mtu.edu (141.219.150.6) at 08:00:20:9C:2F:97 [ether] on eth0
jmayo>
```

Mapping between Symbolic Name and IP Address

- Domain Name System: maps between hostnames and IP addresses

Mapping between Symbolic Name and IP Address

- Domain Name System: maps between hostnames and IP addresses
  - $Nameserver$ provides resolution service

Mapping between Symbolic Name and IP Address

- Domain Name System: maps between hostnames and IP addresses
  - $Nameserver$ provides resolution service
    - typically BIND (Berkely Internet Name Domain)

Mapping between Symbolic Name and IP Address

- Domain Name System: maps between hostnames and IP addresses
  - $\star$ *Nameserver* provides resolution service
    - $*$ typically BIND (Berkely Internet Name Domain)
    - $*$ `/etc/resolv.conf` - contains nameserver locations

Mapping between Symbolic Name and IP Address

- Domain Name System: maps between hostnames and IP addresses
  - *Nameserver* provides resolution service
    - typically BIND (Berkely Internet Name Domain)
    - `/etc/resolv.conf` - contains nameserver locations
- Other mechanisms:
  - static hosts files, e.g. `/etc/hosts`

Mapping between Symbolic Name and IP Address

- Domain Name System: maps between hostnames and IP addresses
  - *Nameserver* provides resolution service
    - typically BIND (Berkely Internet Name Domain)
    - `/etc/resolv.conf` - contains nameserver locations
- Other mechanisms:
  - static hosts files, e.g. `/etc/hosts`
  - Network Information System (NIS)

Mapping between Symbolic Name and IP Address

- Domain Name System: maps between hostnames and IP addresses
  - ⋆ *Nameserver* provides resolution service
    - ∗ typically BIND (Berkely Internet Name Domain)
    - ∗ `/etc/resolv.conf` - contains nameserver locations
- Other mechanisms:
  - ⋆ static hosts files, e.g. `/etc/hosts`
  - ⋆ Network Information System (NIS)
- Access DNS via library routines

  resolver library; `gethostbyname`,`gethostbyaddr`,etc.

jmayo: traceroute www.sydney.com.au

```
traceroute to www.sydney.com.au (209.66.116.64), 30 hops max, 38 byte packets
 1  brtr11.tc.mtu.edu (141.219.148.1)  0.791 ms  0.585 ms  0.587 ms
 2  bfs001-backbone.tc.mtu.edu (141.219.72.6)  2.321 ms  1.086 ms  1.101 ms
 3  fe1-0-0.mtu2.mich.net (198.110.131.61)  1.600 ms  1.790 ms  1.590 ms
 4  198.108.23.141 (198.108.23.141)  13.184 ms  11.848 ms  11.948 ms
 5  aads.above.net (206.220.243.71)  74.809 ms  75.470 ms  74.266 ms
 6  core1-chicago-2.ord.above.net (216.200.254.89)  76.393 ms  73.363 ms  74.317 ms
 7  sjc-ord-oc12.sjc2.above.net (207.126.96.118)  73.871 ms  72.713 ms  73.291 ms
 8  core5-core1-oc48.sjc.above.net (216.200.0.177)  73.828 ms  72.795 ms  72.903 ms
 9  main2-core5-oc3.sjc.above.net (216.200.0.206)  77.723 ms  75.180 ms  73.493 ms
10  sydney.com.au (209.66.116.64)  77.176 ms  75.010 ms  75.228 ms
```

jmayo: traceroute cs.wm.edu

```
traceroute to cs.wm.edu (128.239.2.31), 30 hops max, 38 byte packets
 1  brtr11.tc.mtu.edu (141.219.148.1)  14.237 ms  0.713 ms  0.765 ms
 2  bfs001-backbone.tc.mtu.edu (141.219.72.6)  1.955 ms  1.127 ms  0.920 ms
 3  fe1-0-0.mtu2.mich.net (198.110.131.61)  2.114 ms  1.755 ms  1.609 ms
 4  198.108.23.141 (198.108.23.141)  12.832 ms  12.052 ms  12.678 ms
 5  192.122.182.18 (192.122.182.18)  16.851 ms  16.688 ms  16.350 ms
 6  clev-ipls.abilene.ucaid.edu (198.32.8.26)  22.950 ms  23.675 ms  23.446 ms
 7  nycm-clev.abilene.ucaid.edu (198.32.8.30)  34.659 ms  35.171 ms  34.652 ms
 8  cisco7200-at-wm-to-noc-at-abilene.wm.edu (128.239.12.1)  46.280 ms  45.460 ms  46.232 ms
 9  128.239.14.6 (128.239.14.6)  46.616 ms  45.396 ms  45.868 ms
10  va.cs.wm.edu (128.239.2.31)  47.630 ms  46.582 ms  47.162 ms
```