Interprocess Communication Mechanisms

- shared storage
  - These mechanisms have already been covered. examples:
    * shared virtual memory
    * shared files
  - processes must agree on a name (e.g., a file name, or a shared virtual memory key) in order to establish communication
- message based
  - signals
  - sockets
  - pipes
  - . . .

Signals

- signals permit asynchronous one-way communication
  - from a process to another process, or to a group of processes
  - from the kernel to a process, or to a group of processes
- there are many types of signals
- the arrival of a signal causes the execution of a *signal handler* in the receiving process
- there may be a different handler for each type of signal

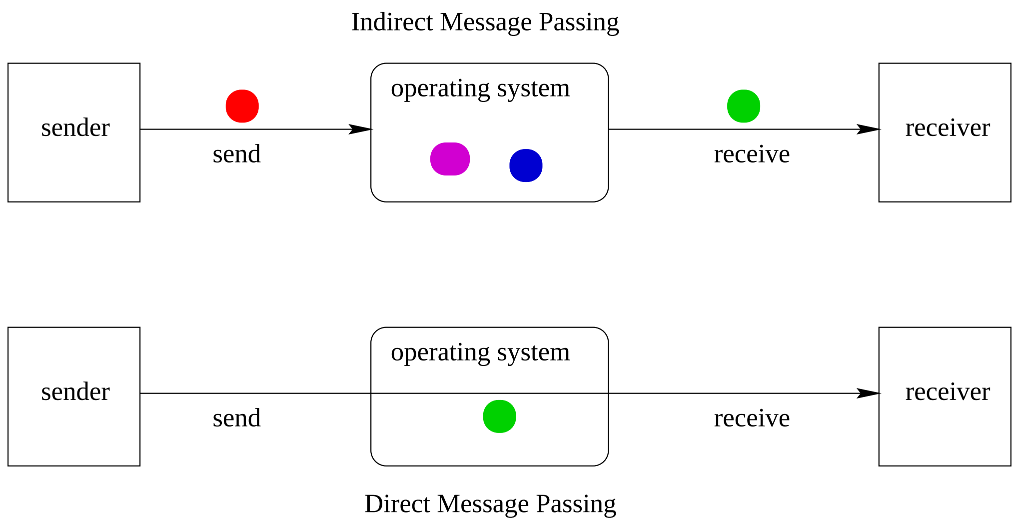
Examples of Signal Types

Signal	Value	Action	Comment
SIGINT	2	Term	Interrupt from keyboard
SIGILL	4	Core	Illegal Instruction
SIGKILL	9	Term	Kill signal
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGBUS	10,7,10	Core	Bus error
SIGXCPU	24,24,30	Core	CPU time limit exceeded
SIGSTOP	17,19,23	Stop	Stop process

Signal Handling

- operating system determined default signal handling for each new process
- example default actions:
  - ignore (do nothing)
  - kill (terminate the process)
  - stop (block the process)
- running processes can change the default for some or all types of signals
- signal-related system calls
  - calls to set non-default signal handlers, e.g., Unix `signal`, `sigaction`
  - calls to send signals, e.g., Unix `kill`

Message Passing



If message passing is indirect, the message passing system must have some capacity to buffer (store) messages.

Properties of Message Passing Mechanisms

Addressing: how to identify where a message should go

Directionality:

- simplex (one-way)
- duplex (two-way)
- half-duplex (two-way, but only one way at a time)

Message Boundaries:

datagram model: message boundaries

stream model: no boundaries

Properties of Message Passing Mechanisms (cont'd)

Connections: need to connect before communicating?

- in connection-oriented models, recipient is specified at time of connection, not by individual send operations. All messages sent over a connection have the same recipient.
- in connectionless models, recipient is specified as a parameter to each send operation.

Reliability:

- can messages get lost?
- can messages get reordered?
- can messages get damaged?

Sockets

- a socket is a communication *end-point*
- if two processes are to communicate, each process must create its own socket
- two common types of sockets

  stream sockets: support connection-oriented, reliable, duplex communication under the stream model (no message boundaries)

  datagram sockets: support connectionless, best-effort (unreliable), duplex communication under the datagram model (message boundaries)

- both types of sockets also support a variety of address domains, e.g.,

  Unix domain: useful for communication between processes running on the same machine

  INET domain: useful for communication between process running on different machines that can communicate using the TCP/IP protocols.

Using Datagram Sockets (Receiver)

```
s = socket(addressType, SOCK_DGRAM);
bind(s,address)
recvfrom(s,buf,bufLength,sourceAddress);
...
close(s);
```

- `socket` creates a socket
- `bind` assigns an address to the socket
- `recvfrom` receives a message from the socket
  - `buf` is a buffer to hold the incoming message
  - `sourceAddress` is a buffer to hold the address of the message sender
- both `buf` and `sourceAddress` are filled by the `recvfrom` call

Using Datagram Sockets (Sender)

```
s = socket(addressType, SOCK_DGRAM);
sendto(s,buf,msgLength,targetAddress)
...
close(s);
```

- `socket` creates a socket
- `sendto` send a message using the socket
  - `buf` is a buffer that contains the message to be sent
  - `msgLength` indicates the length of the message in the buffer
  - `targetAddress` is the address of the socket to which the message is to be delivered

More on Datagram Sockets

- `sendto` and `recvfrom` calls *may* block
  - `recvfrom` blocks if there are no messages to be received from the specified socket
  - `sendto` blocks if the system has no more room to buffer undelivered messages
- datagram socket communications are (in general) unreliable
  - messages (datagrams) may be lost
  - messages may be reordered
- The sending process must know the address of the receive process's socket. How does it know this?

A Socket Address Convention

```
Service         Port      Description
-------------------------------------------------------
echo            7/udp
systat          11/tcp
netstat         15/tcp
chargen         19/udp
ftp             21/tcp
ssh             22/tcp    # SSH Remote Login Protocol
telnet          23/tcp
smtp            25/tcp
time            37/udp
gopher          70/tcp    # Internet Gopher
finger          79/tcp
www             80/tcp    # WorldWideWeb HTTP
pop2            109/tcp   # POP version 2
imap2           143/tcp   # IMAP
```

Using Stream Sockets (Passive Process)

```
s = socket(addressType, SOCK_STREAM);
bind(s,address);
listen(s,backlog);
ns = accept(s,sourceAddress);
recv(ns,buf,bufLength);
send(ns,buf,bufLength);
...
close(ns); // close accepted connection
close(s);  // don't accept more connections
```

- `listen` specifies the number of connection requests for this socket that will be queued by the kernel
- `accept` accepts a connection request and creates a new socket (`ns`)
- `recv` receives up to `bufLength` bytes of data from the connection
- `send` sends `bufLength` bytes of data over the connection.

Notes on Using Stream Sockets (Passive Process)

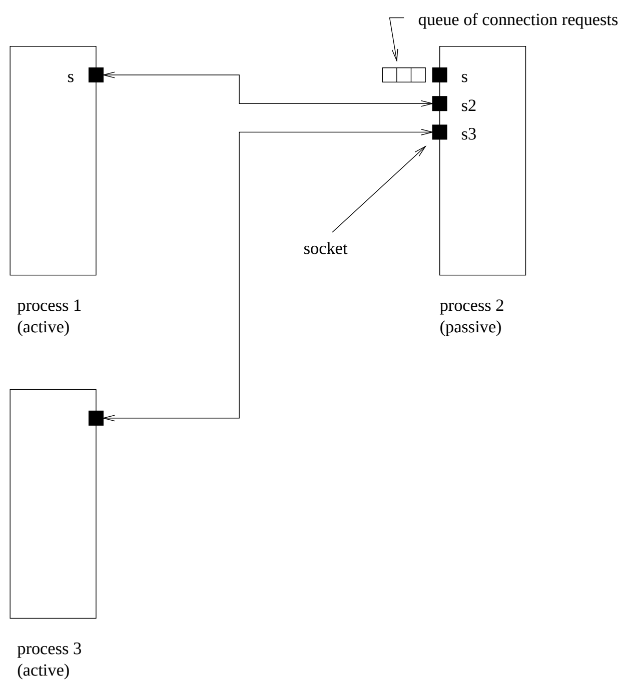
- `accept` creates a new socket (`ns`) for the new connection
- `sourceAddress` is an address buffer. `accept` fills it with the address of the socket that has made the connection request
- additional connection requests can be accepted using more `accept` calls on the original socket (`s`)
- `accept` blocks if there are no pending connection requests
- connection is duplex (both `send` and `recv` can be used)

Using Stream Sockets (Active Process)

```
s = socket(addressType, SOCK_STREAM);
connect(s,targetAddress);
send(s,buf,bufLength);
recv(s,buf,bufLength);
...
close(s);
```

- `connect` requests a connection request to the socket with the specified address
  - `connect` blocks until the connection request has been accepted
- active process may (optionally) bind an address to the socket (using `bind`) before connecting. This is the address that will be returned by the `accept` call in the passive process
- if the active process does not choose an address, the system will choose one

Illustration of Stream Socket Connections



Socket Example: Client

```
#include "defs.h"

#define USAGE "client serverhost port#\n"
#define ERROR_STR_LEN  (80)

int
main(int argc, char *argv[])
{
  struct hostent *hostp;
  int socketfd, server_port, num;

  char error_str[ERROR_STR_LEN];
  char read_buf[BUF_LEN];
  char *hostname;
  struct sockaddr_in server_addr;
  struct in_addr tmp_addr;
```


Socket Example: Client (continued)

```
  if (argc != 3) {
    fprintf(stderr, "%s", USAGE);
    exit(-1);
  }

  /* get hostname and port for the server */
  hostname = argv[1];
  server_port = atoi(argv[2]);

  /* get the server hosts address */
  if ((hostp = (struct hostent *)
        gethostbyname(hostname)) ==
        (struct hostent *) NULL) {
    sprintf(error_str,
        "client: gethostbyname fails for host  %s",
         hostname);
    perror(error_str);
    exit(-1);
  }
```

Socket Example: Client (continued)

```
/* create a socket to connect to server */
if ((socketfd = socket(DOMAIN, SOCK_STREAM, 0)) < 0) {
  perror("client: can't create socket ");
  exit(1);
}

/* zero the socket address structure */
memset((char *) &server_addr, 0, sizeof(server_addr));

/* start constructing the server socket addr */
memcpy(&tmp_addr, hostp->h_addr_list[0],
       hostp->h_length);

printf("Using server IP addr = %s\n",
        inet_ntoa(tmp_addr));
```


Socket Example: Client (continued)

```
/* set servers address field, port number and family */
memcpy((char *) &server_addr.sin_addr,
       (char *) &tmp_addr,
       (unsigned int) hostp->h_length);
server_addr.sin_port = htons(server_port);
server_addr.sin_family = DOMAIN;

/* connect to the server */
if (connect(socketfd, (struct sockaddr  *) &server_addr,
  sizeof(server_addr)) < 0) {
  perror("client: can't connect socket ");
  exit(1);
}
```

Socket Example: Client (continued)

```
  /* send from the client to the server */
  num = write(socketfd, CLIENT_STR, CLIENT_BYTES);
  if (num != CLIENT_BYTES) {
    perror("client: write to socket failed\n");
    exit(1);
  }

  /* receive data sent back by the server */
  num = read(socketfd, &read_buf, SERVER_BYTES);
  if (num != SERVER_BYTES) {
    perror("client: read from socket failed\n");
    exit(1);
  }

  printf("sent %s\n", CLIENT_STR);
  printf("received %s\n", read_buf);

  close(socketfd);
  exit(0);
}
```


Socket Example: Server

```
#include "defs.h"

int
main()
{
  int serverfd, clientfd;
  struct sockaddr_in server_addr, client_addr;
  int size, num;
  char read_buf[BUF_LEN];
  struct sockaddr_in bound_addr;

  serverfd = socket(DOMAIN, SOCK_STREAM, 0);

  if (serverfd < 0) {
    perror("server: unable to create socket ");
    exit(1);
  }
```

Socket Example: Server (continued)

```
/* zero the server_addr structure */
memset((char *) &server_addr, 0, sizeof (server_addr));

/* set up addresses server will accept connections on */
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(PORT);
server_addr.sin_family = DOMAIN;

/* assign address to the socket */
if (bind (serverfd, (struct sockaddr *) &server_addr,
  sizeof(server_addr)) < 0) {
  perror("server: unable to bind socket ");
  exit(1);
}
```


Socket Example: Server (continued)

```
/* Willing to accept connections on this socket. */
/* Maximum backlog of 5 clients can be queued */
listen(serverfd, 5);

for (;;) {
  /* wait for and return next completed connection */
  size = sizeof(client_addr);
  clientfd = accept(serverfd,
      (struct sockaddr *) &client_addr, &size);

  if (clientfd < 0) {
    perror("server: accept failed ");
    exit(1);
  }

  /* get the data sent by the client */
  num = read(clientfd, read_buf, CLIENT_BYTES);
  if (num != CLIENT_BYTES) {
    perror("server: read from client socket failed ");
    exit(1);
  }
```

Socket Example: Server (continued)

```
    /* process the client info / request here */
    printf("client sent %s\n", read_buf);
    printf("server sending %s\n", SERVER_STR);

    /* send the data back to the client */
    num = write(clientfd, SERVER_STR, SERVER_BYTES);
    if (num != SERVER_BYTES) {
      perror("server: write to client socket failed ");
      exit(1);
    }

    close(clientfd);
  }
  exit(0);
}
```


Pipes

- pipes are communications objects (not end-points)
- pipes use the stream model and are connection-oriented and reliable
- some pipes are simplex, some are duplex
- pipes use an implicit addressing mechanism that limits their use to communication between *related* processes, typically a child process and its parent
- a `pipe()` system call creates a pipe and returns two descriptors, one for each end of the pipe
  - for a simplex pipe, one descriptor is for reading, the other is for writing
  - for a duplex pipe, both descriptors can be used for reading and writing

One-way Child/Parent Communication Using a Simplex Pipe

```
int fd[2];
char m[] = "message for parent";
char y[100];
pipe(fd); // create pipe
pid = fork(); // create child process
if (pid == 0) {
  // child executes this
  close(fd[0]); // close read end of pipe
  write(fd[1],m,19);
  ...
} else {
  // parent executes this
  close(fd[1]); // close write end of pipe
  read(fd[0],y,100);
  ...
}
```


Illustration of Example (after `pipe()`)

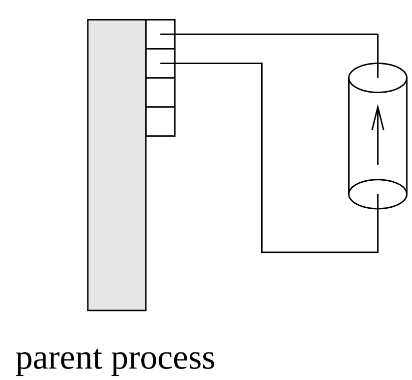


Illustration of Example (after `fork()`)

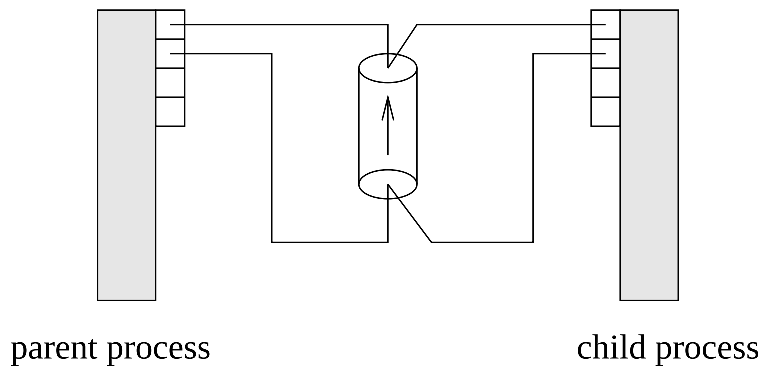
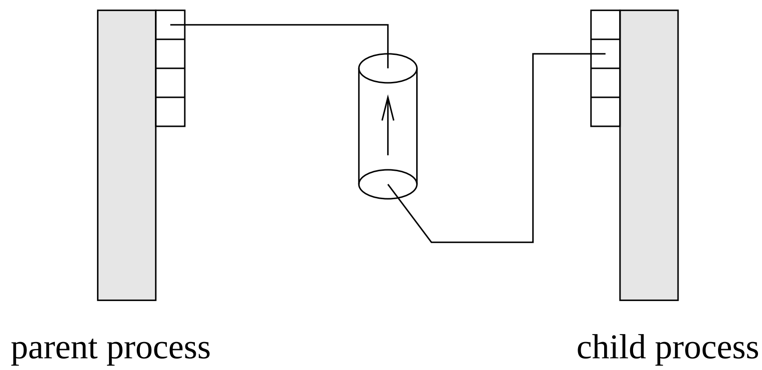


Illustration of Example (after `close()`)



Examples of Other Interprocess Communication Mechanisms

named pipe:

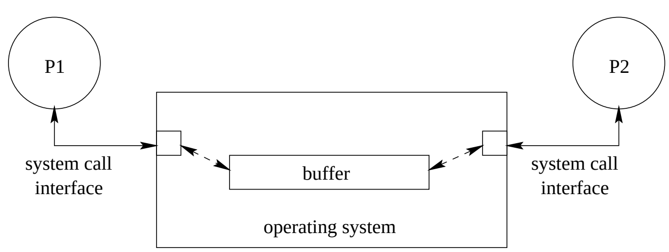
- similar to pipes, but with an associated name (usually a file name)
- name allows arbitrary processes to communicate by opening the same named pipe
- must be explicitly deleted, unlike an unnamed pipe

message queue:

- like a named pipe, except that there are message boundaries
- `msgsend` call sends a message into the queue, `msgrecv` call receives the next message from the queue

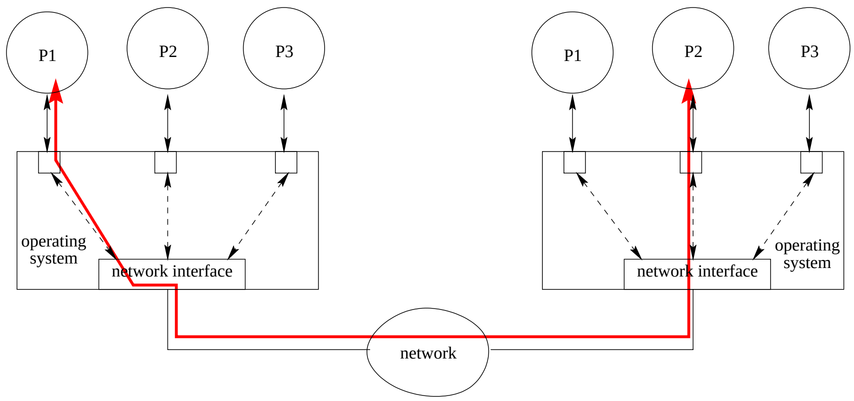
Implementing IPC

- application processes use descriptors (identifiers) provided by the kernel to refer to specific sockets and pipe, as well as files and other objects
- kernel *descriptor tables* (or other similar mechanism) are used to associate descriptors with kernel data structures that implement IPC objects
- kernel provides bounded buffer space for data that has been sent using an IPC mechanism, but that has not yet been received
  - for IPC objects, like pipes, buffering is usually on a per object basis
  - IPC end points, like sockets, buffering is associated with each endpoint



Network Interprocess Communication

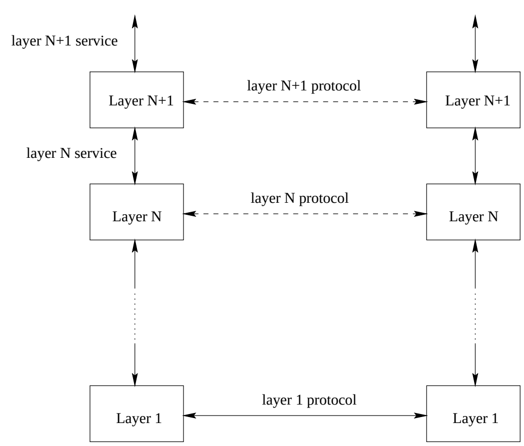
- some sockets can be used to connect processes that are running on different machine
- the kernel:
  - controls access to network interfaces
  - multiplexes socket connections across the network



Networking Reference Models

- ISO/OSI Reference Model

7	Application Layer
6	Presentation Layer
5	Session Layer
4	Transport Layer
3	Network Layer
2	Data Link Layer
1	Physical Layer



- Internet Model
  - layers 1-4 and 7

Internet Protocol (IP): Layer 3

- every machine has one (or more) IP address, in addition to its data link layer address(es)
- In IPv4, addresses are 32 bits, and are commonly written using "dot" notation, e.g.:
  - cpu06.student.cs = 129.97.152.106
  - www.google.ca = 216.239.37.99 or 216.239.51.104 or ...
- IP moves packets (datagrams) from one machine to another machine
- principal function of IP is *routing*: determining the network path that a packet should take to reach its destination
- IP packet delivery is "best effort" (unreliable)

IP Routing Table Example

- Routing table for zonker.uwaterloo.ca, which is on three networks, and has IP addresses 129.97.74.66, 172.16.162.1, and 192.168.148.1 (one per network):

Destination	Gateway	Interface
172.16.162.*	-	vmnet1
129.97.74.*	-	eth0
192.168.148.*	-	vmnet8
default	129.97.74.1	eth0

- routing table key:

  destination: ultimate destination of packet

  gateway: next hop towards destination (or "-" if destination is directly reachable)

  interface: which network interface to use to send this packet

Internet Transport Protocols

TCP: transport control protocol

- connection-oriented
- reliable
- stream
- congestion control
- used to implement INET domain stream sockets

UDP: user datagram protocol

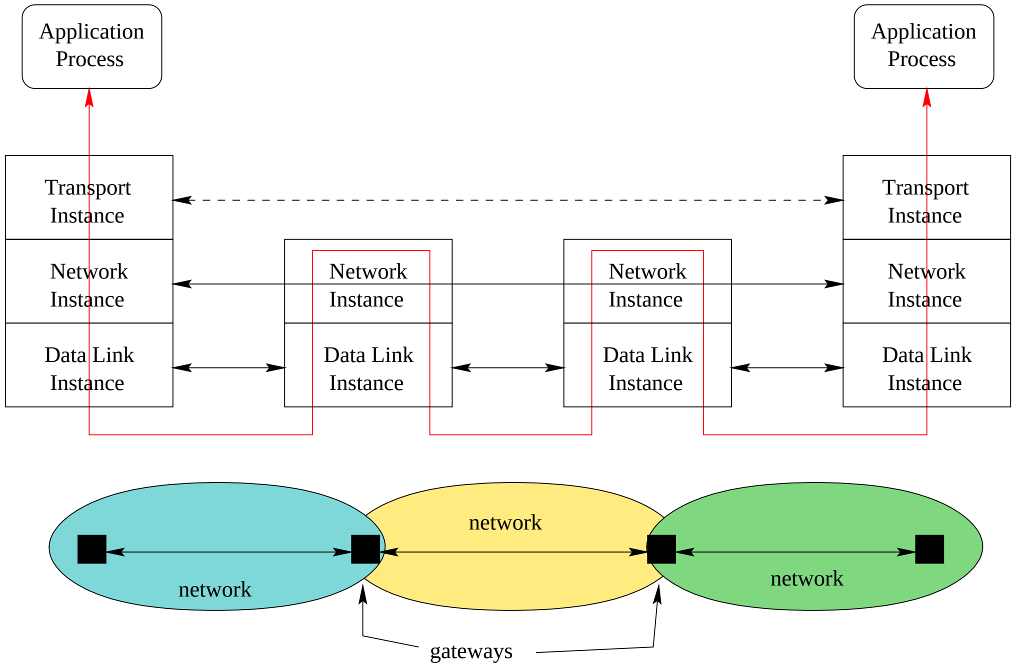
- connectionless
- unreliable
- datagram
- no congestion control
- used to implement INET domain datagram sockets

TCP and UDP Ports

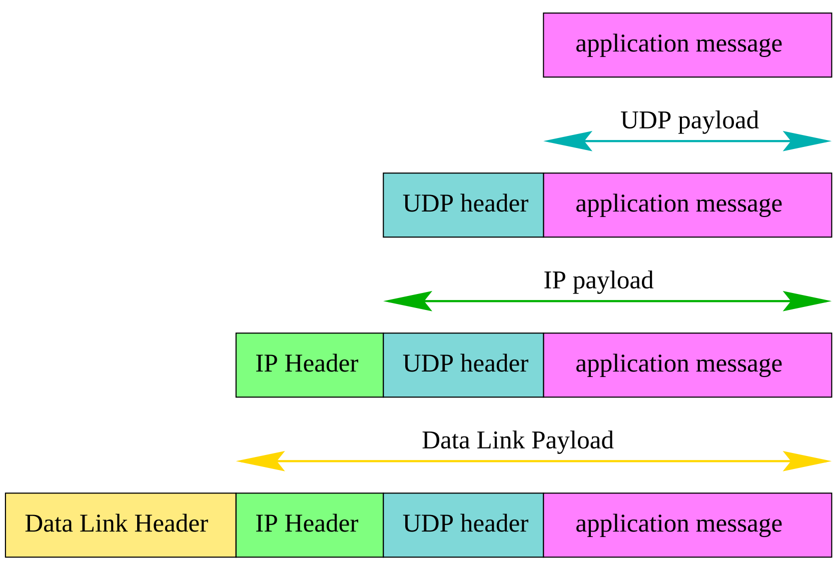
- since there can be many TCP or UDP communications end points (sockets) on a single machine, there must be a way to distinguish among them
- each TCP or UDP address can be thought of as having two parts:

  (machine name, port number)
- The machine name is the IP address of a machine, and the port number serves to distinguish among the end points on that machine.
- INET domain socket addresses are TCP or UDP addresses (depending on whether the socket is a stream socket or a datagram socket).

Example of Network Layers



Network Packets (UDP Example)



BSD Unix Networking Layers