Interprocess Communication (IPC)

A process is of little use if it is unable to communicate with other processes (and users). There are many forms of IPC in UNIX. We have already seen how pipes (and "named pipes") can be used to connect processes to form a pipeline. UNIX also provides the **System V IPC** mechanisms of shared memory, message queues and IPC semaphores. These are all somewhat dated and cumbersome, although they are still sometimes used in new code. A newer message queues implementation was standardized by POSIX and is available on recent UNIX versions; it is even less popular than SystemV IPC.

All of the above IPC methods are limited to processes executing on the same machine. Under UNIX, the **socket layer** provides access to network protocols which can connect processes on different machines. Through socket programming, the same API can be used regardless of whether the target process is local or remote.

The socket layer can be used with a wide variety of networking protocols. In this unit, we will consider the application of sockets to the internet. It is important to note that the sockets API can be used to access a number of networking protocols, and that the internet protocols described in this unit are not defined by the sockets API, and could be accessed by other interfaces. However, this particular combination of sockets and the "TCP/IP" internet protocol suite is very popular and basis for many applications and services.

The material in this unit is intended as an introduction to a complex topic that is at the fringes of Operating Systems vs Computer Networks. The reader should consult other authorities, such as Steven's *Advanced Programming in the UNIX Environment* for a full presentation on TCP/IP and sockets programming in UNIX.

What is the Internet

The roots of the **Internet** trace back to about 1969, although the protocols that form the basis of today's internet did not emerge until the early 1980s.

At its simplest, the internet is a loose collection of private networks, sharing some common protocols and policy conventions. This allows computers and other devices connected to these networks to exchange data with each other and to participate in distributed applications. This same technology can be used both on the global internet and within private networks in so-called "intra-net" applications.

The basic service provided by the Internet is known as Internet Protocol, or IP. IP service is roughly analogous to the Postal Service. Messages, known as **IP packets** are addressed to their destination by means of a unique 32 bit integer, known as the **IP address**. Each packet bears both the destination and the source ("return") address. (*Note: the 32-bit address applies to IPV4. The IPV6 protocol uses a more cumbersome*

128-bit address which will not be covered in these notes)

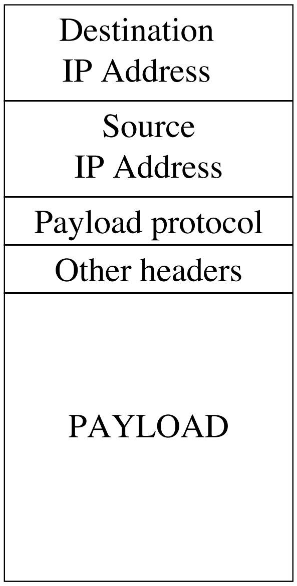
The IP address identifies hosts within the Internet. As with buildings having multiple street entrances and thus multiple Postal addresses, it is possible for one host to have multiple IP addresses, for example, if it is connected to multiple networks.

IP addresses are represented in **dotted decimal notation**, consisting of 4 decimal numbers representing the four bytes of the 32 bit address, delimited by periods, e.g., `192.168.4.5` All numeric values in the Internet protocols are in **network byte order**, which is defined to be big-endian.

The allocation of IP addresses and their manipulation is beyond the scope of this course. Consider the IP address to be just a unique identifier. There is, however, one address which is reserved to be an alias for the local host: `127.0.0.1`.

Each IP packet is self-contained and is stateless, i.e. there is no dependency on other IP packets sent earlier or later. The destination address contains all of the information necessary to deliver the message. That task is given to dedicated devices known as **routers**.

An IP Packet



Message boundaries are preserved. IP does not, as a rule, deliver a partial packet. Either the entire packet is delivered, or it is lost. A message is never split up and delivered as a series of smaller messages (although this splitting may be performed internally, the split messages are never delivered as such), nor are multiple messages ever coalesced and delivered as one.

Like the Postal Service, IP provides "best effort, unreliable" delivery:
- Packets may be lost in transit, in fact, that is a common occurrence.

- The latency incurred in delivering a packet is unpredictable and unguaranteed.
- Packets may be delivered out of order.
- Packets may be delivered in damaged condition (the message has been corrupted), however this is not that common an occurence since most link-layers guard against line errors.

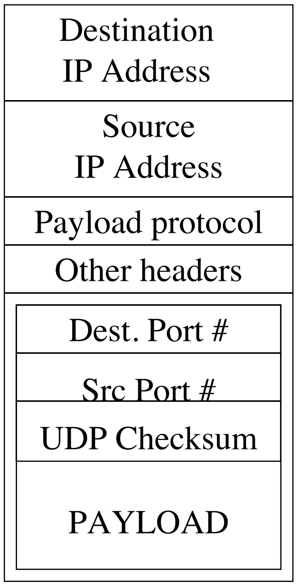
UDP and port numbers

IP addresses refer to specific hosts on the Internet. Since there are typically multiple processes utilizing services on the same machine, **port numbers** are used to disambiguate and create an array of, as it were, mailboxes within a single machine in which processes can receive messages.

The **User Datagram Protocol** is a simple wrapper around IP. It provides a source and destination port number (16 bit integer) which, combined with the source and destination IP address found in the IP packet, identifies both ends of the message transmission.

UDP provides checksumming for error detection, but many implementations ignore it, so it should not be assumed that UDP detects errors. UDP does not provide any other "value-added" services above IP.

A UDP Message
carried within IP packet



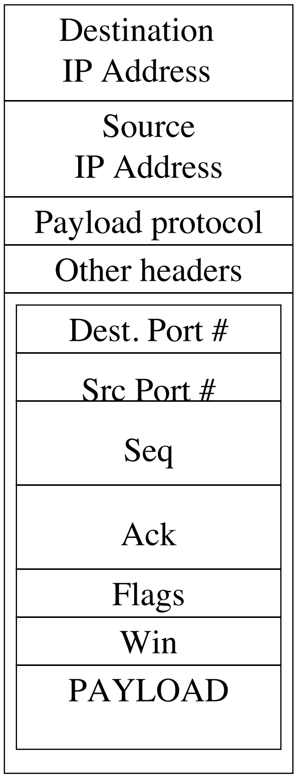
When an IP message arrives at a host, the protocol field is checked. A UDP message appears within the payload field of an IP packet, with a protocol of UDP. The IP payload is extracted from the IP packet and handed up to the UDP module which, in turn, examines the destination port number and finds a process that is waiting for messages on that port number. The payload portion of the UDP message is extracted and becomes the message read by the user-level program.

TCP and reliable transport

The **Transmission Control Protocol (TCP)** utilizes IP to transport its payload, and provides a service akin to a pair of UNIX pipes between hostA:portA and hostB:portB.
• TCP guarantees that data will never be lost (without notification). Since it provides a stream service, it is also vital that it guarantee that data will not be delivered out of order.

TCP is a connection-oriented service which utilizes 16 bit port numbers, just as UDP does, to identify individual "mailboxes" on hosts. The UDP and TCP port number spaces, however, are independent.
• Before data can be sent over TCP, a connection has to be made between hostA:portA, which requests the connection, and hostB:portB, which is ready to accept the connection. Either side can shut down the connection in either the read, write or both directions, just as is the case with a UNIX pipe, resulting in an "end-of-file" or "broken-pipe" indication on the remote end, as appropriate.
• Unlike the message-oriented service provided by UDP, TCP is a byte stream service. There is no concept of a distinguishable "message", again, just like a UNIX pipe.
• TCP provides flow control which limits the total amount of data that can be queued against a receiver, just as is the case with a UNIX pipe. This prevents a fast writer from overrunning a slow reader.

A TCP Segment
carried within IP packet



• TCP provides reliability by buffering written data locally until an acknowledgement message arrives from the remote end. TCP bundles pending data into a **tcp segment** which is carried within an IP packet. After sending data, TCP sets a timer which is reset when the data are acknowledged. If the timer goes off, the data are presumed lost in transit and are re-sent.

• In each message, the SEQ and ACK fields are modulo-2ˆ32 numbers which provide a way of identifying each byte in each of the directional streams, while WIN indicates the maximum amount of additional data which the host is willing to accept on the receive side. If the stream carries more than 2ˆ32 bytes in its lifetime, the sequence numbers simply roll over.

• The Sequence Number corresponding to the first byte sent over the connection after connection establishment is known as the Initial Sequence Number. It is chosen randomly for reasons of performance and security. The sequence numbers in the transmit and receive direction are completely independent of each other. The ACK field contains the last sequence number received from the other direction, plus one (or in other words, the next expected sequence number).

• When the last byte of data has been sent over the connection in a particular direction, the TCP protocol assigns a sequence number to represent the end-of-data mark. This is known as a **FIN**. The fact that FIN has a sequence number means that not only are the data themselves reliably retransmitted, but also the end-of-data mark, so that when that FIN has been acknowledged, the sender has assurance that all of the data sent have been received, and that the receiver also knows that no more data are coming.

• Further details of the functioning of the TCP protocol are beyond the scope of this course.

The role of the operating system

On a multitasking operating system, the kernel is responsible for running the protocol code necessary to implement IP, TCP and UDP (at the least), and provides a facility whereby individual processes can utilize network services and associate themselves with particular ports.

Several standard APIs have existed for accessing network services. The most popular one has been the "sockets" model introduced with the 4BSD flavor of UNIX in the early 1980s. Many non-UNIX operating systems use a variant of this model, e.g. the "winsock" API under the MS Windows product family.

The first step in sending or receiving data across the network is to obtain a **socket**. A socket is just a file descriptor, created by the `socket` system call.

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

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

domain specifies an address family to be used for this socket. For our applications, we will use `AF_INET`, which specifies TCP/IP, using IP version 4 addressing. sockets is a generic API which can support a wide range of protocols and address families. Another common address family is AF_UNIX, or **UNIX-Domain sockets**, in which the socket addresses appear to be non-persistent names in the filesystem. UNIX-domain sockets are restricted to intra-host communications, however, and will not be covered further. `AF_INET6` sockets use IP version 6 addressing, which is an extension of IP version 4. Most applications in the internet today are still based on version 4 addressing.

type specifies the type of protocol desired. In normal sockets usage, *protocol* is left as 0 and the operating system chooses the protocol based on *domain* and *type*. Within the AF_INET domain, only 2 types are supported:

> `SOCK_DGRAM` implies the use of UDP.
>
> `SOCK_STREAM` implies the use of TCP.

`socket` returns the file descriptor of the new socket, or -1 on error.

Under the sockets API, an address is specified with a `struct sockaddr`:

```
struct sockaddr {
        u_short sa_family;              /* address family */
        char    sa_data[14];            /* bare minimum address buffer */
};
```

In practice, a struct sockaddr is never used. Rather, the address structure for the particular family (e.g. `struct sockaddr_in)` is used. Since each such structure has the address family as the first 2 bytes, this works and the kernel is able to decode the address further.

Under TCP/IP, an address consists of a **network address**, which identifies a particular host, and a **port number** which resolves individual sockets on the same host. This is embodied in `struct sockaddr_in`:

```
#include <netinet/in.h>                /* Defines sockaddr_in */
#include <arpa/inet.h>                 /* Historical, include it! */

struct in_addr {
         unsigned long  s_addr;                          /* Network byte order */
};

struct sockaddr_in {
         short sin_family;             /* Always AF_INET */
         unsigned short sin_port;      /* Network byte order */
         struct in_addr sin_addr;      /* A struct, historical! */
         char sin_zero[8];             /* Useless padding */
};
```

Note the seemingly pointless use of a struct to hold the network address. See <netinet/in.h> for the historical reasons behind this. Also note that s_addr and sin_port are in network byte order (big endian), not native (host) byte order! The sockets interface is generic, and although we, as trained students of TCP/IP, know that IP addresses and port numbers can be interpreted as integers, from the standpoint of the sockets API, they are "opaque data" which must be passed through to the appropriate protocol driver in the kernel without interpretation or modification.

Thus to read or write these network byte order fields in a portable fashion requires the use of the byteorder macros `htons, htonl, ntohs` and `ntohl` which are either no-ops, or byte swappers, depending on the native byte order of the machine on which the program is running.

All sockets must be bound to a source address before data are transmitted from them. To explicitly bind a specific port number, use the `bind` system call:

```
int bind(int s,struct sockaddr *name,int namelen)
```

0 is returned on success, -1 on error. Possible errors include:

> `EACCES`: attempt to access bind a privileged address and caller is not root (Note: UNIX TCP/IP, privileged addresses are port numbers <1024. Traditionally, the capability of binding a privileged port is reserved to the superuser (uid==0),

however in modern kernels, finer-grained control is possible and non-root processes can bind these ports).

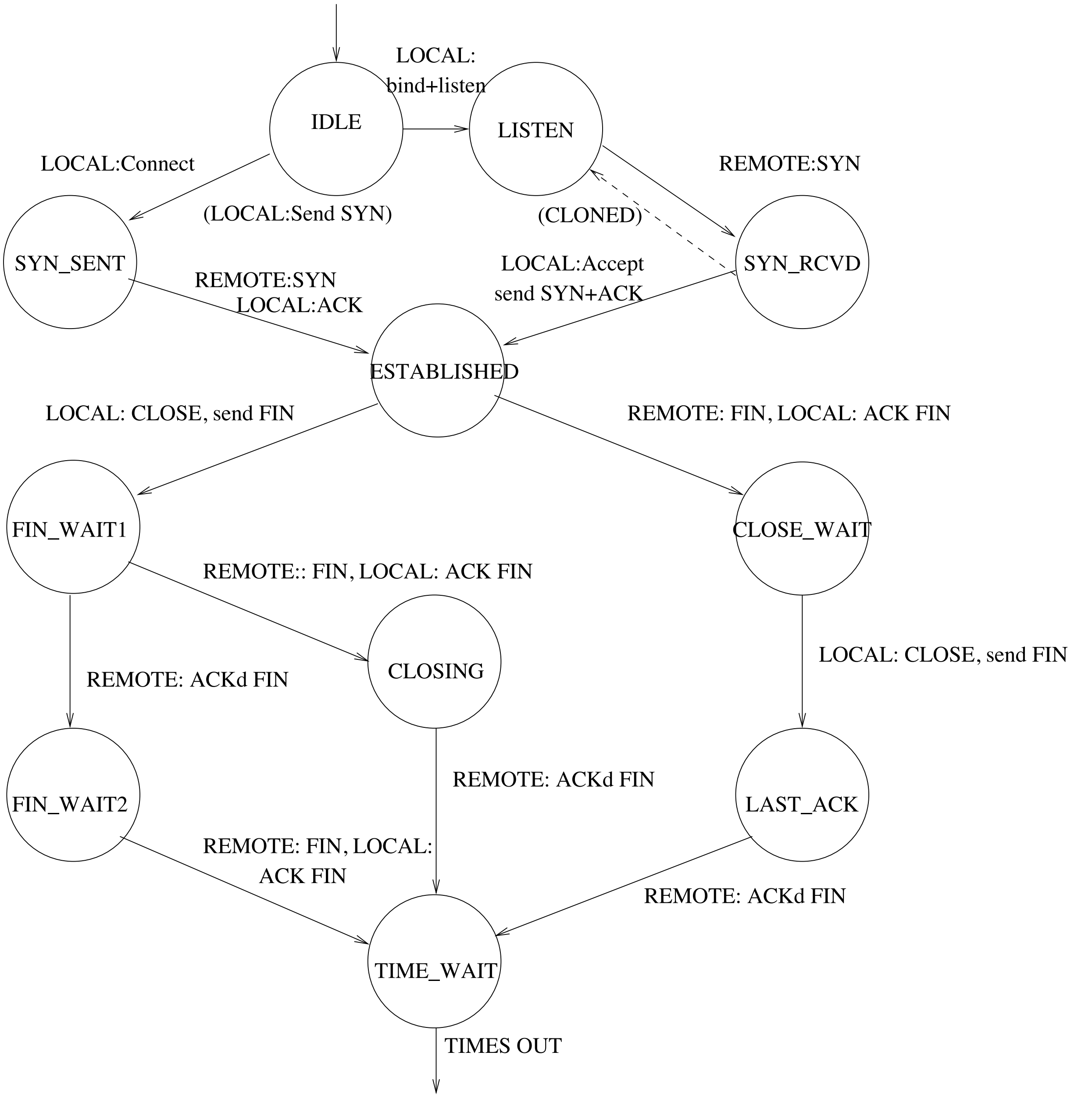
`EADDRINUSE`: The requested address is already bound to another socket somewhere on the machine.

The use of `bind` is often skipped because the sockets API specifies an automatic binding to the next available port number under certain operations (connect, send). Specifying a *sin_port* of 0 and *s_addr* of INADDR_ANY when calling `bind` also has this effect.

Note that `bind` takes a `struct sockaddr *`, which is generic. However, in sockets programming, one would typically be working with a specific structure, e.g. `struct sockaddr_in`. Therefore a type cast is needed to avoid compiler warnings.

At this point, semantics diverge between SOCK_DGRAM and SOCK_STREAM. For SOCK_STREAM in AF_INET (TCP), a connection must be established before data can be transferred. SOCK_DGRAM (UDP) is a stateless, connectionless protocol and does not require connection before data transfer.

The state diagram below is derived from the TCP protocol, with extra information along the edges to describe actions taken by either the local or the remote process. The TCP state of all sockets on the system can be queried with the `netstat` command.



There are two sides to opening a SOCK_STREAM(TCP) connection. First, the PASSIVE opener binds a specific port number and issues a *listen*, enabling that port to receive connection requests. Then ACTIVE opener issues a *connect* specifying a remote address and port number.

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

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

This causes the TCP protocol to query the remote system (*sin_addr*) to see if anyone is listening on the specified *sin_port*. If not, `connect` comes back with ECONNREFUSED. Otherwise, the connection proceeds. `connect` blocks until the connection is either established, refused or times out.

`connect` can be called for SOCK_DGRAM sockets as a convenience function. It causes the kernel to remember the destination address, but does not result in any actual protocol action.

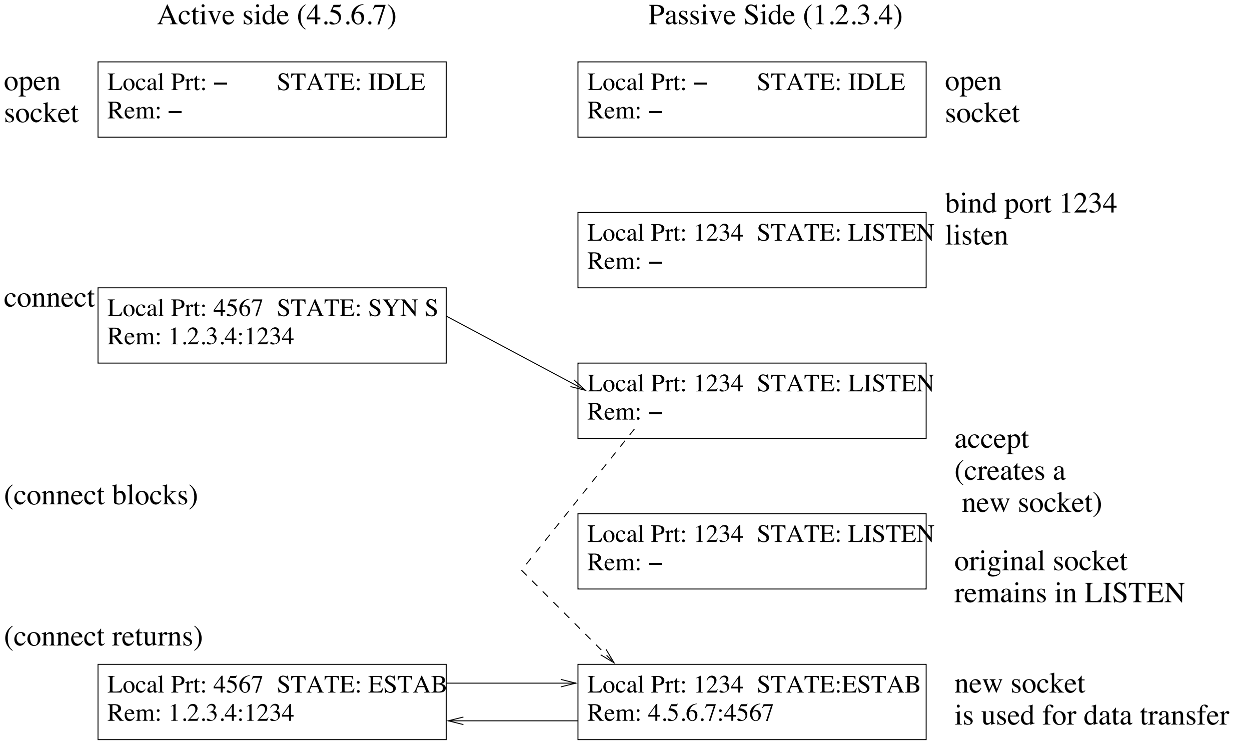
On the PASSIVE side, the listening process uses `accept` to receive and accept the incoming connection.

```
int accept(int listener_fd,struct sockaddr *addr,int *addrlen)
```

addrlen is a **value-result** parameter. It must be set before calling `accept` to the maximum allowable size to return in *addr*. When `accept` returns, it contains the actual size used. For TCP/IP, the address is fixed-length, so this is a superfluous feature.

`accept` blocks until a connection arrives. `accept` creates a **new socket** to reference the incoming connection, and returns its file descriptor. The remote address is returned in *addr*. The original socket, *listener_fd*, continues to exist in the LISTEN state, where more connection requests can be accepted. The returned socket advances to the ESTABLISHED state, where data can be transferred between the hosts. If multiple connection requests arrive rapidly, the cloned sockets reside in the SYN_RCVD state until, one by one, they are accept'd by the process. The operating system may impose a limit on these "backlogged" nascent connections. This limit can be set (within a range) by the second parameter in the `listen` system call.

Establishing a TCP connection



Determining at what address to listen is no small task. Certain services are generally assigned to specific port numbers, which are controlled by internet standards bodies. These are the so-called "well-known ports," such as the SMTP mail protocol on port 25, and the SSH remote login service on port 22.

This raises the issue of uniqueness. In TCP, a connection is identified by the pair {local port, local address}/{remote port, remote address}. This allows the local port number to be overloaded for servers. For example, at any time there may be multiple sockets open with a local port number of 23 (TELNET), but each is connected to a different remote address/port.

Let’s look at an example of binding, listening and accepting a connection on the PASSIVE side:

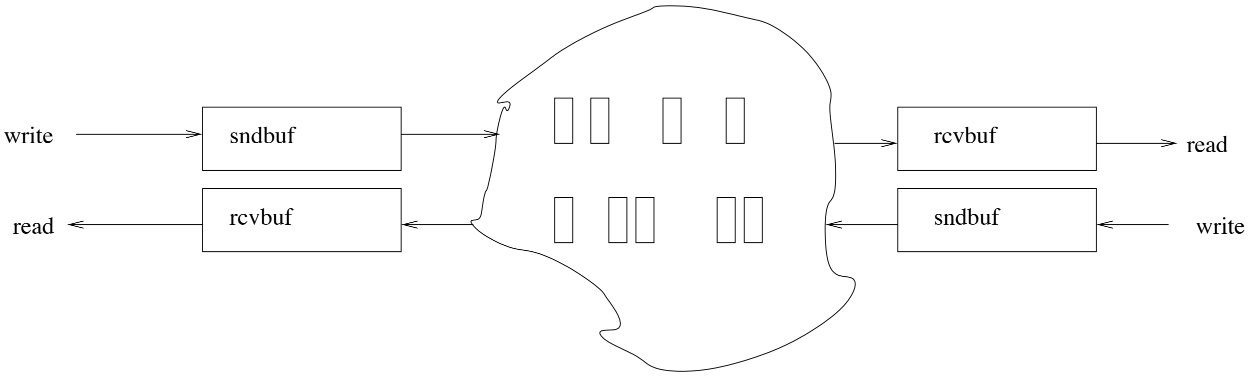
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

f()
{
struct sockaddr_in sin;
int s,s2,len;
          if ((s=socket(AF_INET,SOCK_STREAM,0))<0)
          {
                    perror("socket");
                    return -1;
          }
          sin.sin_family=AF_INET;
          sin.sin_port=htons(25);                /* The mail port */
          sin.sin_addr.s_addr=INADDR_ANY;
          if (bind(s,(struct sockaddr *)&sin,sizeof sin)<0)
          {
                    perror("bind");
                    close(s);
                    return -1;
          }
          if (listen(s,128)<0)
          {
                    perror("listen");
                              return -1;
          }
          len=sizeof sin;
          if ((s2=accept(s,(struct sockaddr *)&sin,&len))<0)
          {
                    perror("accept");
                    return -1;
          }
          return s2;
}
```

Sockets and read/write

Sockets are a form of file, and through the abstraction layer of the virtual file system code within the kernel, many of the same system calls which work for regular files on the disk also work for sockets. Under the Linux kernel, an in-core inode exists for each open socket. It belongs to file system type `sockfs` (in much the same way that pipes belong to `pipefs`). The inode type is `S_IFSOCK`.

We can think of a TCP connection as a pair of pipes connecting the two endpoints. In particular, once a connection has reached the TCP ESTABLISHED state, the two directions are independent of each other.



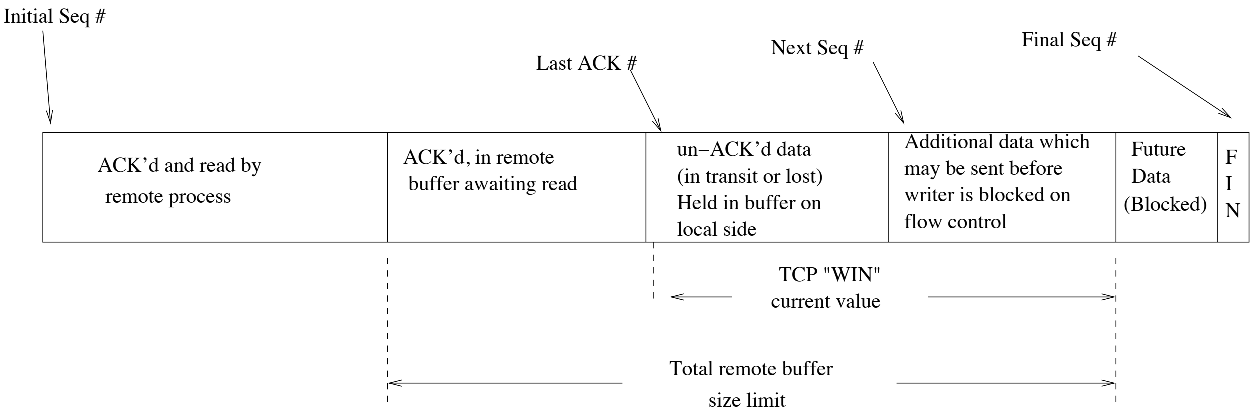
However unlike pipes which are contained in a single host, data involved in a TCP connection can be in one of three places:

- Buffered by the kernel on the sending side
- In transit over the network
- Buffered by the kernel on the receiving side

The reliability of TCP means that on the sending side, copies of the data must be retained in the send buffer until those bytes have been received at the far end (as proved by the receipt of ACK messages). Likewise at the receiving end, the kernel must buffer the data until the user-level program reads it.

Data transfer over a TCP connection can be accomplished with the normal read and write systems calls. Since TCP does not preserve message boundaries, a read will simply return all currently queued data sitting in the receive buffer, ignoring any the boundaries with which the data were originally sent. Reading from a TCP connection is akin to reading from a pipe. The read system call will block until the far end writes some data, and then will return with whatever data are available. When the remote side shuts down the connection in that direction, read returns an EOF (after all queued data have been read).

We can look at the lifetime of each half of a TCP connection as being a stream of data identified by sequence numbers. We can further categorize parts of that stream as depicted below:


Initial Seq #
Last ACK #
Next Seq #
Final Seq #
ACK'd and read by remote process
ACK'd, in remote buffer awaiting read
un−ACK'd data (in transit or lost) Held in buffer on local side
Additional data which may be sent before writer is blocked on flow control
Future Data (Blocked)
F I N
TCP "WIN" current value
Total remote buffer size limit

The send buffer and receive buffer are of finite size, typically less than 64K. Likewise, TCP messages that are in transit are taking up buffer resources in the network, so the protocol desires to limit the total amount of data in transit (again typically under 64K). When the sender is too fast, either because the receiver is slow, or the network is slow, or both, flow control engages to keep things in balance.

The remote side will advertise a WINdow with each ACK, relative to the highest sequence number it has ACK'd, such that its receive buffer will not be overflowed. On the local side, when a process attempts to write data which would exceed the window, it blocks until the remote side moves the window. If network throughput is slow, the data which are "in transit" will grow until the they fill the window, also engaging flow control.

It is possible for "short writes" to occur when writing to a TCP socket if the number of bytes which the process is attempting to write exceeds the window size. The kernel may write whatever bytes fit, and return a short write count, or it may block the process, depending on the kernel version.

Sending and receiving UDP messages

For UDP, *sendto* and *recvfrom* are used for data transfer. Since UDP is a connectionless protocol, the destination address and port number must be specified with each transmitted message:

```
int sendto(int sockfd,
          char *msg,
          int msglen,
          int flags,
          struct sockaddr *to,
          int tolen)
```

flags is normally left 0. If a UDP socket has been *connect*ed, the destination address for all messages sent on that socket is set. *send* can then be used, ommitting the address:

```
send(sockfd,msg,msglen,flags)
```

To receive a message:

```
int recvfrom(int sockfd,
          char *buf,
          int buflen,
          int flags,
          struct sockaddr *from,
          int *fromlen)
```

`recvfrom` will block until a message is available. The next message is placed into *buf*. If the message overflows *buflen*, excess bytes are discarded for UDP sockets. *fromlen* is value-result parameter which is set to the length of *from*, and is filled in with the actual length on return, just like `accept`. The address placed in *from* is the remote address associated with the recieved message.

If a UDP socket is `connect`ed, then only messages from the connected remote address/port# will be received, and `recv` can be used instead:

```
recv(sockfd,buf,buflen,flags)
```

Closing a connection

UDP is a connectionless protocol (nevermind the fakery of `connect` on a UDP socket), so there is no notion of establishing and tearing down a connection. The `close` system call on a UDP socket simply releases the socket and file descriptor back into the resource pool, and discards any pending, unread messages.

With TCP, on the other hand, the connection is a pair of one-way pipes. Each direction can be shut down independently.

`shutdown` closes one or both directions of a connected socket:

```
int shutdown(socket,how)
```

If *how* is 0, the receive direction is shut down. If *how* is 1, the transmit direction is shut down. If *how* is 2, both directions are closed.

When the transmit side is shut down, the remote end will receive an end-of-file indication when it has read all remaining data. Any data that have been sent but not yet received by the remote end remain queued until read.

When the receiving side is shut down, any pending unread data are discarded. An attempt by the remote system to write to the socket which has either been shutdown in the write direction locally, or in the read direction at the far end, will result in EPIPE or delivery of SIGPIPE, just as is the case with local pipes. Be aware, however, that some Linux kernel versions will not generate a SIGPIPE but instead will fail write with an ECONNRESET if the TCP connection times out with data pending in the buffer and additional writers are attempted.

When a TCP socket is closed, either with an explicit `close` system call or at process exit, both directions are also closed down, as if `shutdown(s,2)` had been invoked. Any data which are queued to be transmitted at the network layer are **not** discarded. The TCP connection continues to exist even after the close. However, if those queued data are ultimately not deliverable, that error status would be lost. An option can be set (SO_LINGER) using the `setsockopt` system call which will force the `close` to block until all queued data are successfully transmitted to the far end. For this reason, it is important to check the return value from close.

Using the DNS

Because 32 bit integers are not terribly mnemonic host names, the Internet provides a distributed database called the Domain Name System (DNS) to translate between numeric IP addresses and hierarchical, textual names, such as `cooper.edu`. DNS requests and responses are via UDP port 53. Applications use a library to isolate themselves from the details of the DNS protocol.

This mapping is potentially many-to-many. While the case of a single IP address equating to a single hostname is the most common, there are frequent cases where a single name has multiple IP addresses, or a single IP address has multiple names.

UNIX provides standard library routines to query the Domain Name System. `gethostbyname` takes a domain name and returns a list of IP addresses: `gethostbyaddr` performs the reverse operation, taking an IP adddress and returning a list of names. Both are not re-entrant safe, in that they return a pointer to static data (and furthermore the underlying structure contains further pointers to static data).

```
#include <netdb.h>
extern int h_errno;
```

```
struct hostent *gethostbyname(const char *name);

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

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]  /* for backward compatibility */
```

For simplicity, `h_name` and `h_addr` can be used by applications which don't care about the possible existence of multiple names or addresses.

In addition, some utility functions are available to convert between dotted decimal notation (e.g. 10.1.2.3) and an unsigned long:

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

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

unsigned long int inet_addr(const char *cp);

char *inet_ntoa(struct in_addr in);
```

`inet_addr` parses a dotted-decimal string and returns an unsigned long (32 bit) address, already in network byte order. If the string is not parseable, the all-ones value (-1) is returned. This is 255.255.255.255 in dotted quad notation, which is never a valid host address. However, it does represent the broadcast address which is used in certain system applications and thus *inet_addr* is **unclean**. *inet_aton* is preferred although some older UNIX variants do not support it. It returns zero on failure and non-zero on success. Here's an example of looking up a name:

```
f(nm)
char *nm;
{
 struct hostent *he;
 struct sockaddr_in sin;
       if ((sin.sin_addr.s_addr=inet_addr(nm))== -1)
        {
                if (!(he=gethostbyname(nm)))
                {
                        fprintf(stderr,"Unknown host:%s",nm);
                             herror(" ");
                        return -1;
                }
                memcpy(&sin.sin_addr.s_addr,he->h_addr_list[0],
                             sizeof sin.sin_addr.s_addr);
        }
```

The Domain Name System has its own set of error codes, which are distinct from system call error codes. These codes can distinguish, for example, between a negative confirmation (requested name or address definitely doesn't exist) and a system failure (no answer could be determined at this time). `herror` prints a symbolic representation of the error code of the last failed DNS operation.

Recent UNIX distributions have a new interface called `getaddrinfo`, and declare that functions such as `gethostbyname` are obsolescent. Programmers should beware that sometimes using the fancy new function makes it difficult to get your code to work on older systems.

Handling multiple sockets with select

Read and write operations on sockets may block, which causes problems for programs which need to simultaneously communicate on multiple sockets. There are two approaches to solving this: using a multi-threaded program, and using non-blocking I/O in conjunction with the `select` system call. We will not discuss threads in this unit.

To put a socket (or any other file descriptor) into non-blocking mode, use `fcntl`:

```
          s=socket(....)
          /* connect, etc. */
          if (fcntl(s,F_SETFL,O_NONBLOCK)<0) perror("fcntl");
          n=read(s,buf,sizeof buf);
          if (n<0)
          {
                    if (errno==EWOULDBLOCK) /*it's ok */
                    else perror("socket read");
          }
```

Now we can use `read`, `write`,`recvfrom` and `sendto` without fear of our program becoming "stuck" in a blocking operation. The system call will just fail temporarily with an error of `EWOULDBLOCK`. (In some UNIX variants, `EAGAIN` is the error code). We can try again. However, this is only half the battle. If we were to write a program which just keeps on banging on the operation in a tight loop when it encounters `EWOULDBLOCK`, this would be what is known as a "spin loop" or "busy-wait" loop. It would mean that the program is consuming vast amounts of user and cpu time to accomplish nothing. What we need is a way of blocking and keeping an eye on multiple file descriptors at once.

The `select` system call can be used to monitor any number of file descriptors to see when reading or writing operations would succeed (at least partially) without blocking. The syntax is somewhat awkward:

```
/* Read man page on select for some discussion of new vs old
   ways of getting the right header files.  */
#include <sys/types.h>
#include <unistd.h>
#include <sys/select.h>



main()
{
 int s1,s2,maxfd;
 fd_set readfds,writefds;
          /* open sockets s1 and s2, connect, etc. */
          if (s1>s2) maxfd=s1; else maxfd=s2;
          if (fcntl(s1,F_SETFL,O_NONBLOCK)<0) {perror("fcntl s1");return -1;}
          if (fcntl(s2,F_SETFL,O_NONBLOCK)<0) {perror("fcntl s2");return -1;}
          for(;;)
          {
                    FD_ZERO(&readfds);
```

```
                FD_ZERO(&writefds);
                FD_SET(s1,&readfds);
                FD_SET(s2,&writefds);
                if (select(maxfd+1,&readfds,&writefds,NULL,NULL)<0)
                {
                        perror("select error");
                        return -1;
                }
                if (FD_ISSET(s1,&readfs))
                {
                        while ((n=read(s1,...)))>0  // should read something
                                /*...*/
                }
                if (FD_ISSET(s2,&writefds))
                {
                        while ((n=write(s2,...))>0) // should write at least 1 byte
                                /*...*/
                }
        }
```

The `select` system call operates on bit vectors which represent a set of file descriptors. The mechanics of manipulating these bit vectors are best left to the macros such as `FD_SET` which are provided by the include files. The second, third and fourth parameters to `select` are pointers to these bit vectors. They are each file descriptor sets that we are "interested in". The second argument represents file descriptors for which we want to monitor read-ready events, the next is a similar set for write-ready events, and the last is described as "exception events". That's how the man page describes the sets, but to be more precise, at least with respect to Linux kernel:

- A "read-ready" event is any of the following: at least one byte of data available to be read; there has been an end-of-file condition (including remote end of socket connection has closed); there has been an error posted against the file descriptor.
- A "write-ready" event is any of the following: At least one byte of data may be written without blocking; there has been an error posted against the file descriptor.
- The "exception" event is only used for an obscure sockets programming feature called "out-of-band" or "urgent" data delivery, and will not be discussed further here.

The first argument to `select` is the highest file descriptor number set in **any** of the bit vectors being passed in, **plus one**. Another way of looking at it is: the maximum length of the bit vectors. This is a problematic part of the API. While one could just set this to some rather large number (there are various ways, some static, some run-time, of querying the maximum number of open files), doing so imposes a performance penalty in the kernel, which must examine each of the passed vectors to the limit that the first argument gives. Many programmers write utilities to front-end select and manage the bit vectors so as to keep track of the highest file descriptor number.

The fifth argument to `select` is a timeout. It is represented by a pointer to a `struct timeval`, which gives both seconds and microseconds. If the argument is `NULL`, then there is no timeout and select blocks until one or more of the events we are interested in arrives. The `struct timeval` may be set with both elements zero, in which case `select` returns immediately.

Upon return from select, the bit vectors which were passed as pointers are modified to reflect which file descriptors had the corresponding events pending. The return value from select is the total number of bits which are set in all three vectors. Most programmers do not make use of this number except to detect error (-1 return), and instead, knowing which file descriptors were registered going in, query the result vector and act accordingly. Since the vectors are over-written by the kernel, it is often useful to make a backup copy.

poll system call

`select` is a potentially tricky interface to use, and many programmers prefer to use threads. However, multi-threaded programming has its own sets of pitfalls and gotchas too. There is another system call `poll` which performs essentially the same task as `select`, but with a different interface. In many ways it is a superior interface, but because historically `poll` was not available on many UNIX variants, most programmers gravitate towards `select`. The example below should be self-explanatory with the assistance of the man page for poll(2).

```
struct pollfd pollarry[3];
          pollarry[0].fd=fd_readsocket;
          pollarry[0].events=POLLIN;
          pollarry[1].fd=fd_writesocket;
          pollarry[1].events=POLLOUT;
          pollarry[2].fd=fd_controlsocket;
          pollarry[2].events=POLLIN|POLLOUT;
          if (poll(&pollfd,3,-1)<0)
                    perror("poll");
          else
          {
                    if (pollarry[0].revents&POLLIN)
                              do_read(pollarry[0].fd);
                    if (pollarry[1].revents & (POLLERR|POLLHUP))
                              do_error(pollarry[1].fd);
                    //etc.
          }
```