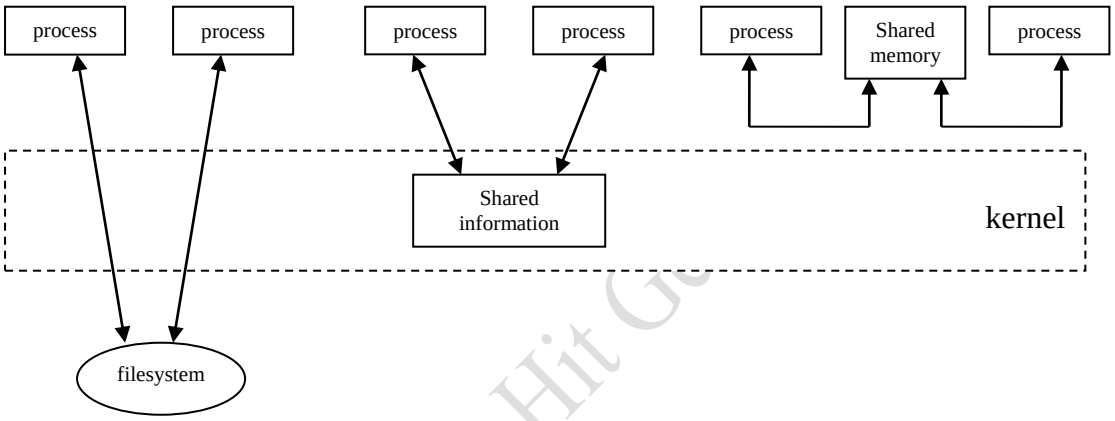
UNIT 1: Network Programming Fundamentals

Computer network programming involves writing computer programs that enable process to communicate with each other across a computer network or within same system in order to share information and resources.

In the traditional UNIX programming model, there are multiple processes running on a system, with each process having its own address space. Information can be shared in various ways:

- The two processes share some information that resides in a file in the filesystem. To access this data, each process must go through the kernel (e.g. read, write, lseek, etc). In order to protect multiple writers from each other, and to protect one or more readers from a writer, some form of synchronization is required when a file is being updated.
- The two processes share some information that resides within the kernel. Pipe, System V message queues and System V Semaphores are the examples of this type of sharing. Each operation has to access the shared information now involves a system call into the kernel.
- The two processes have a region of shared memory that each process can reference. Once the shared memory is set up by each process, the processes can access the data in the shared memory without involving the kernel at all. Some form of synchronization is required by the processes that are sharing the memory.



Three ways to share information between UNIX processes

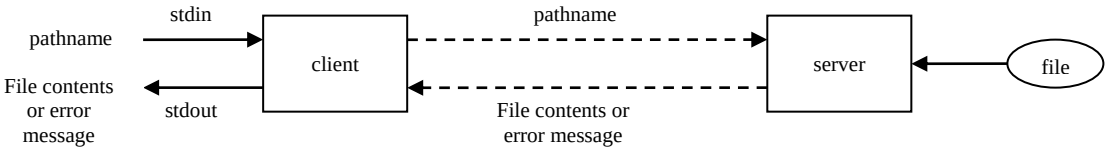
A Simple Client-Server Mode:

When writing programs that communicate across a computer network, one must first invent a protocol, an agreement on how those programs will communicate. Before delving into the design details of a protocol, high-level decisions must be made about which program is expected to initiate communication and when responses are expected.

In general an application that initiates peer-to-peer communication is called a client. End users usually invoke client software when they use a network service. Most client software consists of conventional application programs. Each time a client application executes it contacts a server, send request, and awaits a response. When the response arrives, the client continues processing. Clients are easier to build than server, and usually require no special system privileges to operate.

A server is any program that wait for incoming communication requests from a client. The server receives a client's request. Perform the necessary computation, and returns the result to the client.

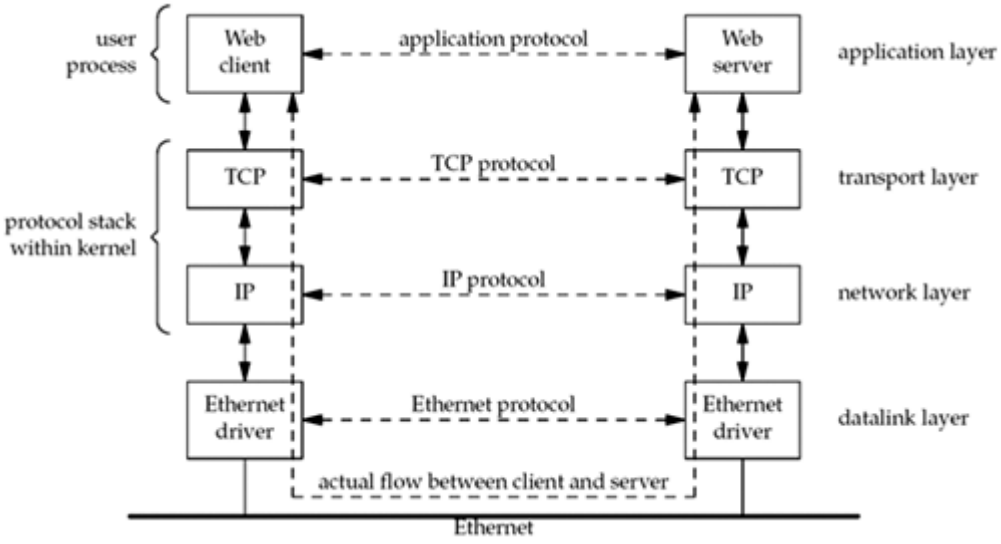
Example



Client-server example

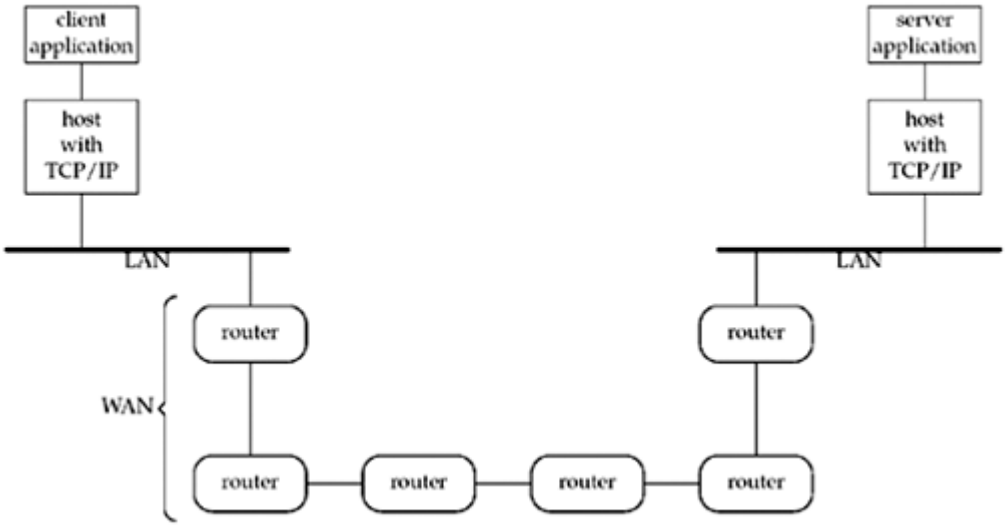
Here, the client reads a pathname from the standard input and writes it to the IPC channel. The server reads this pathname from the IPC channel and tries to open the file for reading. If the server can open the file, the server responds by reading the file and writing it to the IPC channel. Otherwise, the server responds with an error message. Then the client reads from the IPC channel, writing what it receives to the standard output. If the file cannot be read by the server, the client reads an error message from the channel. Otherwise, the client reads the contents of the file.

Client and server on the same Ethernet communicating using TCP.



Even though the client and server communicate using an application protocol, the transport layers communicate using TCP.

Client and server on different LANs connected through a WAN



Client-Server Design issue

1. Privilege and complexity: sever must contain code that handles the issues of
    a. Authentication
    b. Authorization
    c. Data security
    d. Privacy
    e. Protection
2. Standard vs. nonstandard client software
    a. Standard application: consists of those services defined by TCP/IP and assigned well-known, universally recognized protocol identifiers.
    b. Nonstandard application: locally defined application services or nonstandard application services.
3. Parameterization of client: client software allows the user to specify both the remote machine on which a server operates and the protocol port number at which the server is listening. Eg telnet machine-name port no
4. Connectionless vs Connection-oriented servers
5. Stateless vs stateful servers: Information that a server maintains about state of ongoing interaction with client is called state information. Server that donot keep any sate information are called stateless servers.

Communication Protocol

- Communicating systems use well-defined formats for exchanging messages. Each message has an exact meaning intended to provoke a defined response of the receiver. A protocol therefore describes the syntax, semantics, and synchronization of communication.
- Communications protocol is a formal description of digital message formats and the rules for exchanging those messages in or between computing systems and in telecommunications.
- Protocols may include signaling, authentication and error detection and correction capabilities.
- A protocol defines the syntax, semantics, and synchronization of communication, and the specified behaviour is typically independent of how it is to be implemented. A protocol can therefore be implemented as hardware or software or both.
- To ease design, communications protocols are structured using a layering scheme as a basis. Instead of using a single universal protocol to handle all transmission tasks.
- In digital computing systems, the rules can be expressed by algorithms and data structures. Expressing the algorithms in a portable programming language, makes the protocol software operating system independent.

Basic requirements of protocols (for information)

Messages are sent and received on communicating systems to establish communications. Protocols should therefore specify rules governing the transmission. In general, much of the following should be addressed:

- *Data formats for data exchange.* Digital message bitstrings are exchanged. The bitstrings are divided in fields and each field carries information relevant to the protocol. Conceptually the bitstring is divided into two parts called the header area and the data area. The actual message is stored in the data area, so the header area contains the fields with more relevance to the protocol. The transmissions are limited in size, bitstrings longer than the maximum transmission unit(MTU) are divided in pieces of appropriate size. Each piece has almost the same header area contents.
- *Address formats for data exchange.* Addresses are used to identify both the sender and the intended receiver(s). The addresses are stored in the header area of the bitstrings, allowing the receivers to determine whether the bitstrings are intended for themselves and should be processed or should be discarded. A connection between a sender and a receiver can be identified using an address pair (sender address, receiver address).
- *Address mapping.* Sometimes protocols need to map addresses of one scheme on addresses of another scheme. For instance to translate a logical IP address specified by the application to a hardware address. This is referred to as address mapping.
- *Routing.* When systems are not directly connected, intermediary systems along the route to the intended receiver(s) need to forward messages on behalf of the sender. Determining the route the message should take is called routing. On the Internet, the networks are connected using routers. This way of connecting networks is called internetworking.

- *Detection of transmission errors* is necessary on networks which cannot guarantee error-free operation. In a common approach, CRCs of the data area are added to the end of packets, making it possible for the receiver to detect differences caused by errors. The receiver rejects the packets on CRC differences and arranges somehow for retransmission.
- *Acknowledgements* of correct reception of packets by the receiver may be used to prevent the sender from retransmitting the packets. Some protocols, notably datagram protocols like the Internet Protocol (IP), do not acknowledge.
- *Loss of information* - timeouts and retries. Sometimes packets are lost on the network or suffer from long delays. To cope with this, a sender expects an acknowledgement of correct reception from the receiver within a certain amount of time. On timeouts, the packet is retransmitted. In case of a broken link the retransmission has no effect, so the number of retransmissions is limited. Exceeding the retry limit is considered an error.
- *Direction of information flow* needs to be addressed if transmissions can only occur in one direction at a time as on half-duplex links. This is known as Media Access Control.
- *Sequence control*. We have seen that long bitstrings are divided in pieces, that are send on the network individually. The pieces may get 'lost' on the network or arrive out of sequence, because the pieces can take different routes to their destination on some types of networks. Pieces may be needlessly retransmitted resulting in duplicate pieces. By sequencing the pieces at the sender, the receiver can determine what was lost or duplicated and ask for retransmissions, if necessary, the pieces may be reassembled.
- *Flow control* is needed when the sender transmits faster than the receiver or intermediate network equipment can process the transmissions. Flow control can be implemented by messaging from receiver to sender.

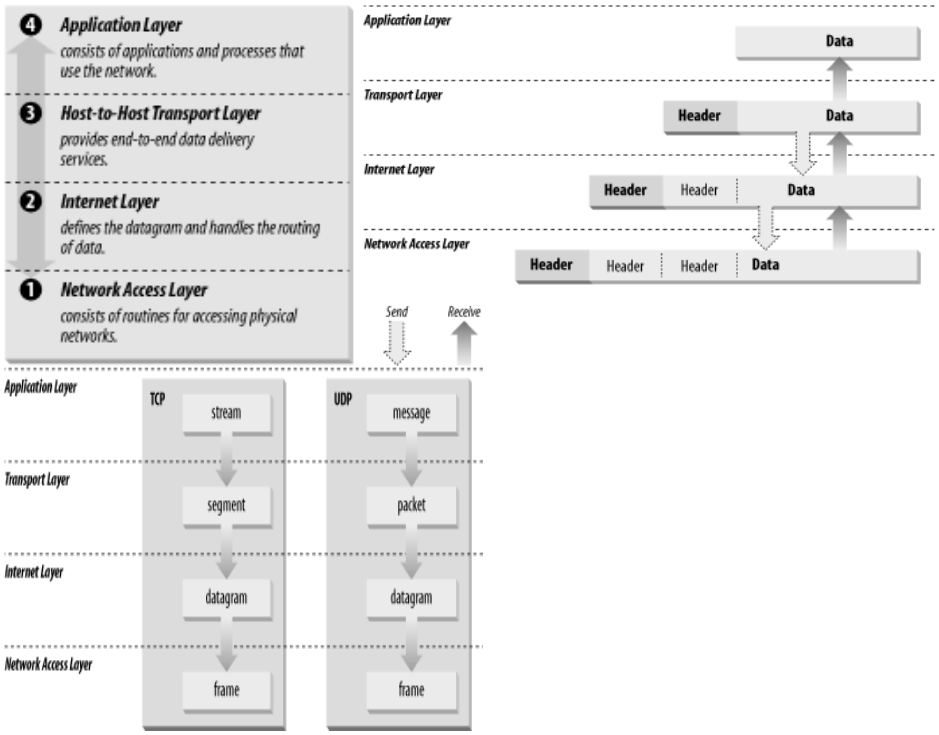
TCP/IP PROTOCOL.

- 1968 First proposal for ARPANET – military & gov' Contracted to Bolt, Beranek & Newman
- 1971 ARPANET enters regular use
- 1973/4 redesign of lower level protocols leads to TCP/IP
- 1983 Berkeley TCP/IP implementation for 4.2BSD public domain code
- 1980s rapid growth of NSFNET – broad academic use
- 1990s WWW and public access to the Internet

TCP/IP Features

The popularity of the TCP/IP protocols did not grow rapidly just because the protocols were there, or because connecting to the Internet mandated their use. They met an important need (worldwide data communication) at the right time, and they had several important features that allowed them to meet this need. These features are:

- Open protocol standards, freely available and developed independently from any specific computer hardware or operating system. Because it is so widely supported, TCP/IP is ideal for uniting different hardware and software components, even if you don't communicate over the Internet.
- Independence from specific physical network hardware. This allows TCP/IP to integrate many different kinds of networks. TCP/IP can be run over an Ethernet, a DSL connection, a dial-up line, an optical network, and virtually any other kind of physical transmission medium.
- A common addressing scheme that allows any TCP/IP device to uniquely address any other device in the entire network, even if the network is as large as the worldwide Internet.
- Standardized high-level protocols for consistent, widely available user services.



The TCP/IP architecture Data encapsulation Data structures

1. Internet Protocol

The Internet Protocol is the building block of the Internet. Its functions include:

- Defining the datagram, which is the basic unit of transmission in the Internet
- Defining the Internet addressing scheme
- Moving data between the Network Access Layer and the Transport Layer
- Routing datagrams to remote hosts
- Performing fragmentation and re-assembly of datagrams

Before describing these functions in more detail, let's look at some of IP's characteristics. First, IP is a *connectionless protocol*. This means that it does not exchange control information (called a "handshake") to establish an end-to-end connection before transmitting data. In contrast, a *connection-oriented protocol* exchanges control information with the remote system to verify that it is ready to receive data before any data is sent. When the handshaking is successful, the systems are said to have established a *connection*. The Internet Protocol relies on protocols in other layers to establish the connection if they require connection-oriented service.

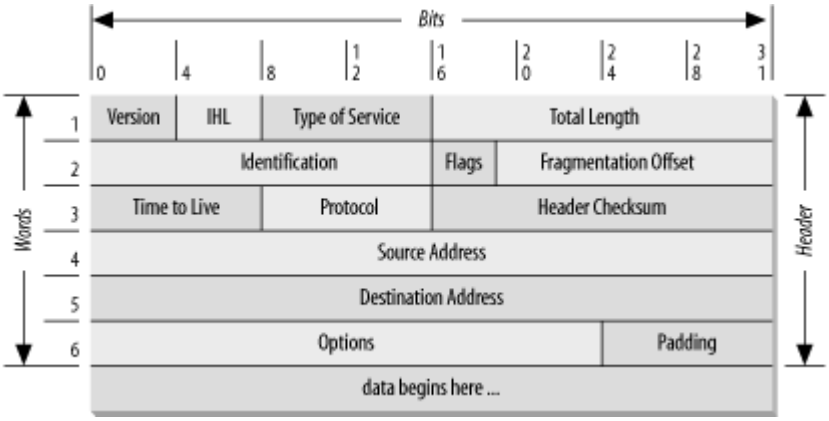
IP also relies on protocols in the other layers to provide error detection and error recovery. The Internet Protocol is sometimes called an *unreliable protocol* because it contains no error detection and recovery code. This is not to say that the protocol cannot be relied onquite the contrary. IP can be relied upon to accurately deliver your data to the connected network, but it doesn't check whether that data was correctly received. Protocols in other layers of the TCP/IP architecture provide this checking when it is required.

1.1 The datagram

The TCP/IP protocols were built to transmit data over the ARPAnet, which was a *packet-switching network*. A *packet* is a block of data that carries with it the information necessary to deliver it, similar to a postal letter, which has an address written on its envelope. A packet-switching network uses the addressing information in the packets to switch packets from one physical network to another, moving them toward their final destination. Each packet travels the network independently of any other packet.

The *datagram* is the packet format defined by the Internet Protocol. Figure 2.1 is a pictorial representation of an IP datagram. The first five or six 32-bit words of the datagram are control information called the *header*. By default, the header is five words long; the sixth word is optional. Because the header's length is variable, it includes a field called *Internet Header Length* (IHL) that indicates the header's length in words. The header contains all the information necessary to deliver the packet.

Figure 2.1. IP datagram format



The Internet Protocol delivers the datagram by checking the *Destination Address* in word 5 of the header. The Destination Address is a standard 32-bit IP address that identifies the destination network and the specific host on that network. If the Destination Address is the address of a host on the local network, the packet is delivered directly to the destination. If the Destination Address is not on the local network, the packet is passed to a gateway for delivery. *Gateways* are devices that switch packets between the different physical networks. Deciding which gateway to use is called *routing*. IP makes the routing decision for each individual packet.

UDP: User Datagram Protocol

UDP is a simple transport-layer protocol. The application writes a “datagram” to a UDP socket, which is encapsulated as either an IPv4 datagram or an IPv6 datagram. It is then sent to its destination. But there is no guarantee that a UDP datagram ever reaches its final destination. Problem that is encountered with network programming using UDP is its lack of reliability. If it is required to be certain that a datagram reaches its destination, numerous features will have to be built into our application such as acknowledgements from the other end, timeouts, retransmissions, etc.

Each UDP datagram has a length. If the datagram reaches its final destination correctly, then the length of the datagram is passed to the receiving application. As UDP provides a connectionless service, there need not be any long-term relationship between a UDP client and server. For instance a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server. Similarly a UDP server can receive five datagrams in a row on a single UDP socket, each from five different clients.

TCP: Transmission Control Protocol

The service provided by TCP to an application varies from that of UDP. First, TCP provides “connection” between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.

TCP also provides “reliability”. When TCP sends data to the other end, it requires an acknowledgement in return. If an acknowledgement is not received, TCP automatically retransmits the data and waits for a specific time period. After some retransmissions, TCP gives up. TCP contains algorithms to estimate the “round-trip time” (RTT) between a client and server dynamically so that it knows how long to wait for an acknowledgement. For instance, the RTT on a LAN can be milliseconds while across a WAN, it can be in seconds.

TCP also “sequences” the data by associating a sequence number with every byte that it sends. For instance, assume an application writes 2048 bytes to a TCP socket, causing TCP to send two segments; the first containing the data with sequence numbers 1-1024 and the second containing the data with sequence numbers 1025-2048. Here, a “segment” refers to the unit of data that TCP passes to IP. If the segments arrive out of order, the receiving TCP will reorder the two

segments based on their sequence numbers before passing the data to the receiving application (or the higher layers). If TCP receives duplicate data from its peer (say the sender thought a segment was lost and retransmitted it, when it wasn't really lost, the network was just overloaded), it can detect that the data has been duplicated from the sequence number and the duplicate data is discarded.

TCP further provides "flow control". TCP always tells its peer exactly how many bytes of data it is willing to accept from the peer. This is called the "advertised window". At a instance, the window is the amount of room currently available in the receive buffer, guaranteeing that the sender cannot overflow the receiver's buffer. The window changes dynamically: as data is received from the sender, the window size decreases, but as the receiving application reads data from the buffer, the window increases.

Finally, a TCP connection is also "full-duplex" i.e. an application can send and receive data in both directions on a given connection at anytime. For it, TCP must keep track of state information such as sequence numbers and window sizes for each direction of data flow: sending and receiving.

4. SCTP (Stream Control Transmission Protocol)

- Connection Oriented protocol:
- Reliable:
- Full duplex associations
- Provides a message service which maintains record boundaries:

TCP State Transition Diagram

Connection establishment

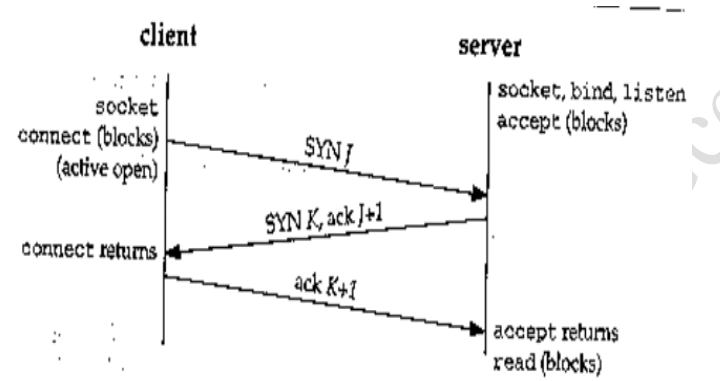
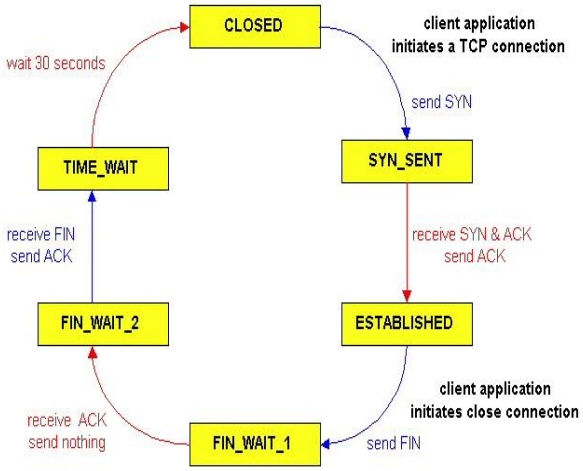


Figure 2.2 TCP three-way handshake.



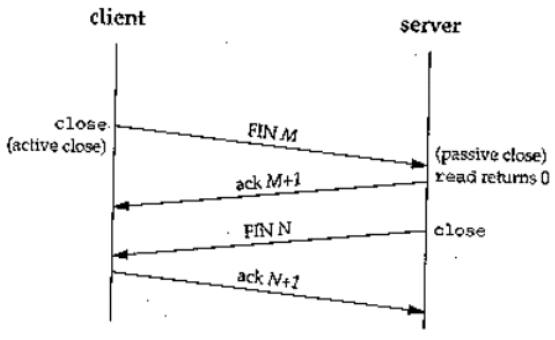
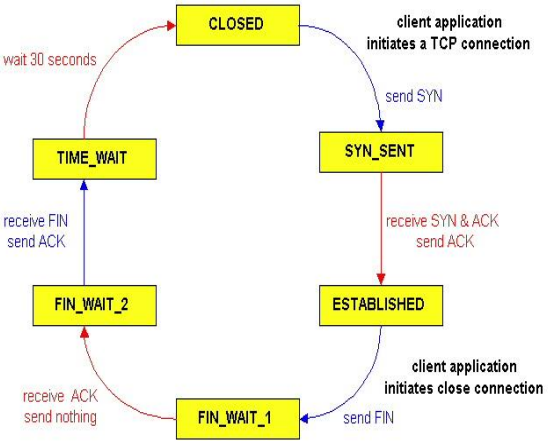
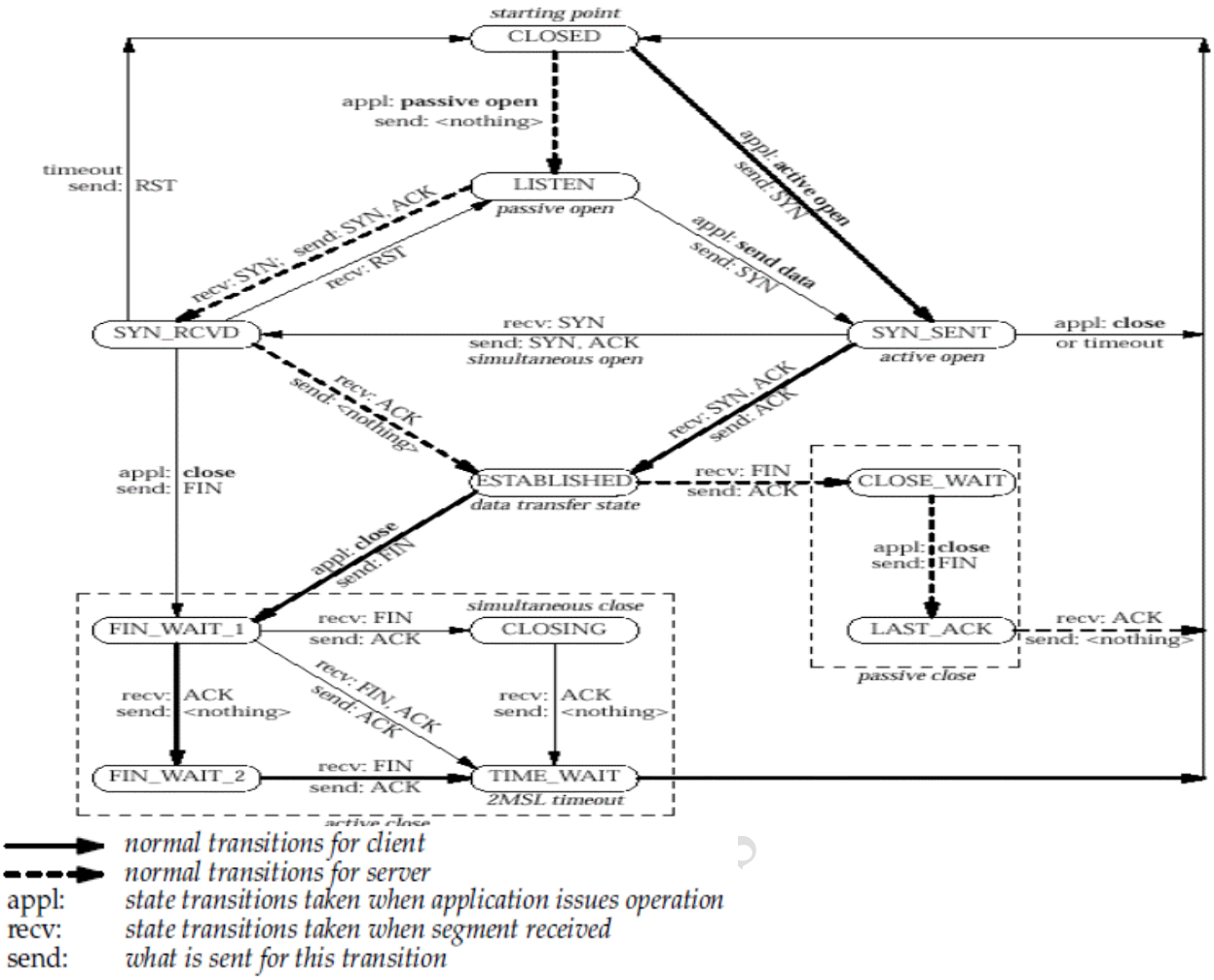


Figure 2.3 Packets exchanged when a TCP connection is closed.



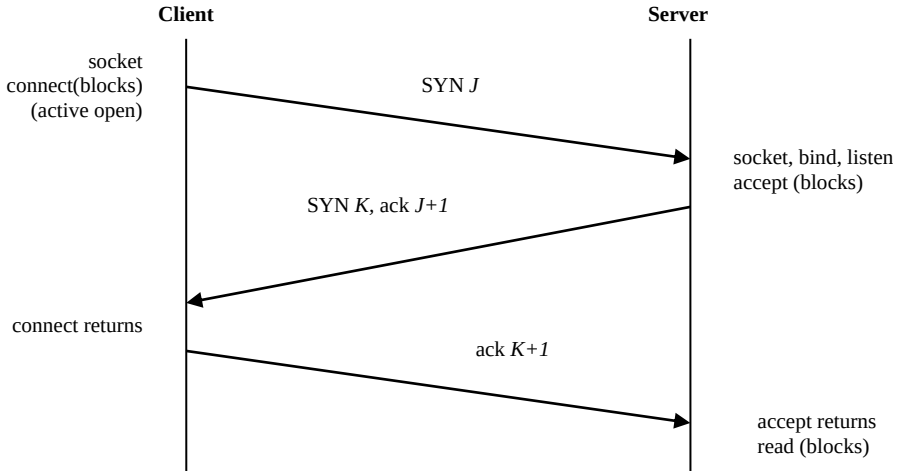


TCP Connection Establishment and Termination

Three-Way Handshake:

When a TCP connection is established, the following scenario occurs:

1. The server must be prepared to accept an incoming connection. This is normally done by called *socket, bind & listen.* It is known as a *passive open.*
2. The client issues an *active open* by calling *connect.* This causes the client TCP to send a SYN segment (synchronize) to tell the server about the client's initial sequence number for the data. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header, and possible TCP options.
3. The server must acknowledge the client's SYN and the server must also send its own SYN containing the initial sequence number. The sever sends its SYN and the ACK of the client's SYN in a sing segment.
4. The client must acknowledge the server's SYN.



The minimum number of packets required for this exchange is three; hence this is called TCP's "three-way handshake".

TCP options:

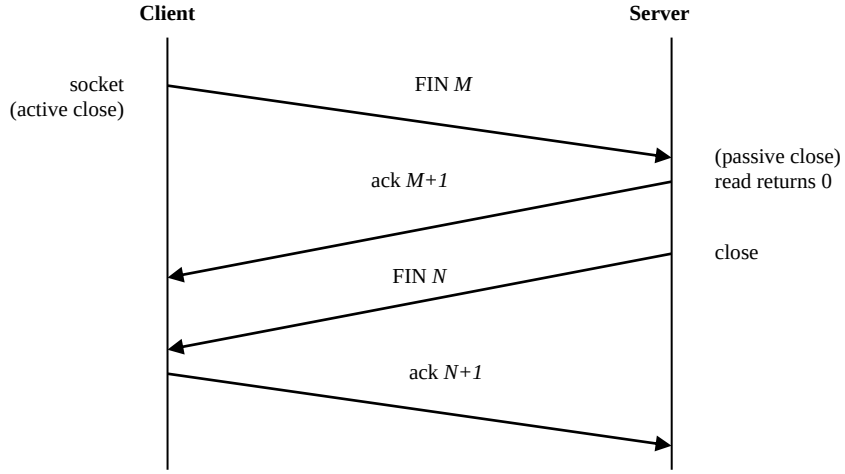
Each SYN can contain TCP options. Commonly used options are as follows:

- *MSS option:* SYN with this option announces its "maximum segment size", which is the maximum amount of data that is willing to accept in each TCP segment on this connection.
- *Window scale option:* refers to the maximum window that TCP can advertise to the other TCP is 65535.
- *Timestamp option:* is needed for high-speed connections to prevent possible data corruption due to lost packets which reappear.

TCP Connection Termination:

While it takes three segments to establish a connection, it takes four segments to terminate connection.

1. A peer calls "close", which refers that this end performs the "active close". This end's TCP sends a FIN segment, which means it is finished sending data.
2. The other peer that receives Fin performs the "passive close". The receipt of the FIN is also passed to the application as an end-of-file, which means the application will never receive any additional data on the connection, except those that may already be queued for the application to receive.
3. The application that received the end-of-file will "close" its socket. This causes its TCP to send a FIN.
4. The TCP on the system that receives this final FIN acknowledges the FIN.



Packets exchanged when a TCP connection is closed

Maximum Segment Lifetime:

MSL is the maximum amount of time that any given IP datagram can live in the Internet. During every implementation of TCP, a value for the MSL must be chosen. The recommended value

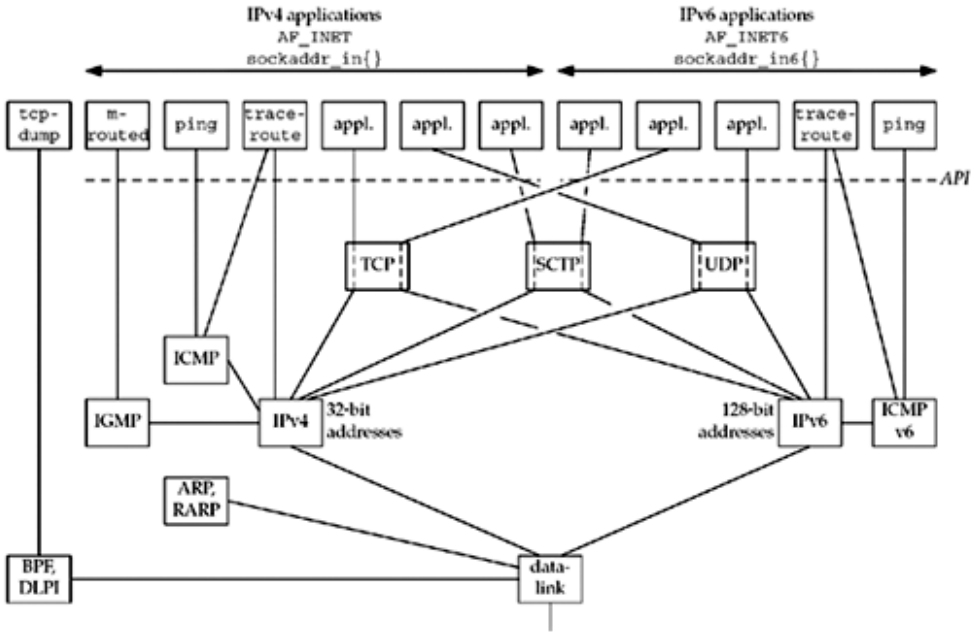
in RFC 1122 is 2 minutes, although Berkeley-derived implementations have traditionally used a value of 30 seconds instead. TIME_WAIT state is the one when the active close connection occurs. The duration of TIME_WAIT state is double of the MSL i.e. between 1 and 4 minutes. This time is bounded because every datagram contains an 8-bit hop limit with a maximum value of 255. The assumption is made that a packet with a maximum hop limit of 255 cannot exist in the Internet for more than MSL seconds.

The way in which a packet gets "lost" in the Internet is usually the result of routing irregularities. A router crashes or a link between two routers goes down and it takes the routing protocols some seconds or minutes to stabilize and find an alternate path. During that time period, routing loops can occur (router A sends packets to router B, and B sends them back to A) and packets can get caught in these loops. Meanwhile, assuming the lost packet is a TCP segment, the sending TCP times out and retransmits the packet, and the retransmitted packet gets to the final destination by some alternative path. But sometime later, if the routing loop is corrected and the packet that was lost in the loop is sent to the final destination. This original packet is called a "lost duplicate" or a "wandering duplicate". TCP must handle such conditions. There as two reasons for the TIME_WAIT state:

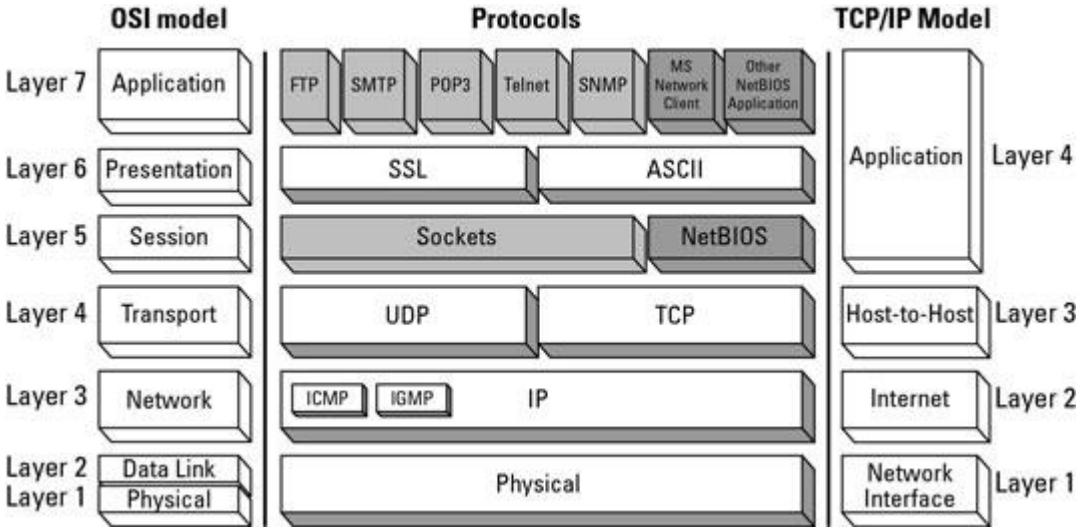
1. to implement TCP's full duplex connection termination reliably
2. to allow old duplicate segments to expire in the network

Protocol comparison

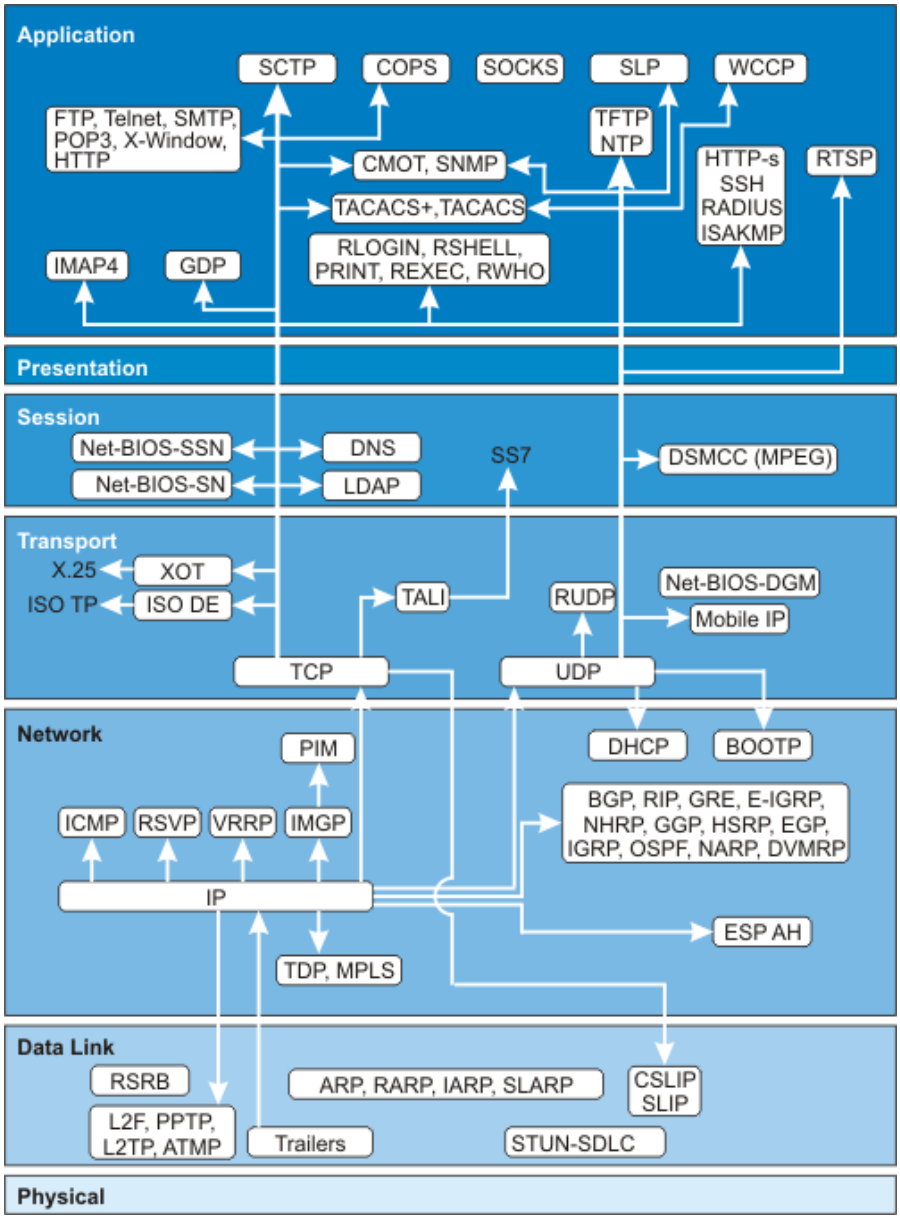
Figure Overview of TCP/IP protocols.



Below figure shows the general protocols involved in OSI and TCP/IP Model



The TCP/IP suite is illustrated here in relation to the OSI model: with all possible protocols



For LAB Session 1.

For example: 1

```
#include <stdio.h>
#define MAXLINE 1024
main()
{ int n;
char line[MAXLINE];
FILE *fp;

fp=popen("cat .cshrc", "r");

\*read the lines in .cshrc from fp*\
while ((fgets(line, MAXLINE, fp)) != NULL) {
   n=strlen(line);
      if (write(1, line, n)!=n) printf("print data error");
pclose(fp);
}
```

Example 2.

```
/*Named_pipe half duplex server (namedpipeserver.c)*/
/*Named pipes allow two unrelated processes to communicate with each other.
 They are also known as FIFOs (first-in, first-out) and
  can be used to establish a one-way (half-duplex) flow of data.*/

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define MAX_BUF_SIZE 256

int main(int argc, char *argv[])
{
    int fd, ret_val, count, numread;
    char buf[MAX_BUF_SIZE];

    /* Create the named - pipe. This creates a named pipe in the given path called as halfduplex */
    ret_val = mkfifo("/home/hit1/public_html/np/halfduplex", 0666);

    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }

    /* Open the pipe for reading */
    fd = open("/home/hit1/public_html/np/halfduplex", O_RDONLY);

    /* Read from the pipe */
    numread = read(fd, buf, MAX_BUF_SIZE);

    buf[numread] = '0';

    printf("Half Duplex Server : Read From thepipe : %s\n", buf);
```

```
    /* Convert to the string to upper case */
    count = 0;
    while (count < numread) {
        buf[count] = toupper(buf[count]);
        count++;
    }

    printf("Half Duplex Server : Converted String : %s\n", buf);
}
```

Example 3.

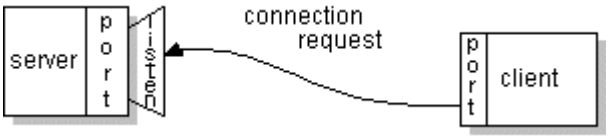
```
/*Named_pipe half duplex server (namedpipeclient.c)*/
/*Named pipes allow two unrelated processes to communicate with each other.
 They are also known as FIFOs (first-in, first-out) and
  can be used to establish a one-way (half-duplex) flow of data.*/

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int fd;

    /* Check if an argument was specified. */

    if (argc != 2) {
        printf("Usage : %s <string to be sent to the server>n", argv[0]);
        exit (1);
    }

    /* Open the pipe for writing */
    fd = open("/home/hit1/public_html/np/abc", O_WRONLY);

    /* Write to the pipe */
    write(fd, argv[1], strlen(argv[1]));
}
```

UNIT– 2: UNIX Programming

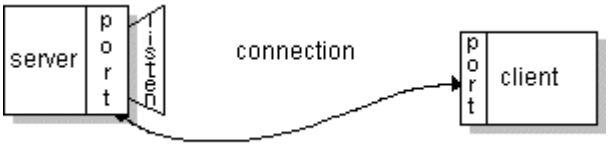
What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition: A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

Berkeley Sockets

The Berkeley sockets application programming interface (API) comprises a library for developing applications in the C programming language that perform inter-process communication, most commonly across a computer network.

Berkeley sockets (also known as the BSD socket API) originated with the 4.2BSD Unix operating system (released in 1983) as an API. Only in 1989, however, could UC Berkeley release versions of its operating system and networking library free from the licensing constraints of AT&T's copyright-protected Unix.

The Berkeley socket API forms the de facto standard abstraction for network sockets. Most other programing languages use a similar interface as the C API.

- Many Unix systems started with some version of the BSD networking code, including the sockets API, and we refer to these implementations as Berkeley-derived implementations.

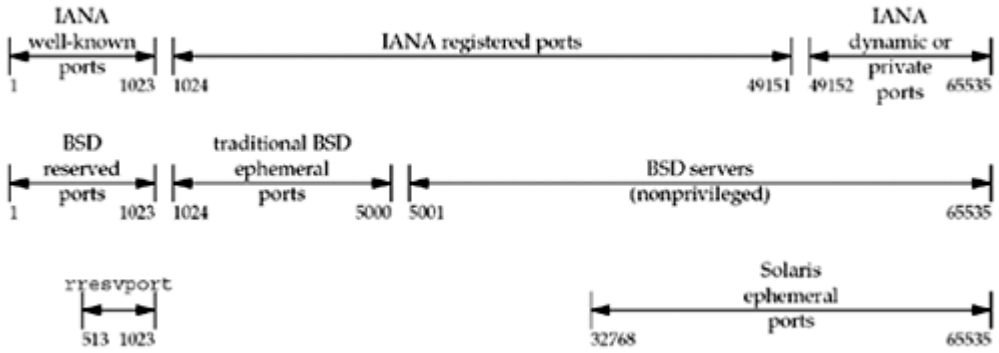
Port Numbers:

At any given time, multiple processes can use either UDP or TCP. Both TCP and UDP use 16-bit integer “port number” to differentiate between these processes. The port number is used

between the two host computers to identify which application program is to receive the incoming traffic. The port numbers are divided into three ranges:

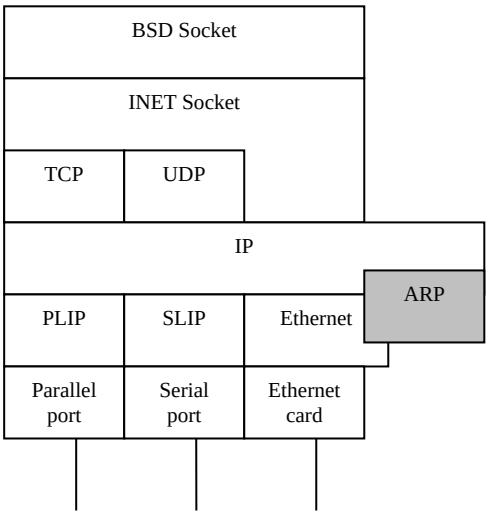
1. *Well-known ports:* 0 through 1023. These port numbers are controlled and assigned by IANA (Internet Assigned Numbers Authority). When possible, the same port is assigned to a given service for both TCP and UDP.
2. *Registered ports:* 1024 through 49151. These ports are not controlled by the IANA, but the IANA registers and lists the uses of these ports as a convenience to the community.
3. *Dynamic or private ports:* 49152 through 65535. These ports are also known as "ephemeral ports" i.e. short-lived ports. These ports are normally assigned automatically by TCP or UDP to the client. Clients normally don't care about the values of these ports; they just need to be certain that the ephemeral port is unique on the client host.

Allocation of port numbers.



The layer model of the network implementation:

When a process communicates via the network, it uses the functions provided by the BSD socket layer, which administers a general data structure for sockets, known as BSD sockets. The BSD socket interface simplifies the porting of network applications which are pretty complex. INET socket layer is below the BSD socket layer that manages the communication end points for the IP-based protocols TCP and UDP. These are represented by the data structure *sock,* which is known as INET sockets. The layer below INET socket layer is dependent of the type of the socket which can be either TCP or UDP layer or the IP layer directly. The UDP layer implements the *User Datagram Protocol* on the basis of IP, and the TCP layer implements *Transmission Control Protocol* for reliable communication links. The IP layer contains the code for the *Internet Protocol.* Below the IP layer are the network devices, to which the IP passes the final packets. These are responsible for the physical transport of the information. True communication occurs between two sides, producing a two-way flow of information. Hence the various layers are also connected together in the opposite direction, i.e. when IP packets are received; they are passed to the IP layer by the network devices and processed.



The Layer structure of a network

Socket Address Structure

Most of the socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begin with "sockaddr_" with a distinct suffix for each protocol suite. A "sockaddr_in" structure contains an "in_addr" structure as a member field.

IPv4 Socket Address Structure:

An IPv4 socket address structure is also known as "Internet socket address structure", which is named as "sockaddr_in" and is defined by including the "<netinet/in.h>" header. The structure is as follows:

```
struct sockaddr_in      {
      uint8_t           sin_len;          /* length of structure (16) */
      sa_family_t       sin_family;       /* AF_INET */
      in_port_t         sin_port;         /* 16-bit TCP or UDP port number */
                                          /* network byte ordered */
      struct in_addr    sin_addr;         /* 32-bit IPv4 address */
                                          /* network byte ordered */
      char              sin_zero;         /* unused */
};

struct in_addr {
      in_addr_t   s_addr;           /* 32-bit IPv4 address */
                                    /* network byte ordered */
};
```

Here, the length member "sin_len" is added with 4.3BSD-Reno for supporting OSI protocols, "sin_family" is an unsigned short. Despite most of the vendors do not supports a length field for socket addresses structures; it simplifies the handling of variable-length socket address structures.

In Berkeley-derived implementations, socket functions "bind, connect, sendto and sendmsg" pass a socket address structure from the process to the kernel through the "sockargs" function. This function copies the socket address structure from the process and sets its "sin_len" member to the size of the structure that was passed as an argument to these four functions. Whereas the five socket functions "accept, recvfrom, recvmsg, getpeername and getsockname" pass a socket address structure from the kernel to the process and set the "sin_len" before returning to the process.

The POSIX specification requires only three members in the structure: "sin_family, sin_addr, and sin_port". Almost all implementations add the "sin_zero" member so that all socket address structures are at least 16 bytes in size. "sin_zero" is unused and it is always set to "0".

Datatypes required by the POSIX specification:

Datatype	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure	<sys/socket.h>
socklen_t	Length of socket address structure	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32-bit	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16-bit	<netinet/in.h>

Generic Socket Address Structure:

Socket address structures are always passed by reference when they are passed as an argument to any of the socket functions. But the socket functions that take one of the pointers as an argument must deal with socket address structures from any of the supported protocol. Here the problem is how to declare the type of pointer that is passed. As a solution, a *generic* socket address structure was defined in the "<sys/socket.h>" header. The socket functions are then

defined as taking a pointer to the generic socket address structure. From an application programmer's point of view, generic socket address structure is only used to cast pointers to protocol specific structures.

```
struct sockaddr {
uint8_t     sa_len;
sa_family_t sa_family;
char        sa_data[14];
};
```

The "sa_family" field specifies the type of protocol. For TCP/IP, this field is always set to AF_INET. The remaining 14 bytes (sa_data) of this structure are always protocol dependent. For TCP/IP, IP addresses and port numbers are placed in this field. To facilitate operating with these fields, a specific type of socket address structure is used instead of the one above.

Browsing the header file reveals that this really isn't the form of the structure. It's really a very complicated union designed to hold an IP address in a variety of ways. Regardless, the "in_addr" struct is exactly 4 bytes long, which is the same size as an IP address. In the "sockaddr_in" structure, the "sin_port" field is a 16-bit unsigned value used to represent a port number. It's important to remember that these fields always need to be set and interpreted in network byte order. For example:

```
struct sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_port = htons(9999)
sin.sin_addr.s_addr = inet_addr("128.227.224.3");
```

In the above code example, the structure sin, holds the IP address, 128.227.224.3, and references the port number 9999. Two utility functions are used to set these values. The function "htons" returns the integer argument passed into it in network byte order. The function "inet_addr" converts the string argument from a dotted-quad into a 32-bit integer. Its return value is also in network byte order.

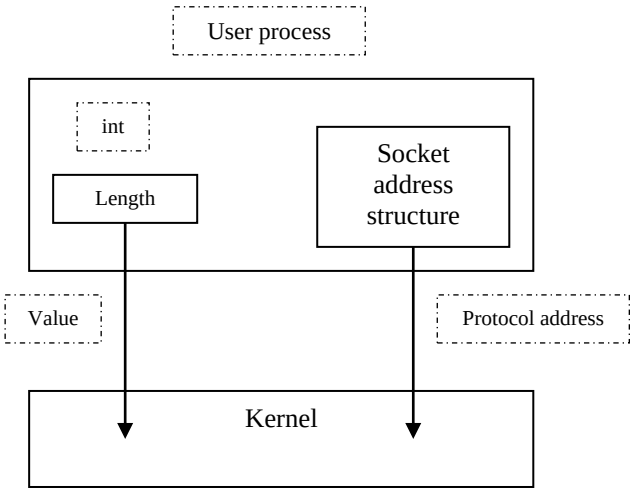
Value-Result Arguments:

When a socket address structure is passed to any of the socket functions, it is always passed by reference, i.e. a pointer to the structure is passed. In addition to it, the length of the structure is also passed as an argument. But the way of passing the length depends on the direction in which the structure is passed; either fro the process to the kernel or vice-versa.

1. The three functions "bind, connect, & sendto" pass a socket address structure from the process to the kernel. One of the arguments to these functions is "the pointer to the socket address structure" and another is "the integer size of the structure":

```
struct sockaddr_in serv;
/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

Since both the pointer and the size of the structure are passed to the pointer, the kernel knows exactly how much of data is to be copied from the process into the kernel.



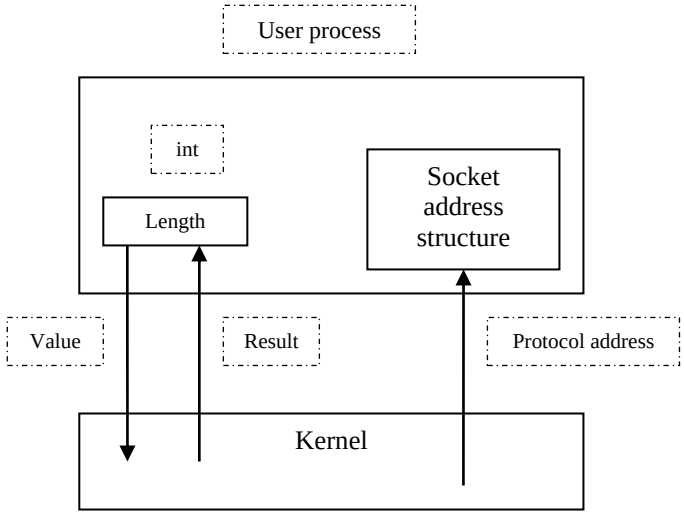
Socket address structure passed from process to kernel

2. The four functions "accept, recvfrom, getsockname, & getpeername" pass a socket address structure from the kernel to the process. The arguments to these functions are "the pointer to the socket address structure" along with "a pointer to an integer containing the size of the structure":

```
struct sockaddr_un cli;        /* Unix domain */
socklen_t   len;

len = sizeof(cli);                    /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

The size changes from an integer to be a pointer to an integer is because the size is both a *value* when the function is called, whereas it is a *result* when the function returns. Such type of argument is called a *value-result* argument. In network programming, the most common example of value-result argument is the length of a returned socket address structure.

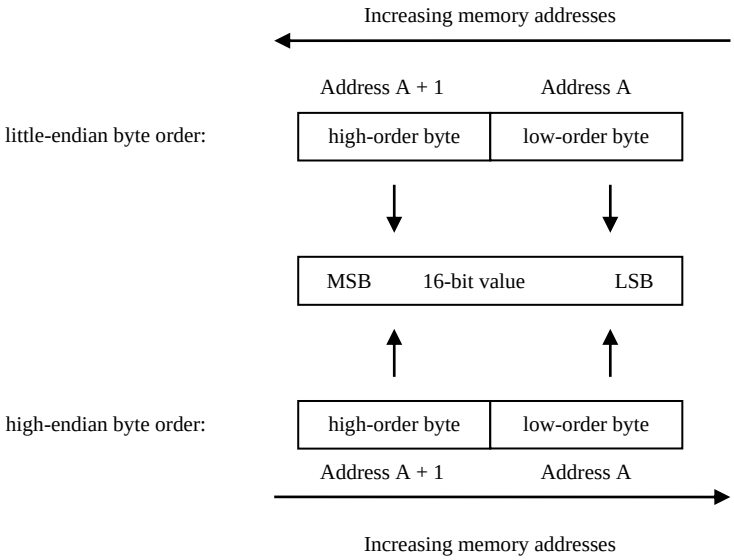
When using value-result arguments for the length of the socket address structure, if the socket address structure is fixed-length, the value returned by the kernel will be of fixed size: 16 for an IPv4 "sockaddr_in" and 28 for IPv6 "sockaddr_in6". But with a variable-length socket address structure such as "sockaddr_un", the value returned can be less than the maximum size of the structure.



Socket address structure passed from kernel to process

Byte Ordering Functions:

There are two ways to store the bytes in memory: with the lower-order byte at the starting address, known as “little-endian byte order”, or with the high-order byte at the starting address, known as “big-endian byte order”.



Little-endian byte order for a 16-bit integer

Here, increasing memory address is shown going from right to left in the top and from left to right in the bottom. The MSB (most significant bit) is shown as the leftmost bit of the 16-bit value and the LSB (least significant bit) as the rightmost value. The terms “little-endian” and “big-endian” indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value. The byte ordering used by a given system is known as the “host byte order”.

It is necessary to deal with the byte ordering differences as network programming because networking protocols must specify a “network byte order”. For example, in a TCP segment, there is a 16-bit port number and a 32-bit IPv4 address. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields are transmitted.

The Internet protocols use big-endian byte ordering for these multibyte integers. An application should be able to store the fields in a socket address structure in host byte order and then convert to and from the network byte order when moving the fields to and from the protocol headers. The following functions are used to convert between host byte order and network byte order:

```
#include <netinet/in.h>
uint16_t htons(unit16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
/* both return value in network byte order */

uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t bitvalue);
/* both returns value in host byte order */
```

In the name of these functions, “h” stands for “host”, “n” stands for “network”, “s” stands for “short” and “l” stands for “long”. When using these functions, it is not necessary to care about the actual values (big-endian or little-endian) for the host byte order and the network byte order. On those systems that have the same byte ordering as the Internet protocols i.e. big-endian, these four functions are usually defined as null macros.

Byte Manipulation Functions:

There are two groups of functions that operate on multibyte fields: without interpreting the data and without assuming that the data is a null-terminated C string. These functions are required when dealing with socket address structures, because manipulation of the fields such as an IP address is required that can contain bytes of 0, but these fields are not C character strings. The functions beginning with “str” (for string) is defined by including <string.h> header to deal with null-terminated C character strings.

The first groups of functions whose name begin with "b" (for byte) are available almost on any system that supports the socket functions. The second groups of functions whose name begin with "mem" (for memory) are from ANSI C standard and are provided with any system that supports ANSI C library.

The Berkeley-derived functions are as follows:

- "bzero" sets the specified number of bytes to 0 in the destination. This function is often used to initialize a socket address structure to 0.
- "bcopy" moves the specified number of bytes from the source to the destination.
- "bcmp" compares two arbitrary bytes strings.

The ANSI C functions are as follows:

- "memset"
- "memcpy"
- "memcmp"

inet_aton, inet_addr and inet_ntoa Functions:

These functions convert Internet address between a dotted-decimal string and its 32-bit network byte ordered binary value.

- "inet_aton" converts the C character string pointed to by "strptr" into its 32-bit binary network byte ordered value, which is stored through the pointer "addrptr".
- "inet_addr" does the same conversion, returning the 32-bit binary network byte ordered value as the return value.
- "inet_aton" is used instead of "inet_addr" in these days.
- "inet_ntoa" converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.

inet_pton and inet_ntop Functions:

These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses. The letters "p" and "n" stand for "presentation" and "numeric" respectively. The presentation format for an address is an ASCII string and the numeric format is the binary value that goes into a socket address structure.

fork and exec Functions:

The *fork* function is used to create a new process in UNIX. This function is called once but it returns twice. *"fork"* returns once in the calling process (parent) with a return value that is the process ID of the newly created process (child). It also returns once in the child, with a return value of 0. Hence the return value tells the process whether it is the parent or the child.

The *fork* returns 0 in the child and it can always obtain the parent's process ID by calling "*getppid",* whereas, a parent can have any number of children, and it is not possible to obtain the process ID of its children. If the parent wants to keep track of the process IDs of its children, it must record the return values from "fork". The parent calls "accept" and then calls "fork". The connected socket is then shared between the parent and child. Finally, it returns "-1" on error.

```
#include <unistd.h>
pid_t fork(void)
```

There are two typical uses of "fork":

1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is common for many network servers.
2. A process wants to execute another program. Since the only way to create a new process is by calling "fork", the process first calls "fork" to make a copy of itself, and then one of the copies (typically the child) calls the "exec" functions to replace itself with the new program. This is common for the shell programming.

"exec" replaces the current process image with the new program file, which normally starts at the "main" function. The process ID does not change. The process that calls "exec" functions is referred as the "calling function" and the newly executed program as the "new program".

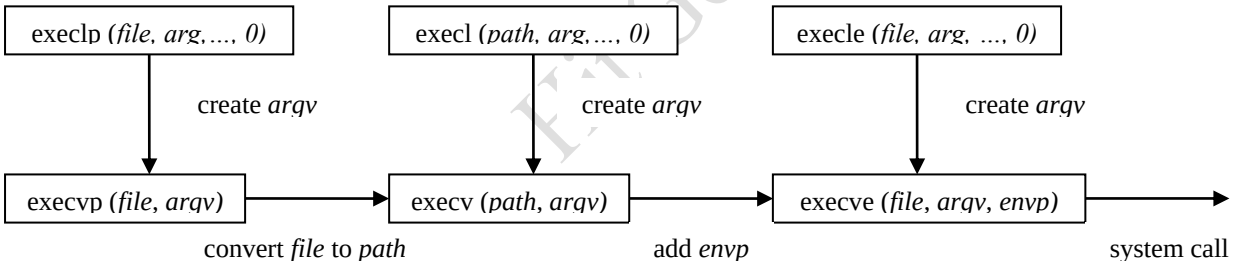
In UNIX, the only way an executable program file on disk can be executed for an existing process is by calling one of the following six "exec" functions:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ...);
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *argv0, ...);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ...);
int execvp(const char *filename, char *const argv[]);
```

The differences in six "exec" functions are:

- whether the program file to be executed is specified by a "filename" or a "pathname"
- whether the arguments to the new program are listed one by one of referenced through an array of pointers
- whether the environment of the calling process is passed to the new program or whether a new environment is specified

Normally, only "execve" is a system call within the kernel and the other five are library functions that call "execve". The relationship among six "exec" functions is shown below:



- The three functions "execlp", "execl" and "execle" specify each argument string as a separate argument to the "exec" function, with a null pointer terminating the variable number of arguments. The three functions "execvp", "execv" and "execve" have "argv" array, containing pointers to the argument strings. The "argv" array must contain a null pointer to specify its end.
- The two functions "execlp" and "execvp" specify a "filename" argument. This is converted into a "pathname" using the current PATH environment variable. The PATH variable is not used if the "filename" argument to these two functions contains a slash (/) anywhere in the string. The four functions "execl", "execv", "execle" and "execve" specify a fully qualified "pathname" argument.
- The four functions "execlp", "execvp", "execl" and "execv" do not specify an explicit environment pointer. Instead, the current value of the external variable "environ" is used for building an environment list that is passed to the new program. The two functions "execle" and "execve" specify an explicit environmentlist. The "envp" array of pointers must be terminated by a null pointer.

```
/*************************************************************
*
*  program  to demonstrate fork() and execl() and system calls
```

```
 *
 ************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main( int argc, char *argv[], char *env[] )
{
   pid_t my_pid, parent_pid, child_pid;
   int status;

/* get and print my pid and my parent's pid. */

   my_pid = getpid();    parent_pid = getppid();
   printf("\n Parent: my pid is %d\n\n", my_pid);
   printf("Parent: my parent's pid is %d\n\n", parent_pid);

/* print error message if fork() fails */
   if((child_pid = fork()) < 0 )
   {
      perror("fork failure");
      exit(1);
   }

/* fork() == 0 for child process */

   if(child_pid == 0)
   {  printf("\nChild: I am a new-born process!\n\n");
      my_pid = getpid();
            parent_pid = getppid();
      printf("Child: my pid is: %d\n\n", my_pid);
      printf("Child: my parent's pid is: %d\n\n", parent_pid);
      printf("Child: I will sleep 3 seconds and then execute - date - command \n\n");

      sleep(3);
      printf("Child: Now, I woke up and am executing date command \n\n");
      execl("/bin/date", "date", 0, 0);
      perror("execl() failure!\n\n");

      printf("This print is after execl() and should not have been executed if execl were successful!
\n\n");

      _exit(1);
   }
```

```
/*
 * parent process
 */
  else
  {
    printf("\nParent: I created a child process.\n\n");
    printf("Parent: my child's pid is: %d\n\n", child_pid);
    system("ps -acefl | grep ercal");  printf("\n \n");
    wait(&status); /* can use wait(NULL) since exit status
                      from child is not used. */
    printf("\n Parent: my child is dead. I am going to leave.\n \n ");
  }

  return 0;
}
```

Unix Domain Protocols

- The Unix domain protocols are not an actual protocol suite, but a way of performing client-server communication on a single host using the same API that is used for clients and servers on different hosts (like sockets or XTI).
  → When the client and server are on the same host, this alternative IPC method is used.
- Two types of sockets are provided in the Unix domain:
  → stream sockets (similar to TCP)
  → datagram sockets (similar to UDP)
- Unix domain sockets are used for these reasons:
  → On many implementations, Unix domain sockets are often twice as fast as a TCP socket, when both peers are on the same host.
    - For example, when an X11 client starts and opens a connection to the X11 server, the client checks the value of the DISPLAY environment variable, which specifies the server's hostname, window, and screen. If the server is on the same host as the client, the client opens a Unix domain stream connection to the server; otherwise the client opens a TCP connection to the server.
  → Unix domain sockets are sued when passing descriptors between processes on the same host.
  → Newer implementations of Unix domain sockets provide the client's credentials (user ID and group ID) to the server, which can provide additional security checking.
- The protocol addresses used to identify clients and servers in the Unix domain are pathnames within the normal file system (as against the IP address, port number combination of an Internet socket). However, these pathnames are not normal Unix files; we cannot read from or write to these files except from a program that has associated the pathname with a Unix domain socket.
- The PF_UNIX (also known as PF_LOCAL) socket family is used to communicate between processes on the same machine efficiently. Unix sockets can be either anonymous (created by socketpair(2)) or associated with a file of type socket. Address family is AF_UNIX (or AF_LOCAL)
  - Posix.1g renames the Unix domain protocals as "local IPC", to remove the dependence of the Unix operating system. The historical constant AF_UNIX becomes AF_LOCAL. Nevertheless, we still use the term "Unix domain" as that has become its de facto name, regardless of the underlying operating system. Also, even with Posix.1g attempting to make these operating system independent, the socket address structure still retains the _un suffix.

Address Structure

- The Unix domain socket address structure is defined by including the <sys/un.h> header.

```
struct sockaddr_un {
        uint8_t         sun_len;
        sa_family_t     sun_family;
        char                    sun_path[104];
}
```

→ sun_len is unsigned integer – length of the structure.
→ sa_family_t is AF_UNIX (or AF_LOCAL)
→ The pathname stored in sun_path array must be null terminated.

-
-
- **// Implementation of Unix Domain Stream Server**
-
- #include <sys/socket.h>
- #include <sys/un.h>
- #include <sys/types.h>
- #include <stdio.h>
-
-
- int main()
- {
- //variable declaration
- int sockfd, clientsockfd,n,addrlen;
- struct sockaddr_un servaddr, clientaddr;
-
- char buffer[256];
-
-
-

```
    addrlen = sizeof(clientaddr);

    // Create socket
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);

    // Delete if the file name already exists
    unlink("/home/hit/NP/mysock");
    //fill zeros in the structure
    bzero(&servaddr, sizeof(servaddr));

    // Fill the structure with necessary information
    servaddr.sun_family = AF_UNIX;
    strcpy(servaddr.sun_path, "/home/hit/NP/mysock");

    // Binding socket
    bind (sockfd, &servaddr, sizeof(servaddr));
    // Listen on the socket
    listen(sockfd, 5);

        //accept a new connection
        clientsockfd = accept(sockfd, &clientaddr, &addrlen);

        //receiving what the client has sent us
        read(clientsockfd, &buffer, sizeof(buffer));
        printf("\n Client sent us : %s \n", buffer);

        // send some reply to the client
        printf("Please enter some message to client: ");

        bzero(buffer,256);
        fgets(buffer,255,stdin);


    write(clientsockfd,&buffer, sizeof(buffer));
    close(clientsockfd);

    close(sockfd);
return 0;
}
//  Implementation of the  Unix Domain Stream Client
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
#include<stdio.h>
int main()
{
    //declare variables here
    int sockfd,n;
    struct sockaddr_un servaddr;
    char buffer[256];
    // Creating socket
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    //fill  the structure with zeros
    bzero(&servaddr, sizeof(servaddr));
```

```
        // Fill the structure with necessary information for later computation
        servaddr.sun_family = AF_UNIX;
        strcpy(servaddr.sun_path, "/home/hit/NP/mysock");


        // Connect to the  server which is listening to the clients
        connect(sockfd, &servaddr, sizeof(servaddr));
        printf("Please enter your message: ");

        bzero(buffer,256);
        fgets(buffer,255,stdin);

        // write some message to the server
        write(sockfd, buffer,sizeof(buffer));

        // get what the server says
        read(sockfd, &buffer, sizeof(buffer));
        printf("\n  Server says : %s \n", buffer);
        close(sockfd);
return 0;
}
```

➢

Socketpair, pipe Functions

➢

- The socketpair function creates two sockets that are then connected together. This function applies to only Unix domain sockets.

```
 # include <sys/socket.h>
 int socketpair (int family, int type,
        int protocol, int sockfd[2]);
```

- The family must be AF_LOCAL(AF_UNIX) and the protocol must be 0. The type, however, can be either SOCK_STREAM or SOCK_DGRAM.
- The two socket descriptors that are created are returned as sockfd[0] and sockfd[1].
- The two created sockets are unnamed; that is, there is no implicit bind involved.
- The result of a socketpair with a type of SOCK_STREAM is called a stream pipe. It is similar to a regular Unix pipe (created by the pipe function), but a stream pipe is full-duplex; that is, both descriptors can be read and written. A regular Unix pipe is half-duplex – one descriptor is for reading and one for writing.
- pipe creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing.

The same functions (select, bind, listen, connect, accept, close, read, write, send, recv, sendto, recvfrom, sendmsg, recvmsg, etc.) are used irrespective of whether the socket is an Internet socket or a Unix socket.

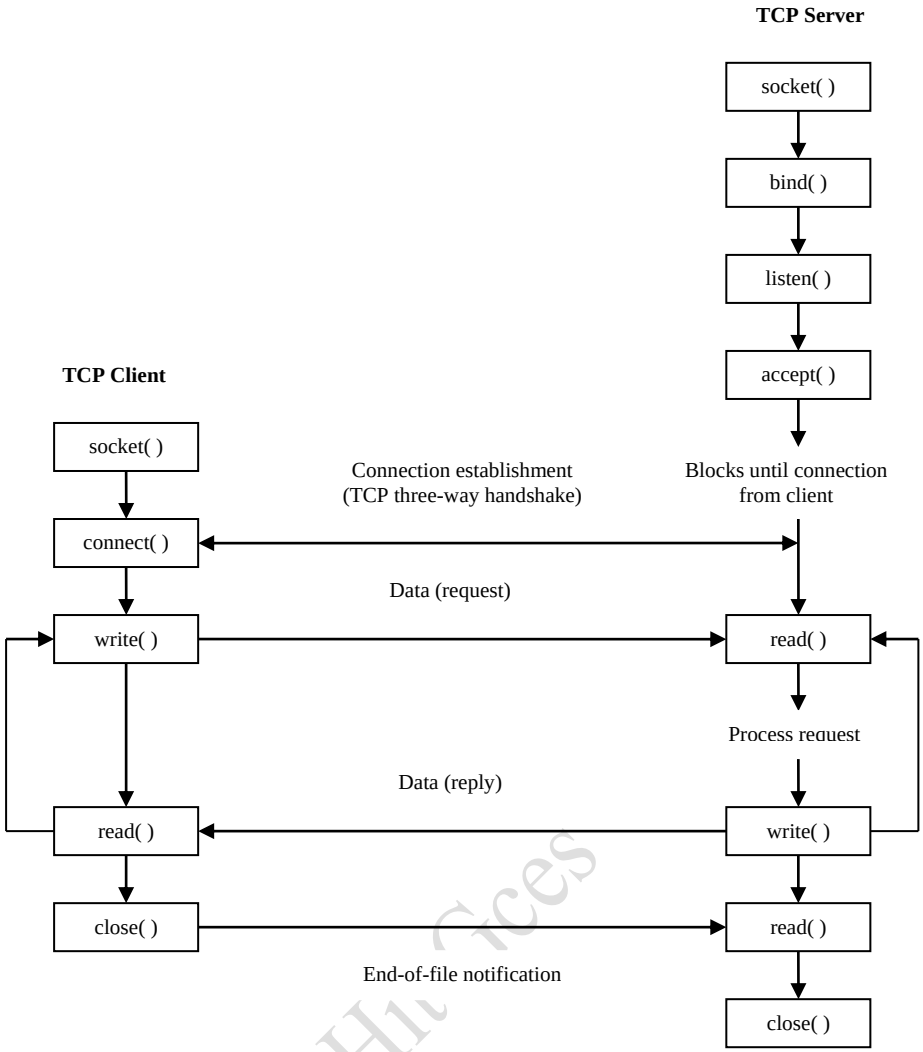
//Socketpair.c

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


main()
{
```

```
        int sockets[2];
        char buff[100];
        // Create a socket pair
        socketpair(AF_UNIX, SOCK_STREAM, 0, sockets);
        printf("\n sockets[0] = %d , sockets[1] = %d\n", sockets[0], sockets[1]);
        //sleep(5);
        if (! fork())
        {               // This is the Child close parent's end
                close(sockets[1]);
                //send message to parent
                write(sockets[0], "Test Message 1", sizeof("Test Message 1"));
                                //read message from parent
                read(sockets[0], buff, sizeof(buff));
                printf("\n I am child.  I got from parent = %s \n", buff);
                //sleep(5);
                //Finally close our end of the socket
                close(sockets[0]);
        }
        else
        {
                // This is the Parent close child's end
                close(sockets[0]);
                                // read message from Child
                read(sockets[1], buff, sizeof(buff));
                printf("\n I am parent.  I got from my child = %s \n", buff);
                                //write message to child
                write(sockets[1], "From Parent", sizeof("From Parent"));
                //sleep(5);
                //Finally close our end of the socket
                close(sockets[1]);
        }
}
```


Elementary TCP Sockets

One of the basic component for writing a complete TCP client and server is the elementary socket function. The figure below shows the interaction between TCP client and server. At first, the server is started, and then sometime later a client is started that connects to the server. It is assumed that the client sends a request to the server, the server processes the request, and the server sends back a reply to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. Finally the server closes its end of the connection and either terminates or waits for a new client connection.



Socket functions for elementary TCP client-server

socket function:

In order to perform I/O, the first thing a process must do is call the *socket* function, specifying required type of communication protocol. The *family* specifies the protocol family, where the constants are shown in the table. Similarly, the constants for socket *type* are also below. Normally, the *protocol* argument to the socket function is set to 0 except for raw sockets.

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

Family	*Description*
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing sockets

Protocol family constants for socket function

Type	*Description*
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_RAW	Raw socket

Type of socket for socket function

Protocol	*Description*

IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol

Protocol of sockets for AF_INET or AF_INET6

In short, this function creates "an end point for communication". The return value from this function is a handle to a socket. This number is passed as a parameter to almost all of the other library calls.

Since the focus of this document is on TCP/IP based sockets, the family parameter should be set to AF_INET. The type parameter can be either SOCK_STREAM (for TCP), or SOCK_DGRAM (for UDP). The protocol field is intended for specifying a specific protocol in case the network model supports different types of stream and datagram models. However, TCP/IP only has one protocol for each, so this field should always be set to 0.

On success, the “socket” function returns a small non-negative integer value known as “socket descriptor” or a “sockfd”.

To create a UDP socket:

```
int s;
s = socket(AF_INET, SOCK_DGRAM, 0);
```

To create a TCP socket:

```
int s;
s = socket(AF_INET, SOCK_STREAM, 0);
```

connect Function:

It is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen) ;
```

Where, “sockfd” is a socket descriptor, which is returned by the socket function. The second and third arguments are a pointer to a socket address structure, and its size. The socket address structures must contain the IP address and port number of the server.

The client does not have to call “bind” before calling “connect”. If required, the kernel will choose both temporary port and the source IP address. In TCP socket, the “connect” function initiates TCP’s three-way handshake. The function returns only when the connection is established or an error occurs. There are several types of possible errors:

1. If the client TCP receives no response to its SYN segment, “ETIMEDOUT” is returned.
2. If the server’s response to the client’s SYN is a reset (RST), it indicates that no process is waiting for connections on the server host at the specified port (or, probably the server process is not running). This is a “hard error” and the error “ECONNREFUSED” is returned to the client as soon as the RST is received.

RST is a type of TCP segment that is sent by TCP when something is wrong. There are three conditions that generate RST state:

- when a SYN arrives for a port that has no listening server
- when TCP wants to abort an existing connection
- when TCP receives a segment for a connection that does not exist

bind Function:

It assigns a local protocol address to a socket. The protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address with a 16-bit TCP or UDP port number. Before sending and receiving data with a socket, it must first be associated with a local source port and a network interface address. The mapping of a socket to a TCP/UDP source port and IP address is called a "binding".

It may be the case where the socket is being used as a server, and thus must be able to listen for client requests on a specific port. It can also be the case that a client program doesn't need a specific source port, since all it's concerned about doing is sending and receiving messages with a specific port on the remote host. Further complications arise when there are more than one network devices on the host running the program. So the question of sending through "which network" must be answered as well. The *bind* function call is used to declare the mapping between the socket, the TCP/UDP source port, and the network interface device.

The prototype for bind is as follows:

```
#include <sys/socket.h>
bind(int sockfd, const struct sockaddr *address, socklen_t addrlen);
```

The first argument is a socket descriptor. The second argument is a protocol-specific address structure and the third is the size of this address structure. With TCP, there are various conditions while calling "bind" such as specifying a port number, IP address, both or neither.

Servers bind their well-known port when they start. If a TCP client or server does not do this, the kernel chooses a temporary port for the socket when either "connect" or "listen" is called. It is normal for a TCP client to let the kernel choose an ephemeral port unless the application requires a reserved port. But since servers are known by their well-known port, it is rare for a TCP server to let the kernel choose an ephemeral port.

A process can "bind" a specific IP address to its socket. The IP address must belong to an interface on the host. For a TCP client, the source IP address is assigned, which will be used for IP datagram sent on to a socket. Whereas for a TCP server, the socket is restricted to receive incoming client connections destined on to that IP address.

Normally, a TCP client does not "bind" an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used. Meanwhile, if a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the client's SYN packet as the server's source IP address. The following table summarizes the values to set for "sin_addr" and "sin_port", or "sin6_addr" and "sin6_port", depending on the desired result.

Process Specifies		**Result**
IP address	**port**	
Wildcard (*)	0	Kernel chooses IP address and port
Wildcard (*)	Non zero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	Non zero	Process specifies IP address and port

If a port number is specified as "0", the kernel chooses an ephemeral port when "bind" is called. But if a wildcard is specified as an IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

In case of IPv4, the wildcard address is specified by the constant "INADDR_ANY", whose value is normally "0". This tells the kernel to choose the IP address.

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* wildcard */
```

Example:

```
struct sockaddr_in sin;
int s;
s = socket(AF_INET, SOCK_DGRAM, 0);
sin.sin_family = AF_INET;
sin.sin_port = htons(9999);
sin.sin_addr.s_addr = INADDR_ANY;

bind(s, (struct sockaddr *)&sin, sizeof(sin));
```

listen Function:

This function is normally called after both the *socket* and *bind* functions, but is necessary to be called before the *accept* function.

```
#include <sys/socket.h>
int listen(int sockfd, int backlog) ;
```

It is called only by a TCP server, which basically performs two actions:

1. When a socket is created by the "socket" function, it is assumed to be an active socket, i.e. a client socket that issues "*connect*". The "*listen*" function converts an unconnected socket into

a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
2. The second argument specifies the maximum number of connections that the kernel should queue for this socket.

This function is normally called after the "socket" and "bind" functions and must be called before calling the "accept" function. In order to understand "backlog" argument, it is necessary to know that for a given listening socket, the kernel maintains two queues:

1. An *incomplete queue,* which contains an entry for each SYN that arrives from a client, for which the server is expecting the completion of TCP three-way handshake.
2. A *complete connection queue,* which contains an entry for each client with whom TCP three-way handshake is completed.

accept Function:

It is called by a TCP server to return the next completed connection from the front of the completed connection queue. If the completed connection queue is empty, the process is put to sleep.

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

The *cliaddr & addrlen* arguments are used to return the protocol address of the connected peer process (client). *addrlen* is a value-result argument i.e. before the call, the integer value is set, referenced by "*addrlen" to the size of the socket address structure pointed to by "cliaddr"; but on return the integer contains the actual number of bytes stored by the kernel in the socket address structure.

If "accept" is successful, its return value is a new descriptor, automatically created by the kernel. The new descriptor refers to the TCP connection with the client. The first argument to "accept" is known as "listening socket" (which is the descriptor created by the socket and then used as the first argument to both "bind" and "listen") and the return value from "accept" is known as "connected socket". It is important to differentiate between these two sockets. Normally, server creates only one listening socket, which exists till the server is on. The kernel creates one connected socket for each client connection that is accepted (i.e. for which the TCP three-way handshake completes). As soon as the server finishes serving its client, the connected socket is closed.

The "accept" function returns three values:
1. Socket descriptor or an error indication
2. The protocol address of the client process (through the "cliaddr" pointer)
3. The size of this address (through the "addrlen" pointer)

If the server does not require to the protocol address of the client, both "cliaddr" and "addrlen" can be set to null pointer.

close Function:

The "close" function is used to close a socket and terminate a TCP connection.

```
#include <unistd.h>
int close(int sockfd);
```

The default action of "close" with a TCP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process: It cannot be used as an argument to "read" and "write". But TCP will try to send any data that is already queued to be sent to other end, after this occurs; the normal TCP connection termination sequence takes place.

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

gethostbyname() searches for information for a host with the hostname specified by the character-string parameter name.

RETURN VALUES
Host entries are represented by the struct hostent structure defined in <netdb.h>:

```
struct hostent {
    char    *h_name;            /* canonical 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 */
};
```

getsockname and getpeername Functions:

These two functions return the local protocol address associated with a socket and the foreign protocol address associated with a socket respectively.

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
```

On success, both functions return 0 or returns -1 on failure. It should be noted that the final argument for both functions is a value-result argument i.e. both functions fill in the socket address structure pointed to by "localaddr" or "peeraddr".

These two functions are required for the following reasons:

- After "connect" successfully returns in a TCP client, "getsockname" returns the local IP address and local port number assigned to the connection by the kernel.
- After calling "bind" with port number "0" (telling the kernel to choose the local port number), "getsockname" returns the assigned local port number.
- "getsockname" can be called to obtain the address family of a socket.
- In a TCP server that "bind"s the wildcard IP address, once a connection is established with a client, the server can call "getsockname" to obtain the local IP address assigned to the connection.
- When a server calls "exec", the only way it can obtain the identity of the client is to call "getpeername". For instance in case of a TCP server, "xinetd" calls "accept" that returns two values: the connected socket descriptor "connfd" (the return value of the function) and an Internet socket address structure (peer's address) of the client. Then "fork" is called and a child of "xinetd" is created. Since the child starts with a copy of the parent's memory image, the socket address structure is available to the child, as the connected socket descriptor. But when the child "exec"s the real server (say the Telnet server), the memory image of the child is replaced with the new program file for the Telnet server (i.e. the socket address structure containing the peer's address is lost). One of the first functions called by the Telnet server is "getpeername" to obtain the IP address and port number of the client.

Data Transfer

The system call pairs (read, write), (recv, send), (recvfrom, sendto), (recvmsg, sendmsg), and (readv, writev) can all be used to transfer data on sockets. The most appropriate call depends on the exact functionality required. "send" and "recv" are typically used with connected stream sockets. They can also be used with datagram sockets if the sender has previously done a "connect" or the receiver does not care who the sender is. "sendto" and "recvfrom" are used with datagram sockets. "sendto" allows one to specify the destination of the datagram, while "revcfrom" returns the name of the remote socket sending the message. "read" and "write" can be used with any connected socket. These two calls may be chosen for efficiency considerations. The remaining data transfer calls can be used for more specialized purposes. "writev" and "readv" make it possible to scatter and gather data to/from separate buffers. "sendmsg" and "recvmsg" allow scatter/gather capability as well as the ability to exchange access rights. The calls "read", "write", "readv", and "writev" take either a socket descriptor or a file descriptor as their first argument; all the rest of the calls require a socket descriptor.

```
count = send(sock, buf, buflen, flags)
int count, sock, buflen, flags; char *buf;

count = recv(sock, buf, buflen, flags)
int count, sock, buflen, flags; char *buf;

count = sendto(sock, buf, buflen, flags, to, tolen)
int count, sock, buflen, flags, tolen;
char *buf;
struct sockaddr *to;

count = recvfrom(sock, buf, buflen, flags, from, fromlen)
int count, sock, buflen, flags, *fromlen;
char *buf;
struct sockaddr *from;
```

For the send calls, count returns the number of bytes accepted by the transport layer, or -1 if some error is detected locally. A positive return count is no indication of the success of the data transfer. For receive calls, count returns the number of bytes actually received, or -1 if some error is detected.

The first parameter for each call is a valid socket descriptor. The parameter “buf” is a pointer to the caller's data buffer. In the “send” calls, “buflen” is the number of bytes being sent; in the “receive” calls, it indicates the size of the data area and the maximum number of bytes the caller is willing to receive. The parameter “to” in the “sendto” call specifies the destination address and “tolen” specifies its length. The parameter from in the “recvfrom” call specifies the source address of the message. “fromlen” is a value/result parameter that initially gives the size of the structure pointed to by from and then is modified on return to indicate the actual length of the address.

The flags parameter, which is usually given zero as an argument, allows several special operations on stream sockets. It is possible to send out-of-band data or "peek" at the incoming message without actually reading it. The flags “MSG_OOB” and “MSG_PEEK” are defined in <sys/socket.h>. Out-of-band data is high priority data (such as an interrupt character) that a user might want to process quickly before all the intervening data on the stream. If out-of-band data were present, a SIGURG signal could be delivered to the user. The actual semantics of out-of band data is determined by the relevant protocol. ISO protocols treat it as “expedited data”, while Internet protocols treat it as “urgent data”.

If any of these (as well as other) system calls are interrupted by a signal, such as “SIGALRM” or “SIGIO”, the call will return -1 and the variable “errno” will be set to EINTR 2. The system call will be automatically restarted. It may be advisable to reset “errno” to zero.

Note: Error numbers are defined in the file <errno.h>

Synchronization:

By default, all the reading and writing calls are blocking: Read calls do not return until at least one byte of data is available for reading and write calls block until there is enough buffer space to accept some or all of the data being sent. Some applications need to service several network connections simultaneously, performing operations on connections as they become enabled. There are three techniques available to support such applications: non blocking sockets, asynchronous notifications, and the “select” system call. The “select” system call is by far the most commonly used; non-blocking sockets are less common, and asynchronous notifications are rarely used.

// simple tcp server code

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

#define MYPORT 5555;
main()
{
        int sockfd, len;
```

```
        struct sockaddr_in myaddr, youraddr;
        char buffer[50] = "Congratulations, connected !";

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        myaddr.sin_family = AF_INET;
        myaddr.sin_port = htons(MYPORT);
        myaddr.sin_addr.s_addr = htons(INADDR_ANY);

        bind (sockfd, &myaddr, sizeof(myaddr));

        listen (sockfd, 4);

        accept (sockfd, &youraddr, sizeof(youraddr));
        len = strlen(buffer);

        send(sockfd, &buffer, sizeof(buffer), 0);

        close(sockfd);
}
```

// simple tcp client

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

#define YOURPORT 5555;
#define YOURIP "192.168.0.2";

main()
{
        int sockfd, len;
        struct sockaddr_in  youraddr;
        char buffer[50];

        sockfd = socket(PF_INET, SOCK_STREAM, 0);

        youraddr.sin_family = AF_INET;
        youraddr.sin_port = htons(5555);
        youraddr.sin_addr.s_addr = inet_addr("192.168.0.2");


        connect (sockfd, &youraddr, sizeof(youraddr));

        recv(sockfd, &buffer, len, 0);

        printf("Server sent %s \n", buffer);

        close(sockfd);
}
```


Concurrent Servers

- The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.
- When a connection is established, accept returns, the server calls fork, and then the child process services the client(on the connected socket) and the parent process waits for another connection(in the listening socket).
- The parent closes the connected socket since the child handles this new client. The child closes the listening socket since the parent is listening anyway.

Handling Zombies

- The purpose of the zombie state is to maintain information about the child for the parent to fetch at some later time. This information includes the process ID of the child, its termination status, and information on the resource utilisation of the child (CPU time, memory, etc.)
- If a process terminates, and that process has children in the zombie state, the parent process ID of all zombie children is set to 1 (the init process), which will inherit the children and clean them.
- But in our concurrent server, since the parent is still alive, the children remain as zombies.
- Whenever we fork children, we must wait for them to prevent them from becoming zombies.
- To do this, we establish a signal handler to catch SIGCHLD and within the handler, we call wait.

```
/*************************************************************
 *
 *   program  to demonstrate concurrent tcp server
 *
 *************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

main()
{
   int                 sockfd, newsockfd ; /* Socket descriptors */
   int                 clilen;
   struct sockaddr_in  cli_addr, serv_addr;

   int i;
   char buf[100];
   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
         printf("Cannot create socket\n");
         exit(0);
   }


   serv_addr.sin_family                = AF_INET;
   serv_addr.sin_addr.s_addr = INADDR_ANY;
   serv_addr.sin_port            = htons(6000);

   if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
         printf("Unable to bind local address\n");
         exit(0);
   }

   listen(sockfd, 5);
   while (1) {

         clilen = sizeof(cli_addr);
         newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen) ;

         if (newsockfd < 0) {
                  printf("Accept error\n");
                  exit(0);
         }

         if (fork() == 0) {

                  /* This child process will now communicate with the   client through the send() and
recv() system calls.     */
                  close(sockfd);  /* Close the old socket since all  communications will be through  the
new socket.     */
```

```
                strcpy(buf,"Message from server");
                send(newsockfd, buf, strlen(buf) + 1, 0);

                /* We again initialize the buffer, and receive a  message from the client.   */
                for(i=0; i < 100; i++) buf[i] = '\0';
                recv(newsockfd, buf, 100, 0);
                printf("%s\n", buf);

                close(newsockfd);
                exit(0);
        }

        close(newsockfd);
   }
}



/*************************************************************
 *
 *   program  to demonstrate concurrent tcp client
 *
 *************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

main()
{
   int                  sockfd ;
   struct sockaddr_in  serv_addr;

   int i;
   char buf[100];

   /* Opening a socket is exactly similar to the server process */
   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("Unable to create socket\n");
        exit(0);
   }
   serv_addr.sin_family                = AF_INET;
   serv_addr.sin_addr.s_addr = inet_addr("192.168.0.121");
   serv_addr.sin_port            = htons(6000);


   if ((connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))) < 0) {
        printf("Unable to connect to server\n");
        exit(0);
   }

   for(i=0; i < 100; i++) buf[i] = '\0';
   recv(sockfd, buf, 100, 0);
   printf("%s\n", buf);


   strcpy(buf,"Message from client");
   send(sockfd, buf, strlen(buf) + 1, 0);

   close(sockfd);
}
```

File Sharing and passing file descriptor

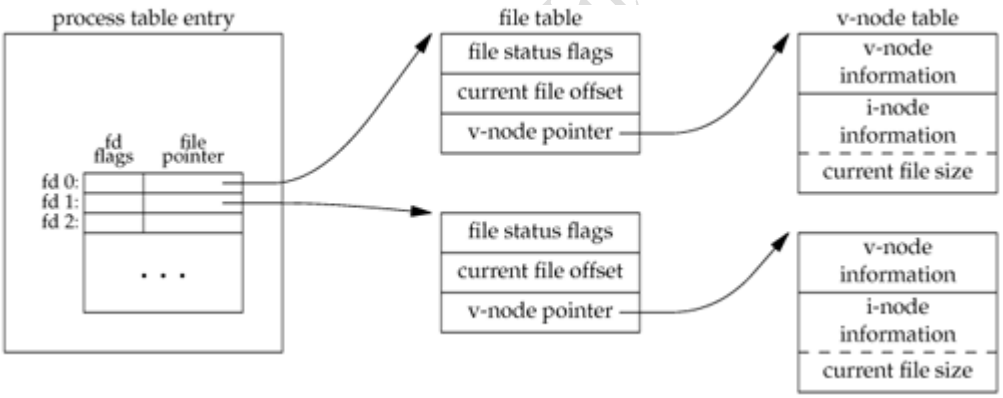
The UNIX System supports the sharing of open files among different processes. To do this, we'll examine the data structures used by the kernel for all I/O.

The kernel uses three data structures to represent an open file, and the relationships among them determine the effect one process has on another with regard to file sharing.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
    a. The file descriptor flags
    b. A pointer to a file table entry
2. The kernel maintains a file table for all open files. Each file table entry contains
    a. The file status flags for the file, such as read, write, append, sync, and nonblocking
    b. The current file offset
    c. A pointer to the v-node table entry for the file
3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, and so on.

Linux has no v-node. Instead, a generic i-node structure is used. Although the implementations differ, the v-node is conceptually the same as a generic i-node. Both point to an i-node structure specific to the file system.

Figure Kernel data structures for open files

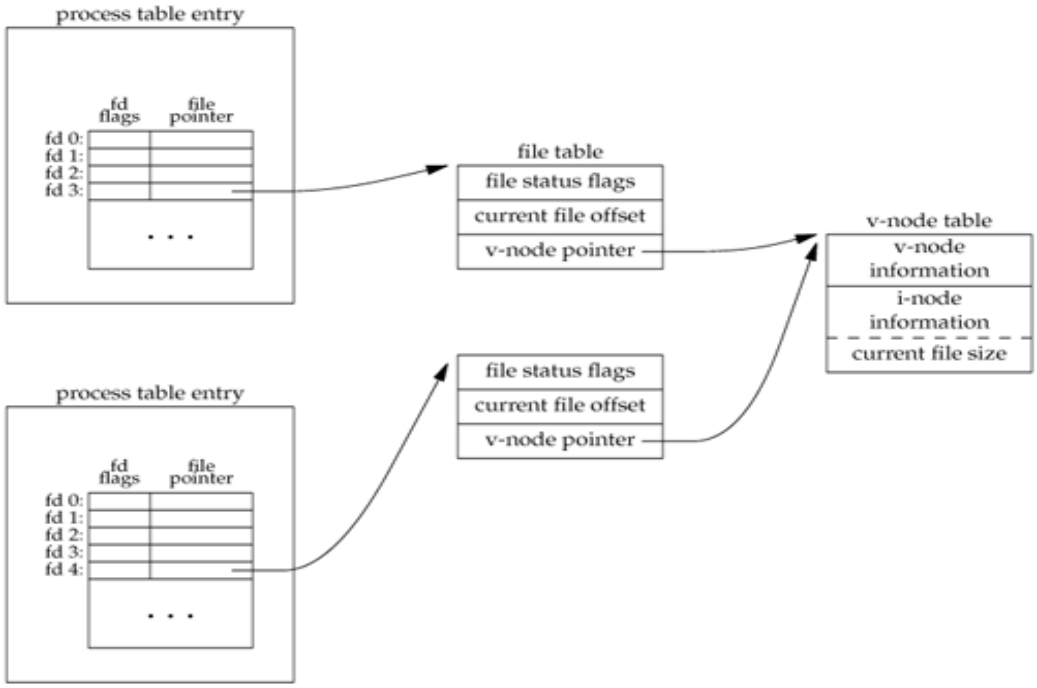


Passing File Descriptors

The ability to pass an open file descriptor between processes is powerful. It can lead to different ways of designing clientserver applications. It allows one process (typically a server) to do everything that is required to open a file (involving such cdetails as translating a network name to a network address, dialing a modem, negotiating locks for the file, etc.) and simply pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

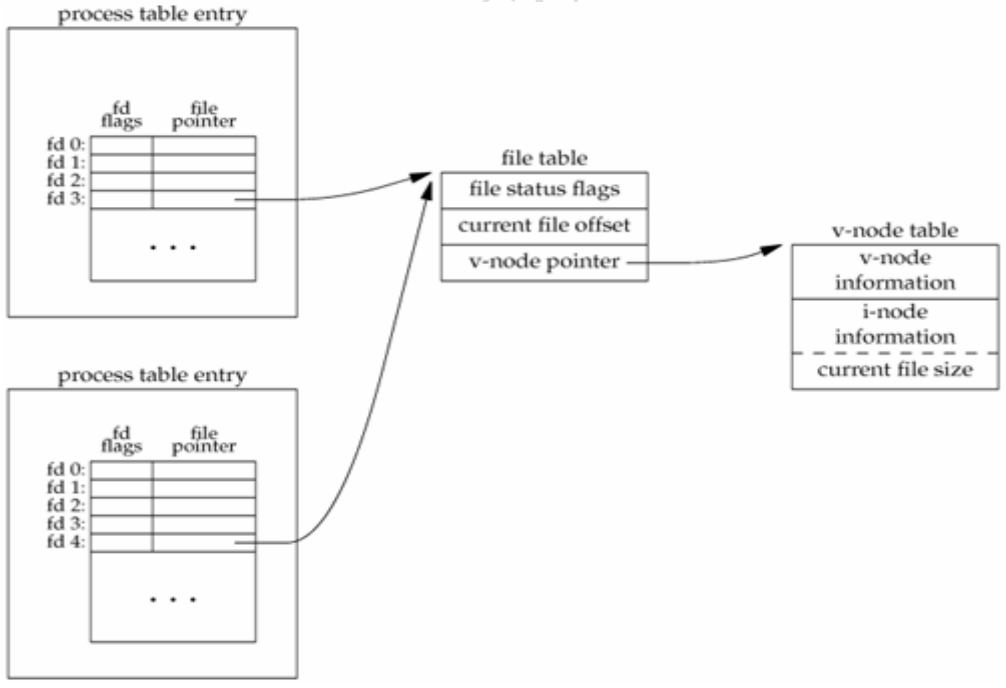
We must be more specific about what we mean by "passing an open file descriptor" from one process to another. Figure below showed two processes that have opened the same file. Although they share the same v-node, each process has its own file table entry.

Figure Two independent processes with the same file open



When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Figure shows the desired arrangement.

Figure Passing an open file from the top process to the bottom process



Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.)

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by

the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

UNIX I/O Models

There are five basic types of I/O models, which are available in UNIX:

- blocking I/O
- nonblocking I/O
- I/O multiplexing (select and poll)
- Signal driven I/O (SIGIO)
- Asynchronous I/O (the POSIX "aio_" functions)

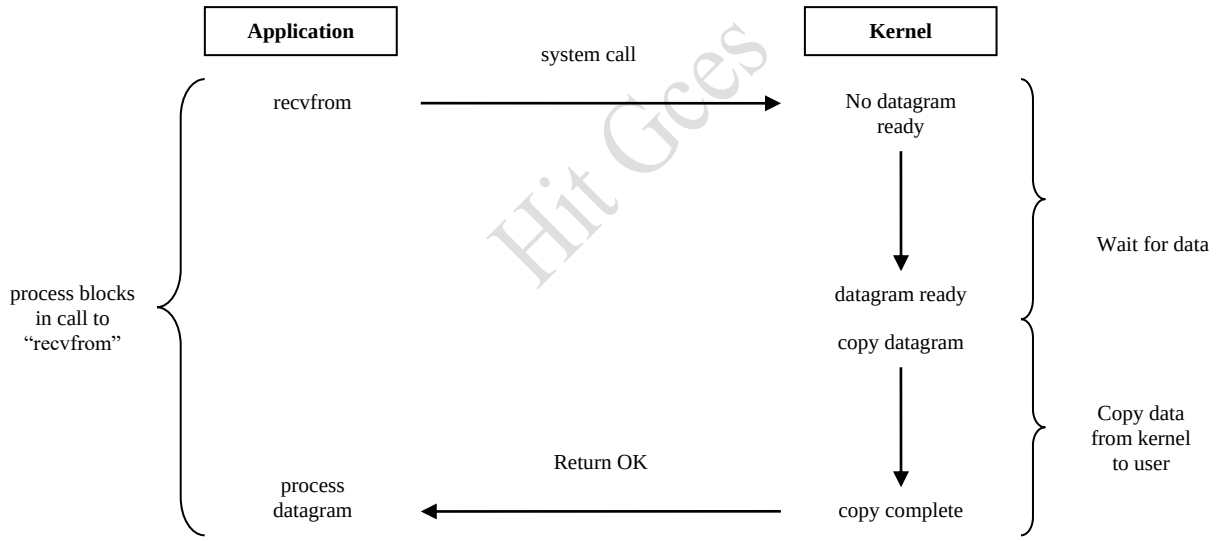
Normally, there are two distinct phases for an input operation:

1. Waiting for the data to be ready
2. Copying the data from the kernel to the process

For an input operation on a socket, the first step involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel. The second step is copying the data from the kernel's buffer into our application buffer.

Blocking I/O Model:

By default, all sockets are blocking. It is further explained by taking a reference of UDP in the figure below:

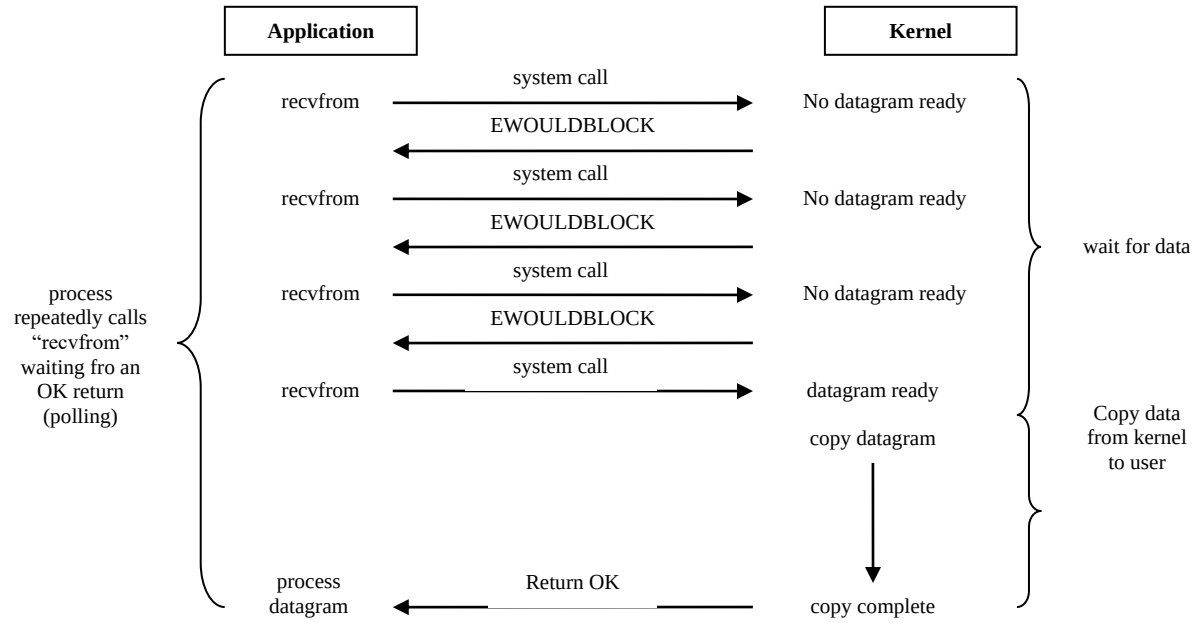


Blocking I/O Model

Here, the process calls "recvfrom" and the system call do not return until the datagram arrives and is copied into the buffer or an error occurs. The process is said to be blocked the entire time from when it calls "recvfrom" until it returns. When "recvfrom" returns successfully, an application processes the datagram.

Nonblocking I/O Model:

In nonblocking I/O model, the kernel is asked to return an error instead of putting the process to sleep in case when the requested I/O operation cannot be completed without putting the process to sleep. In the following figure, the first three "recvfrom" call has no data to return. Hence the kernel immediately returns an error of "EWOULDBLOCK" instead. In the fourth time, when "recvfrom" is called, a datagram is ready. It is copied into an application buffer and "recvfrom" returns successfully. Then the data is processed.



Nonblocking I/O model

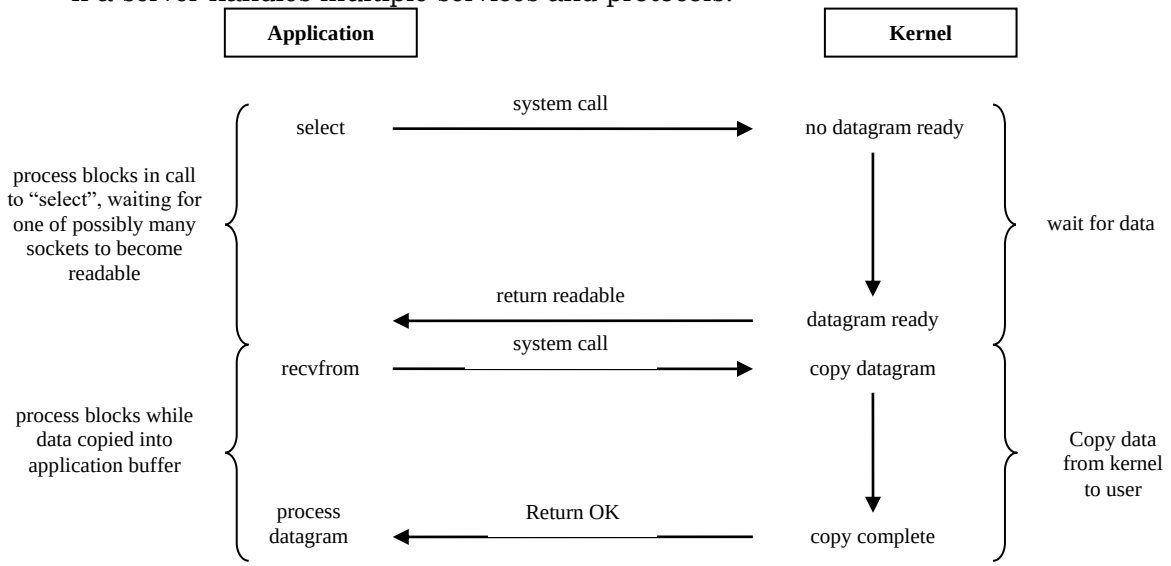
Here, when an application is in a loop calling "recvfrom" on a nonblocking descriptor, it is calling "polling". The application is continually polling the kernel to see if some operation is ready. This is a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O Multiplexing:

Let it be assumed that the TCP server correctly sends a FIN to the client, but if the client process is blocked reading from standard input, it will never see the EOF until it read from the socket. Hence it is necessary to make it capable to tell the kernel what is required to be notified if one or more I/O conditions are ready. This capability is known as "I/O multiplexing" and is provided by the "select" and "poll" functions.

I/O multiplexing is typically used in networking applications in the following cases:

- When a client is handing multiple descriptor (normally interactive inputs and a network socket)
- When a client is handling multiple sockets at the same time, such as in web client.
- If a TCP server handles both a listening socket and its connected sockets.
- If a server handles both TCP and UDP.
- If a server handles multiple services and protocols.



I/O multiplexing model

Here, “select” is called that waits for the datagram socket to be readable. When “select” returns that the socket is readable, then “recvfrom” is called to copy from the datagram into an application buffer.

Select Function

“select” Function: allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occur, or when a specified amount of time has passed.

```
#include <sys/select.h>
#include <sys/time.h>
int select(itn maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval
*timeout);
```

Description of “select” function is started with its final argument, which tells the kernel how long to wait for one of the specified descriptors to become ready. A “timeval” structure specifies the number of seconds and microseconds.

```
struct timeval {
  long tv_sec;   /* seconds */
  long tv_usec;  /* microseconds */
}
```

There are three possibilities:
1. Wait forever – return only when one of the specified descriptors is ready for I/O. For this, “timeout” argument is specified as a null pointer.
2. Wait up to a fixed amount of time – return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the “timeval” structure pointed by the “timeout” argument.
3. Do not wait at all – return immediately after checking the descriptors. This is called “polling”. To specify this, “timeout” argument must point to a “timeval” structure and the timer value must be 0.

Select Function is very useful. When you have situation: that you are a server and you want to listen for incoming connections as well as keep reading from the connections you already have.
No problem, you say, just an accept() and a couple of recv()s. Not so fast, buster! What if you’re blocking on an accept() call? How are you going to recv() data at the same time? "Use non-blocking sockets!" No way! You don’t want to be a CPU hog. What, then?
select() gives you the power to monitor several sockets at the same time. It’ll tell you which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions, if you really want to know that. Without any further ado, I’ll offer the synopsis of select():

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

The function monitors "sets" of file descriptors; in particular readfds, writefds, and exceptfds. If you want to see if you can read from standard input and some socket descriptor, sockfd, just add the file descriptors 0 and sockfd to the set readfds. The parameter numfds should be set to the values of the highest file descriptor plus one. In this example, it should be set to sockfd+1, since it is assuredly higher than standard input (0). When select() returns, readfds will be modified to reflect which of the file descriptors you selected which is ready for reading. You can test them with the macro FD_ISSET(), below
Before progressing much further, I’ll talk about how to manipulate these sets. Each set is of the type fd_set. The following macros operate on this type:
- FD_ZERO(fd_set *set) – clears a file descriptor set
- FD_SET(int fd, fd_set *set) – adds fd to the set
- FD_CLR(int fd, fd_set *set) – removes fd from the set
- FD_ISSET(int fd, fd_set *set) – tests to see if fd is in the set

Finally, what is this weirded out struct timeval? Well, sometimes you don't want to wait forever for someone to send you some data. Maybe every 96 seconds you want to print "Still Going..." to the terminal even though nothing has happened. This time structure allows you to specify a timeout period. If the time is exceeded and select() still hasn't found any ready file descriptors, it'll return so you can continue processing. The struct timeval has the follow fields:

```
struct timeval {
int tv_sec; // seconds
int tv_usec; // microseconds
};
```

Just set tv_sec to the number of seconds to wait, and set tv_usec to the number of microseconds to wait. Yes, that's microseconds, not milliseconds. There are 1,000 microseconds in a millisecond, and 1,000 milliseconds in a second. Thus, there are 1,000,000 microseconds in a second. Why is it "usec"? The "u" is supposed to look like the Greek letter (Mu) that we use for "micro". Also, when the function returns, timeout might be updated to show the time still remaining. This depends on what flavor of Unix you're running.

Yay! We have a microsecond resolution timer! Well, don't count on it. Standard Unix timeslice is around 100 milliseconds, so you might have to wait that long no matter how small you set your struct timeval. Other things of interest: If you set the fields in your struct timeval to 0, select() will timeout immediately, effectively polling all the file descriptors in your sets. If you set the parameter timeout to NULL, it will never timeout, and will wait until the first file descriptor is ready. Finally, if you don't care about waiting for a certain set, you can just set it to NULL in the call to select().

The following code waits 2.5 seconds for something to appear on standard input:

```
/*
** select.c - a select() demo
*/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define STDIN 0 // file descriptor for standard input
int main(void)
{
struct timeval tv;
fd_set readfds;
tv.tv_sec = 2;
tv.tv_usec = 500000;
FD_ZERO(&readfds);
FD_SET(STDIN, &readfds);
// don't care about writefds and exceptfds:
select(STDIN+1, &readfds, NULL, NULL, &tv);
if (FD_ISSET(STDIN, &readfds))
printf("A key was pressed!\n");
else
printf("Timed out.\n");
return 0;
}
```


poll() function

Test for events on multiple sockets simultaneously

Prototypes

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

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Description

This function is very similar to select() in that they both watch sets of file descriptors for events, such as incoming data ready to recv(), socket ready to send() data to, out-of-band data ready to recv(), errors, etc.

The basic idea is that you pass an array of nfds struct pollfds in ufds, along with a timeout in milliseconds (1000 milliseconds in a second.) The timeout can be negative if you want to wait forever. If no event happen on any of the socket descriptors by the timeout, poll() will return.

Each element in the array of struct pollfds represents one socket descriptor, and contains the following fields:

```
struct pollfd {
   int fd;        // the socket descriptor
   short events;  // bitmap of events we're interested in
   short revents; // when poll() returns, bitmap of events that occurred

        };
```

Before calling poll(), load fd with the socket descriptor (if you set fd to a negative number, this struct pollfd is ignored and its revents field is set to zero) and then construct the events field by bitwise-ORing the following macros:

POLLIN	Alert me when data is ready to recv() on this socket.
POLLOUT	Alert me when I can send() data to this socket without blocking.
POLLPRI	Alert me when out-of-band data is ready to recv() on this socket.

Once the poll() call returns, the revents field will be constructed as a bitwise-OR of the above fields, telling you which descriptors actually have had that event occur. Additionally, these other fields might be present:

POLLERR	An error has occurred on this socket.
POLLHUP	The remote side of the connection hung up.
POLLNVAL	Something was wrong with the socket descriptor fd--maybe it's uninitialized?

Return Value

Returns the number of elements in the ufds array that have had event occur on them; this can be zero if the timeout occurred. Also returns -1 on error (and errno will be set accordingly.)

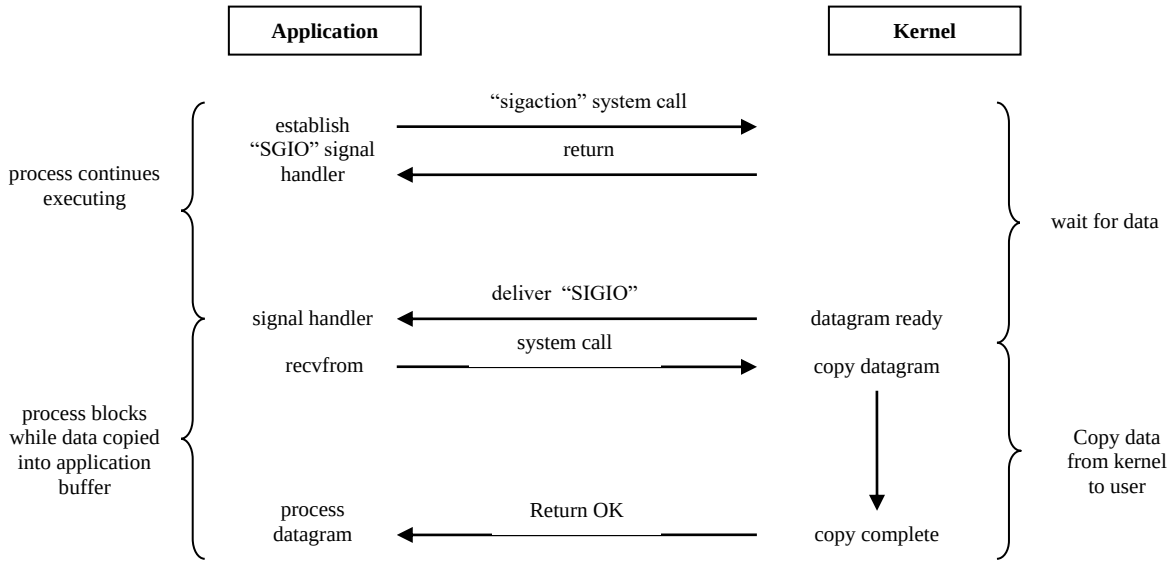
Example

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

s1 = socket(PF_INET, SOCK_STREAM, 0);
```

```
s2 = socket(PF_INET, SOCK_STREAM, 0);

// pretend we've connected both to a server at this point
//connect(s1, ...)...
//connect(s2, ...)...

// set up the array of file descriptors.
//
// in this example, we want to know when there's normal or out-of-band
// data ready to be recv()'d...

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI; // check for normal or out-of-band

ufds[1] = s2;
ufds[1].events = POLLIN; // check for just normal data

// wait for events on the sockets, 3.5 second timeout
rv = poll(ufds, 2, 3500);

if (rv == -1) {
    perror("poll"); // error occurred in poll()
} else if (rv == 0) {
    printf("Timeout occurred!  No data after 3.5 seconds.\n");
} else {
    // check for events on s1:
    if (ufds[0].revents & POLLIN) {
        recv(s1, buf1, sizeof(buf1), 0); // receive normal data
    }
    if (ufds[0].revents & POLLPRI) {
        recv(s1, buf1, sizeof(buf1), MSG_OOB); // out-of-band data
    }

    // check for events on s2:
    if (ufds[1].revents & POLLIN) {
        recv(s1, buf2, sizeof(buf2), 0);
    }
}
```


Signal-driven I/O Model:

Here, the kernel is asked to notify with the "SIGIO" signal when the descriptor is ready.



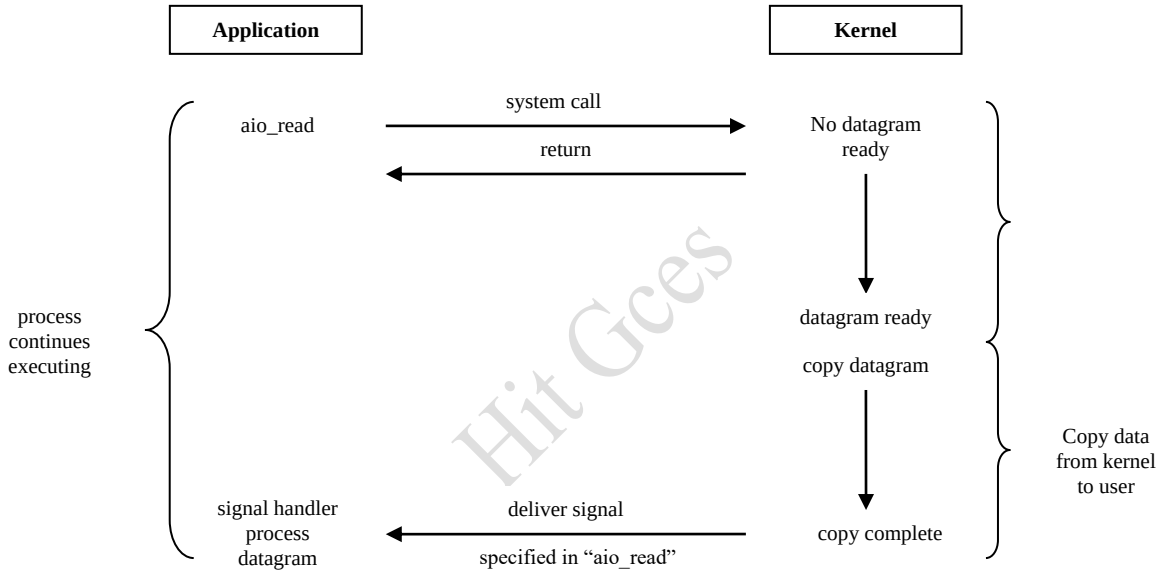
Signal-driven I/O Model

At first, the socket is enabled for signal-driven I/O and a signal handler is installed using the "sigaction" system call. There is an immediate return from "sigaction" and the process continues; i.e. it is not blocked. When the datagram is ready to be read, the "SIGIO" signal is generated for our process. Either it is possible to read the datagram from the signal handler by calling "recvfrom" and notify the main loop that the data is ready to be processed, or notify the main loop and let it read the datagram.

No matter how the signal is handled, the advantage to this model is that there is in no blocking while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler (that either the data is ready to process or the datagram is ready to be read).

Asynchronous I/O Model:

Here, the kernel is asked to start the operation and notify the application when entire operation is completed. The prime difference between asynchronous I/O model and signal-driven model is that, in the later model, the kernel tells when an I/O operation can be initiated, whereas in the former one, the kernel tells when an I/O operation is completed.



Asynchronous I/O Model

Here, "aio_read" is called and the kernel descriptor, buffer pointer, buffer size, file offset are passed and the completion of an entire operation is notified. This system call returns immediately and the process in not blocked while waiting for the I/O to complete.

- A "synchronous I/O operation" causes the requesting process to be blocked until that I/O operation completes.
- An "asynchronous I/O operation" does not cause the requesting process to be blocked.

The first four I/O models – blocking, nonblocking, I/O multiplexing and signal driven I/O are all synchronous because the actual I/O operation "recvfrom" blocks the process. Only the asynchronous I/O model matches the asynchronous I/O definition.

Socket Options

The system calls “getsockopt” and “setsockopt” can be used to set and inspect special options associated with sockets. These might be general options for all sockets or implementation-specific options. Sample options taken from <sys/sockets.h> are SO_DEBUG and SO_REUSEADDR (allow the reuse of local addresses)

“fcntl” and “ioctl” are system calls that make it possible to control files, sockets, and devices in a variety of ways.

```
status = fcntl(sock, command, argument)
int status, sock, command, argument;

status = ioctl(sock, request, buffer)
int status, sock, request;
char *buffer;
```

The command and argument parameters for “fcntl” can be supplied with manifest constants found in “fcntl.h”. The manifest constants for “ioctl”'s request parameter are located in <sys/ioctl.h>. This parameter also specifies how the buffer argument of the call is to be used. Either one of these system calls can be used to enable asynchronous notifications on a socket. Whenever data arrives on such a socket a SIGIO signal will be delivered to the process. The process should have already declared a signal handler for this signal (see Section 6.1). This signal handler can then read the data from the socket. Program execution continues at the point of interruption.

The calling sequence

```
fcntl(sock, F_SETOWN, getpid());
fcntl(sock, F_SETFL, FASYNC);
```

enables asynchronous i/o on the given socket. The first call is necessary to specify the process to be signalled. The second call sets the descriptor status flags to enable SIGIO.

The same calling sequence can be used to make a socket non-blocking; the only change is that the flag FNDELAY is used in the second call instead of FASYNC. In this case, if a read or write operation on the socket would normally block, -1 is returned and the external system variable errno is set to EWOULDBLOCK. This error number can be checked and appropriate action taken. There are various ways to get and set the options that affect a socket:

- the “getsockopt” and “setsockopt” functions
- the “fcntl” options
- the “ioctl” functions

“getsockopt” and “setsockopt” functions:

These two functions apply only to sockets:

```
#include <sys/socket.h>
int getsockopt (int sockfd, int level, int optname, void *optval, socklen_t
*optlen);
int getsockopt (int sockfd, int level, int optname, const void *optval, socklen_t
optlen);
```

Here, “sockfd” refers to an open socket descriptor, “level” specifies the code in the system that interprets the option: the general socket code or some protocol-specific code, “optval” is a pointer to a variable, from which the new value of the option is fetched by “setsockopt”, or into which the current values of the option is stored by “getsockopt” and a value-result for “getsockopt”.

Generic Socket Options

SO_BROADCAST Socket Option:

This option enables or disables the ability of the process to send broadcast messages. Broadcasting is only supported for datagram sockets and only on networks that support the concept of a broadcast message, such as Ethernet, token ring, etc. It is not possible to broadcast on a point-to-point link or any connection-based transport protocol (TCP). An application must set this socket option before sending a broadcast datagram.

SO_DEBUG Socket Option:

This option is supported only by TCP. When enabled for a TCP socket, the kernel keeps track of detailed information about all the packets sent or received by TCP for the socket.

SO_DONTROUTE Socket Option:

This option specifies that outgoing packets are to bypass the normal routing mechanisms of the underlying protocol. For instance in IPv4, the packet is directed to the appropriate local interface, as specified by the network and subnet portions of the destination address. If the local interface cannot be determined from the destination address, "ENETUNREACH" is returned.

SO_ERROR Socket Option:

When an error occurs on a socket, the protocol module in a Berkeley-derived kernels sets a variable name "so_error" for that socket to one of the standard UNIX values. This is called "pending error" for the socket. The process can be immediately notified of the error in one of the following ways:

- If the process is blocked in a call to "select" on the socket for either readability or writability, "select" returns with either or both conditions set.
- If the process is using signal-driven I/O, the SIGIO signal is generated for either the process or the process group.

SO_KEEPALIVE Socket Option:

When the keep-alive option is set for a TCP socket and no data has been exchanged across the socket in either for two hours, TCP automatically sends a "keep-alive probe" to the peer. It is a TCP segment to which the peer must respond in either of the following ways:

- The peer responds with the expected ACK. The application is not notified. TCP will then send another probe after two hours of inactivity.
- The peer responds with an RST, telling the local TCP that the peer host has crashed or rebooted. The socket's pending error is set to "ECONNRESET" and the socket is closed.
- There is no response from the peer to the keep-alive probe. Berkeley-derived TCPs send 8 additional probes, 75 seconds apart, trying to obtain a response. TCP will give up if there is no response within 11minutes and 15 seconds after sending the first probe.
  If there is no response at all to TCP's keep-alive probes, the socket's pending error is set to "ETIMEDOUT" and the socket is closed. But if the socket receives an ICMP error, the corresponding error is returned instead. A common ICMP error in this scenario is "host unreachable" and the pending error is set to "EHOSTUNREACH". This can occur either due to network failure or the remote host has crashed.

SO_LINGER Socket Option:

This option specifies how the "close" function operates for a connection-oriented protocol. By default, "close" returns immediately; but if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer. This can be accomplished by "SO_LINGER" socket option that requires the following structure to be passed between the user process and the kernel. It is defined by including "<sys/socket.h>".

```
struct linger {
      int l_onoff; /* 0=off, nonzero=on */
      int l_linger; /* linger time in seconds */
}
```

Calling "setsockopt" leads to one of the following three scenarios, depending on the values of the two structure members:

1. If "l_onff" is 0, the option is turned off and "close" returns immediately.
2. If "l_onff" is nonzero, and "l_linger" is zero, TCP aborts the connection when it is closed; i.e. TCP discards any data still remaining in the socket send buffer and sends RST to the peer, not the normal 4-packet connection termination sequence.
3. If "l_onoff" is nonzero and "l_linger" is nonzero, the kernel will "linger" when the socket is closed; i.e. if there is any data still remaining in the socket send buffer, the process is put to sleep until either all the data is sent and acknowledged by the peer TCP or the linger time expires.

level	optname	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	Permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	Enable debug tracing	•	int
	SO_DONTROUTE	•	•	Bypass routing table lookup	•	int
	SO_ERROR	•		Get pending error and clear		int
	SO_KEEPALIVE	•	•	Periodically test if connection still alive	•	int
	SO_LINGER	•	•	Linger on close if data to send		linger{}
	SO_OOBINLINE	•	•	Leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	Receive buffer size		int
	SO_SNDBUF	•	•	Send buffer size		int
	SO_RCVLOWAT	•	•	Receive buffer low-water mark		int
	SO_SNDLOWAT	•	•	Send buffer low-water mark		int
	SO_RCVTIMEO	•	•	Receive timeout		timeval{}
	SO_SNDTIMEO	•	•	Send timeout		timeval{}
	SO_REUSEADDR	•	•	Allow local address reuse	•	int
	SO_REUSEPORT	•	•	Allow local port reuse	•	int
	SO_TYPE	•		Get socket type		int
	SO_USELOOPBACK	•	•	Routing socket gets copy of what it sends	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options		(see text)
	IP_RECVDSTADDR	•	•	Return destination IP address	•	int
	IP_RECVIF	•	•	Return received interface index	•	int
	IP_TOS	•	•	Type-of-service and precedence		int
	IP_TTL	•	•	TTL		int
	IP_MULTICAST_IF	•	•	Specify outgoing interface		in_addr{}
	IP_MULTICAST_TTL	•	•	Specify outgoing TTL		u_char
	IP_MULTICAST_LOOP	•	•	Specify loopback		u_char
	IP_{ADD,DROP}_MEMBERSHIP		•	Join or leave multicast group		ip_mreq{}
	IP_{BLOCK,UNBLOCK}_SOURCE		•	Block or unblock multicast source		ip_mreq_source{}
	IP_{ADD,DROP}_SOURCE_MEMBERSHIP		•	Join or leave source-specific multicast		ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Specify ICMPv6 message types to pass		icmp6_filter{}
IPPROTO_IPV6	IPV6_CHECKSUM	•	•	Offset of checksum field for raw sockets		int
	IPV6_DONTFRAG	•	•	Drop instead of fragment large packets	•	int
	IPV6_NEXTHOP	•	•	Specify next-hop address		sockaddr_in6{}
	IPV6_PATHMTU	•		Retrieve current path MTU		ip6_mtuinfo{}
	IPV6_RECVDSTOPTS	•	•	Receive destination options	•	int
	IPV6_RECVHOPLIMIT	•	•	Receive unicast hop limit	•	int
	IPV6_RECVHOPOPTS	•	•	Receive hop-by-hop options	•	int
	IPV6_RECVPATHMTU	•	•	Receive path MTU	•	int
	IPV6_RECVPKTINFO	•	•	Receive packet information	•	int
	IPV6_RECVRTHDR	•	•	Receive source route	•	int
	IPV6_RECVTCLASS	•	•	Receive traffic class	•	int
	IPV6_UNICAST_HOPS	•	•	Default unicast hop limit		int
	IPV6_USE_MIN_MTU	•	•	Use minimum MTU	•	int
	IPV6_V6ONLY	•	•	Disable v4 compatibility	•	int
	IPV6_XXX	•	•	Sticky ancillary data		(see text)
	IPV6_MULTICAST_IF	•	•	Specify outgoing interface		u_int
	IPV6_MULTICAST_HOPS	•	•	Specify outgoing hop limit		int
	IPV6_MULTICAST_LOOP	•	•	Specify loopback	•	u_int
	IPV6_JOIN_GROUP		•	Join multicast group		ipv6_mreq{}
	IPV6_LEAVE_GROUP		•	Leave multicast group		ipv6_mreq{}
IPPROTO_IP or IPPROTO_IPV6	MCAST_JOIN_GROUP		•	Join multicast group		group_req{}
	MCAST_LEAVE_GROUP		•	Leave multicast group		group_source_req{}
	MCAST_BLOCK_SOURCE		•	Block multicast source		group_source_req{}
	MCAST_UNBLOCK_SOURCE		•	Unblock multicast source		group_source_req{}
	MCAST_JOIN_SOURCE_GROUP		•	Join source-specific multicast		group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP		•	Leave source-specific multicast		group_source_req{}

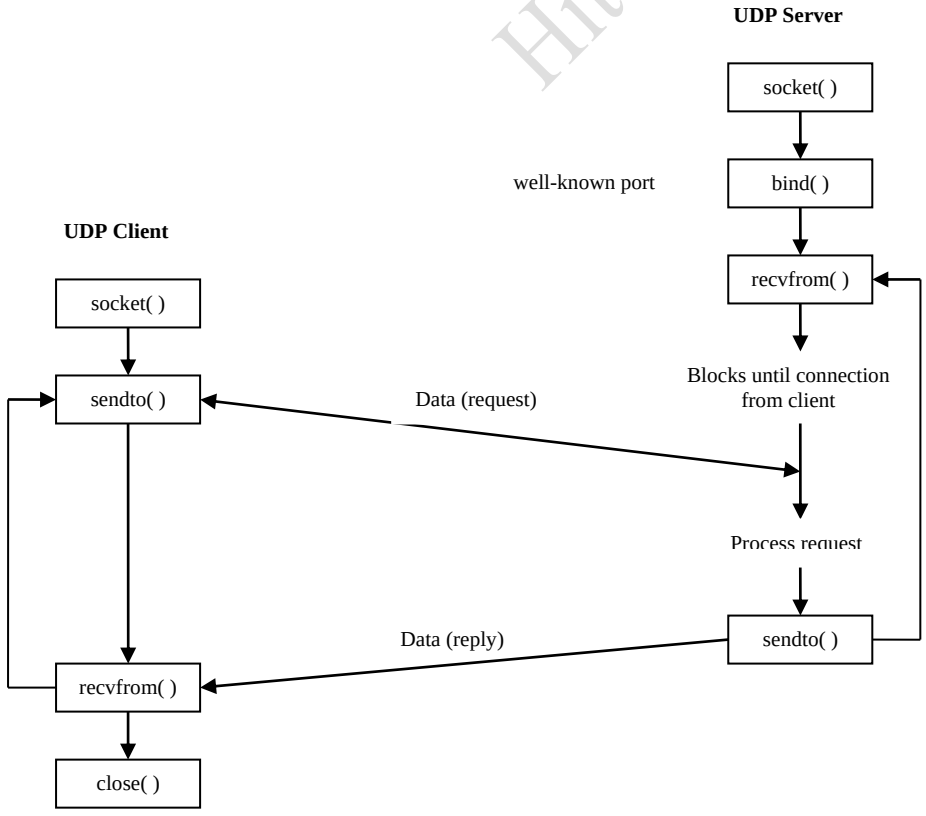
Figure 7.1. Summary of socket and IP-layer socket options for `getsockopt` and `setsockopt` .

level	optname	get	set	Description	Flag	Datatype
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP maximum segment size		int
	TCP_NODELAY	•	•	Disable Nagle algorithm	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	Adaption layer indication		sctp_setadaption{}
	SCTP_ASSOCINFO	†	•	Examine and set association info		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	Autoclose operation		int
	SCTP_DEFAULT_SEND_PARAM	•	•	Default send parameters		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP fragmentation	•	int
	SCTP_EVENTS	•	•	Notification events of interest		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	†		Retrieve peer address status		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	Mapped v4 addresses	•	int
	SCTP_INITMSG	•	•	Default INIT parameters		sctp_initmsg{}
	SCTP_MAXBURST	•	•	Maximum burst size		int
	SCTP_MAXSEG	•	•	Maximum fragmentation size		int
	SCTP_NODELAY	•	•	Disable Nagle algorithm	•	int
	SCTP_PEER_ADDR_PARAMS	†	•	Peer address parameters		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	†	•	Primary destination address		sctp_setprim{}
	SCTP_RTOINFO	†	•	RTO information		sctp_rtoinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR		•	Peer primary destination address		sctp_setpeerprim{}
	SCTP_STATUS	†		Get association status		sctp_status{}

Figure 7.2. Summary of transport-layer socket options.

Elementary UDP Sockets

DNS, NFS and SNMP are some examples for UDP applications. The following figure shows the function calls for a typical UDP client/server. The client does not establish a connection with the server. Instead, the client just sends a datagram into the server using the "sendto" function, which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the "recvfrom" function, which waits until data arrives from some client. "recvfrom" returns the protocol address of the client along with the datagram, so the server can send a response to the correct client.



Socket functions for UDP client/ server

"recvfrom" and "sendto" Functions:

These functions are similar to the standard "read" and "write" functions along with three additional arguments:

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
      struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
      const struct sockaddr *to, socklen_t addrlen);
```

Here, the first three arguments: "sockfd, buff and nbytes" are identical to the first three arguemtns for "read" and "write": descriptor, pointer to buffer to read into or write from, and number of bytes to read and write. The "recvfrom" function fills in the socket address pointed to by "from" with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by "addrlen". The "to" argument is a socket address structure containing the protocol address of where the data is to be sent. The size of this socket address structure is specified by "addrlen". It should be noted that the final argument to "sendto" is an integer value, whereas it is a pointer to an integer value (a value-result argument) in case of "recvfrom".

The final two arguments of "recvfrom" are similar to the final two arguments of "accept", i.e. the contents of the socket address structure upon return call tells either who sent the datagram (in case of UDP), or who initiated the connection (in case of TCP).

The final two arguments of "sendto" are similar to the final two arguments of "connect". The socket address structure is filled either with the protocol address of where to send the datagram (in case of UDP) or, with whom to establish a connection (in case of TCP).

Both functions return the length of the data that was read or written as the value of the function. In "recvfrom", the return value is the amount of user data in the received datagram. Writing a datagram of length "0" is acceptable in UDP, since it is connectionless and there is no necessity for closing UDP connection. If the "from" argument to "recvfrom" is a null pointer, then the corresponding length argument (addrlen) must also be a null pointer, indicating that receiver is not interested in knowning the protocol address of who sent the data.

/* udpserver.c */

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

int main()
{
      int sock;
      int addr_len, bytes_read;
      char recv_data[1024];
      struct sockaddr_in server_addr , client_addr;


      if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
```

```
        perror("Socket");
        exit(1);
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(5000);
    server_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(server_addr.sin_zero),8);


    if (bind(sock,(struct sockaddr *)&server_addr,
        sizeof(struct sockaddr)) == -1)
    {
        perror("Bind");
        exit(1);
    }

    addr_len = sizeof(struct sockaddr);

      printf("\nUDPServer Waiting for client on port 5000");
    fflush(stdout);

      while (1)
      {

     bytes_read = recvfrom(sock,recv_data,1024,0,
                    (struct sockaddr *)&client_addr, &addr_len);


       recv_data[bytes_read] = '\0';

     printf("\n(%s , %d) said : ",inet_ntoa(client_addr.sin_addr),
                            ntohs(client_addr.sin_port));
     printf("%s", recv_data);
        fflush(stdout);

    }
    return 0;
}
```

/* udpclient.c */

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

int main()
{
int sock;
struct sockaddr_in server_addr;
struct hostent *host;
char send_data[1024];

host= (struct hostent *) gethostbyname((char *)"127.0.0.1");
```

```
if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
perror("socket");
exit(1);
}

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(5000);
server_addr.sin_addr = *((struct in_addr *)host->h_addr);
bzero(&(server_addr.sin_zero),8);

   while (1)
   {

    printf("Type Something (q or Q to quit):");
    gets(send_data);

    if ((strcmp(send_data , "q") == 0) || strcmp(send_data , "Q") == 0)
       break;

    else
       sendto(sock, send_data, strlen(send_data), 0,
              (struct sockaddr *)&server_addr, sizeof(struct sockaddr));

   }

}
```


Daemon Process, Syslogd Daemon, Syslog function ioctl operation

A daemon is a process that runs in the background and, performing different administrative tasks and is not associated with a controlling terminal.

The lack of a controlling terminal is typically a side effect of being started by a system initialization script (e.g., at boot-time). But if a daemon is started by a user typing to a shell prompt, it is important for the daemon to disassociate itself from the controlling terminal to avoid any unwanted interraction with job control, terminal session management, or simply to avoid unexpected output to the terminal from the daemon as it runs in the background.

There are numerous ways to start a daemon:

1. During system startup, many daemons are started by the system initialization scripts. These scripts are often in the directory /etc or in a directory whose name begins with /etc/rc, but their location and contents are implementation-dependent. Daemons started by these scripts begin with superuser privileges.

   A few network servers are often started from these scripts: the inetd superserver (covered later in this chapter), a Web server, and a mail server (often sendmail). The syslogd daemon will normally started by one of these scripts.

2. Many network servers are started by the inetd superserver. inetd itself is started from one of the scripts in Step 1. inetd listens for network requests (Telnet, FTP, etc.), and when a request arrives, it invokes the actual server (Telnet server, FTP server, etc.).
3. The execution of programs on a regular basis is performed by the cron daemon, and programs that it invokes run as daemons. The cron daemon itself is started in Step 1 during system startup.

4. The execution of a program at one time in the future is specified by the at command. The cron daemon normally initiates these programs when their time arrives, so these programs run as daemons.
5. Daemons can be started from user terminals, either in the foreground or in the background. This is often done when testing a daemon, or restarting a daemon that was terminated for some reason.

Since a daemon does not have a controlling terminal, it needs some way to output messages when something happens, either normal informational messages or emergency messages that need to be handled by an administrator. The syslog function is the standard way to output these messages, and it sends the messages to the syslogd daemon.

syslogd

Unix systems normally start a daemon named syslogd from one of the system initializations scripts, and it runs as long as the system is up. Berkeley-derived implementations of syslogd perform the following actions on startup:

1. The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log message that the daemon can receive. These messages can be appended to a file (a special case of which is the file /dev/console, which writes the message to the console), written to a specific user (if that user is logged in), or forwarded to the syslogd daemon on another host.
2. A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).
3. A UDP socket is created and bound to port 514 (the syslog service).
4. The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.

The syslogd daemon runs in an infinite loop that calls select, waiting for any one of its three descriptors (from Steps 2, 3, and 4) to be readable; it reads the log message and does what the configuration file says to do with that message. If the daemon receives the SIGHUP signal, it rereads its configuration file.

We could send log messages to the syslogd daemon from our daemons by creating a Unix domain datagram socket and sending our messages to the pathname that the daemon has bound, but an easier interface is the **syslog function.** Alternately, we could create a UDP socket and send our log messages to the loopback address and port 514.

Newer implementations disable the creation of the UDP socket, unless specified by the administrator, as allowing anyone to send UDP datagrams to this port opens the system up to denial-of-service attacks, where someone could fill up the filesystem (e.g., by filling up log files) or cause log messages to be dropped (e.g., by overflowing syslog's socket receive buffer).

Differences exist between the various implementations of syslogd. For example, Unix domain sockets are used by Berkeley-derived implementations, but System V implementations use a STREAMS log driver. Different Berkeley-derived implementations use different pathnames for the Unix domain socket. We can ignore all these details if we use the syslog function.

Syslog Function

```
#include <syslog.h>
```

```
void syslog(int priority, const char *message, ... );
```

The priority argument is a combination of a level and a facility as is given below

Figure *facility* of log messages.

facility	Description
LOG_AUTH	Security/authorization messages
LOG_AUTHPRIV	Security/authorization messages (private)
LOG_CRON	cron daemon
LOG_DAEMON	System daemons
LOG_FTP	FTP daemon
LOG_KERN	Kernel messages
LOG_LOCAL0	Local use
LOG_LOCAL1	Local use
LOG_LOCAL2	Local use
LOG_LOCAL3	Local use
LOG_LOCAL4	Local use
LOG_LOCAL5	Local use
LOG_LOCAL6	Local use
LOG_LOCAL7	Local use
LOG_LPR	Line printer system
LOG_MAIL	Mail system
LOG_NEWS	Network news system
LOG_SYSLOG	Messages generated internally by syslogd
LOG_USER	Random user-level messages (default)
LOG_UUCP	UUCP system

Log messages have a level between 0 and 7.

Figure *level* of log messages.

level	Value	Description
LOG_EMERG	0	System is unusable (highest priority)
LOG_ALERT	1	Action must be taken immediately
LOG_CRIT	2	Critical conditions
LOG_ERR	3	Error conditions
LOG_WARNING	4	Warning conditions
LOG_NOTICE	5	Normal but significant condition (default)
LOG_INFO	6	Informational
LOG_DEBUG	7	Debug-level messages (lowest priority)

When the application calls `syslog` the first time, it creates a Unix domain datagram socket and then calls `connect` to the well-known pathname of the socket created by the `syslogd` daemon (e.g., `/var/run/log`). This socket remains open until the process terminates. Alternately, the process can call `openlog` and `closelog`.

```
#include <syslog.h>
```

```
void openlog(const char *ident, int options, int facility);
```

```
void closelog(void);
```

`openlog` can be called before the first call to `syslog` and `closelog` can be called when the application is finished sending log messages.

ident is a string that will be prepended to each log message by `syslog`. Often this is the program name.

The options argument is formed as the logical OR of one or more of the constants

Figure : *options* for `openlog`.

options	Description
LOG_CONS	Log to console if cannot send to `syslogd` daemon
LOG_NDELAY	Do not delay open, create socket now
LOG_PERROR	Log to standard error as well as sending to `syslogd` daemon
LOG_PID	log the Process ID with each message