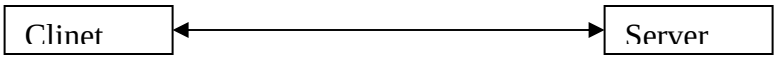
UNIT 1

Introduction:

Most network application can be divided into two programs: client and server with the communication link between them as shown:



Examples are : A web browser communicating with a web server. A FTP client fetching a file form an FTP server etc. A client normally communicates with a server at a time. However, a server is likely to communicate with multiple client. The client and server communication within in the same Ethernet and the communication when LAN connected through WAN is shown below.

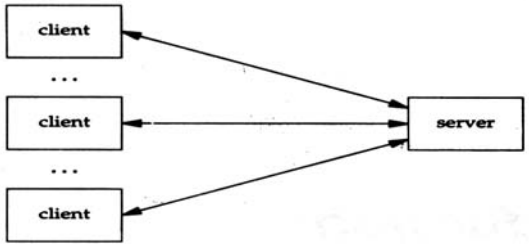


Figure 1.2 Server handling multiple clients at the same time.

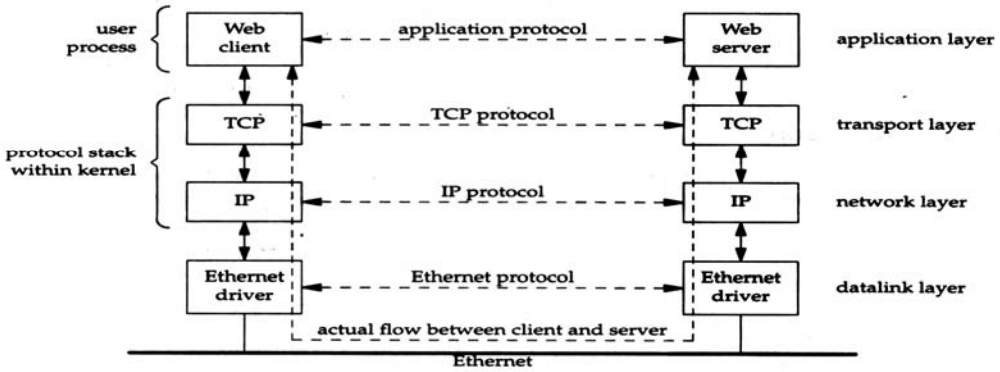


Figure 1.3 Client and server on the same Ethernet communicating using TCP.

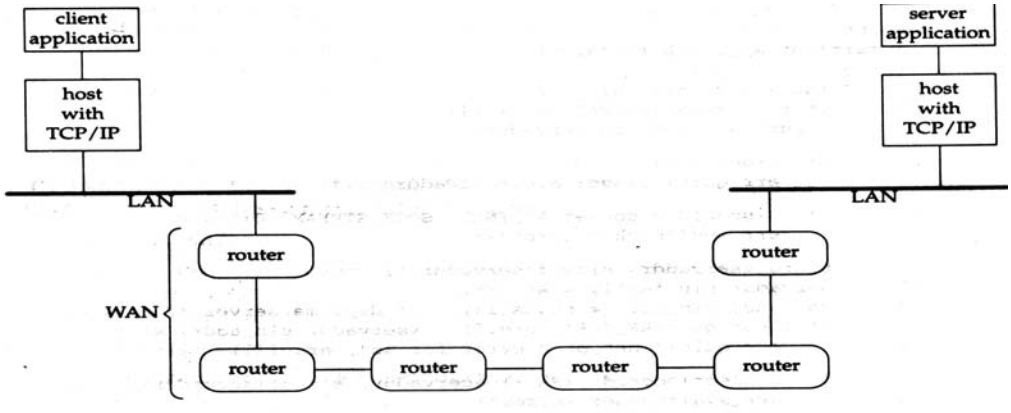


Figure 1.4 Client and server on different LANs connected through a WAN.

As seen in above figure , TCP and IP protocols are normally part of the protocol stack within the kernal. In addition to TCP and IP, other protocol like UDP is also used. IP that was in use since early 1980 is called as IP version 4 (Ipv4). A new version IP version 6 (Ipv6) is being used since mid 1990.

OSI model for the Internet protocol suite: The following figure provides the comparison of OSI reference model with that of Internet protocol suite.

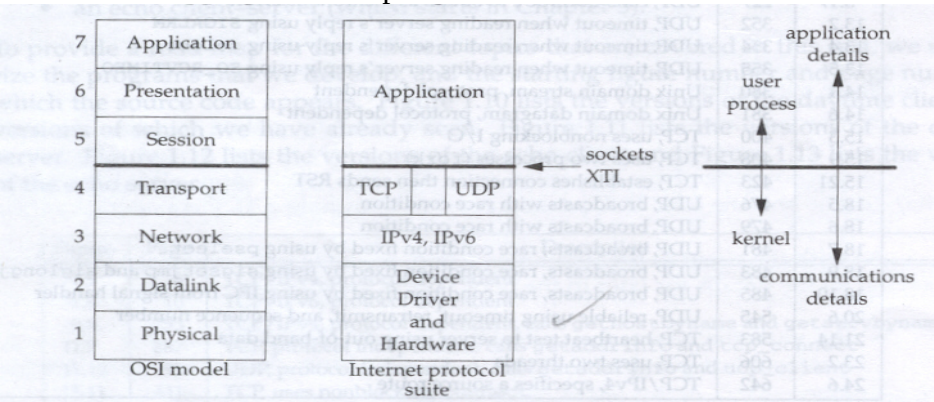


Figure 1.14 Layers in OSI model and Internet protocol suite.

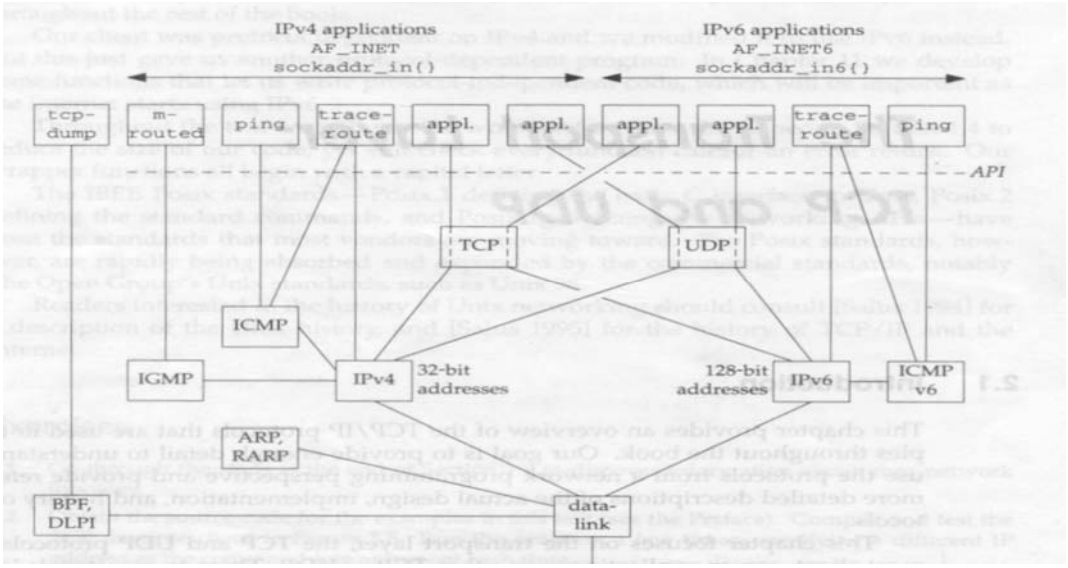
The upper three layers of OSI model are combined into a single layer called the application. This is the Web Client (Browser), Telnet client, the Web Server, FTP server or whatever application we are using. The transport layer that are chosen are TCP and UDP. As shown in the figure, it is possible to bypass both TCP and UDP and the application layer may directly use Ipv4 or Ipv6. This is called raw socket.

The two program interface the we will be studying are **sockets and XTI(X/Open Transport Interface)** . It is intended to study : how to write applications using either Sockets or XTI that use either TCP or UDP. We need this because, upper three layers handle all the details of the application (FTP, Telnet or HTTP etc) and know little about the communication details. Whereas the lower four layers know little about the application but handle all communication details: sending data, waiting for an acknowledgement sequencing data that arrives out of order , calculating and verifying the checksums and so on. Upper three layers are normally provided as part of the operating system kernal.

The Transport Layer:

The transport layer makes use of TCP and UDP protocols. UDP is a simple, unreliable, datagram protocol while TCP is a sophisticated, reliable, byte stream protocol. We need to understand the services provided by these two transport layer to the application, so that we know what is handled by the protocol and what we must handle in the application.

Although, it is known as TCP / IP suite, there are more members to the family as shown below.



Both Ipv4, Ipv6 are shown in the above figure. Moving from right to left in this figure, the rightmost four application are using **Ipv6**. We need to understand **AF_INET 6** and ***sockaddr_in6*** in this. The next five applications use **Ipv4**. The ***tcpdump*** communicates directly with the data link using either **BPF** (BSD Packet Filter) or **DLPI** (Data Link Provider Interface). The dashed line marked as API is normally either sockets or XTI The interface to BPF or DLPI **does not use** sockets or XTI. ***traceroute*** uses two sockets : one for IP and other for ICMP. Each of the protocol is described below.

IPv4: It provides the **packet delivery service** for **TCP, UDP, ICMP and IGMP**. It uses 32 bit address.
IPv6: It Is designed as replacement for IPv4. IT has larger address made up of **128 bits**. It provides packet delivery service for **TCP, UDP and ICMPv6**.
TCP: It is a **connection oriented protocol** that provides a **reliable, full duplex, bytestream** for a user process. It takes care of details such as **acknowledgment, timeouts, retransmissions** etc. TCP can **use either IPv4 or IPv6**.
UDP: It is a **connectionless protocol** and UDP sockets are example of **datagram sockets**. In this there is **no guarantee** that UDP datagram ever **reach their intended destination**. It can also use **IPv4 or IPv6**.
ICMP: **Internet Control Message Protocol** : It handles **errors and control information between router and hosts.** These are generated and processed by TCP /IP networking software itself.
IGMP : **Internet Group Management Protocol**: It is used with **multicasting** which is optional with IPv4.
ARP : **Address Resolution Protocol, maps** an **IPv4 address** into a **hardware address** (such as an Ethernet address). ARP is normally used on a broadcast networks such as Ethernet, token ring and FDDI but it is not needed in a point to point network.
RARP : **Reverse ARP**: This maps a **hardware address into an IPv4 address**. It is sometimes used when a diskless node such as X terminal is booting.
ICMPv6: It **combines** the functionality of **ICMPv4, IGMP and ARP**.
BPF : BSD Packet Filter: This interface provides the access to the datalink for a process. It is found in Berkley derived kernals.

DLPI: Data Link Provider Interface. This provides access to the datalink and is normally provided with SVR4.

All the above protocols are defined in the **RCF (Request For Comments**), which are supported by their formal specification.

UDP: User Datagram Protocol ():

- The application writes a datagram to a UDP socket which is encapsulated either in an IPv4 or IPv6 datagram and then sent to destination.
- UDP datagram lacks reliability as it requires to build acknowledgements, timeouts, retransmission etc in the protocol.
- Datagram has a length and if its checksum is correct at the receiving end, it is passed to the receiver.
- It is a connectionless service. A client UDP socket can send successive datagram to different server and similarly, the server UDP socket can receive datagram from different clients.

TCP: Transmission Control Protocol:

- A TCP provides a connections 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 provides reliability. When data is sent on a TCP socket, it waits for an acknowledgement for a duration equal or more than the RTT- round trip time. If after reasonable amount of time, the acknowledgment is not received, TCP will give up.
- RTT is calculated periodically to take care of the congestion in the network.
- TCP sequences the data by associating a sequence number with every byte that it sends. (A byte stream larger than 1024 bytes is split into segments of 1024 bytes and sent to IP. The sequence number of 1-1024 is allotted for first segment, 1025 - … allotted for second segment and so on.)
- TCP provides flow control: Receiver advertises the size of data which it can accept to its peer. This size of the window varies dynamically. If the buffer at the receiver is full, the receiver may not accept the data till it is free.
- TCP connection is also fully duplex: This means an application can send and receive data in both direction on a given connection at any time. The TCP must keep track of state information such as sequence number and window size for each direction of data flow: sending and receiving.

TCP connection establishment and termination:

To understand the ***connect, accept and close*** functions and to debug TCP application using *neststat* we need to follow the state transition diagram.

The **three way handshake** that take place is as follows:

1. The server must be prepared to accept an incoming connection. This is normally done by calling ***socket, bind and listen functions*** and is called passive open.
2. The *client* issues an ***active open*** by calling ***connect***. This causes the client TCP to send ***SYN* segment** to tell the server that the client's initial **sequence number** for the data that the client will send on that connection. No data is sent with SYN. ITtcontains an **IP header, TCP header and possible TCP options.**
3. The server acknowledges the client's SYN and sends its own SYN and the ACK of the client's SYN in a single segment.

4. The client must ACK the server's SYN.

As the minimum number of packets required is 3, it is called three way handshake. This is shown in the following figure.

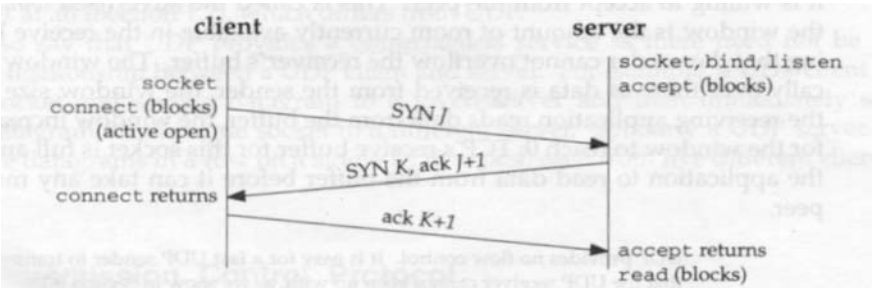


Figure 2.2 TCP three-way handshake.

J is the initial sequence number of client and Kis that of Server. ACK number is he initial sequence number plus 1.

TCP Options:

TCP SYN can contain TCP options. Common options are: MSS (Maximum Segment Size), Window Scale Option, Time Stamp Option.

- **MSS Option**: With this option the TCP sending SYN announces its ***maximum segment size***, the maximum amount of data that it is willing to accept in each TCP segment, on this connection. This option is set by **TCP_ MAXSEG** socket option.
- **Window scale option**: The maximum window that either TCP can advertise to the other TCP is 65535 as the corresponding field in the TCP header occupies 16 bits. But high speed connections (45 Mbps/sec) or long delay paths require larger window which can be set by left shifting (scaling) by 0-14 bits giving rise to one gigabyte. This is effected with **SO_RCVBUF** socket option.
- **Timestamp option**: This option is needed for high speed connections to prevent possible data corruption caused by lost packets that then reappears.

Last two options being new, may not be supported. These are also known as 'long fat pipe' option.

TCP connection Termination:

It takes **four segment** to terminate a TCP connection as shown below:

1. One application calls ***close,*** and we say that this end performs the active close. This end's TCP sends FIN segment, which means it is finished sending data.

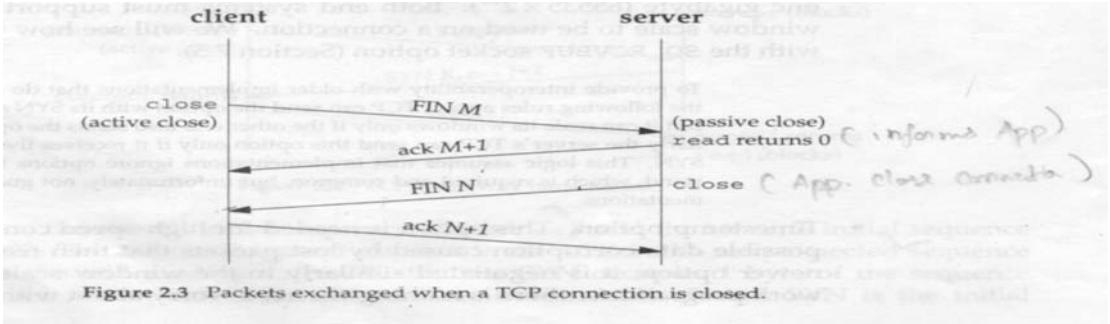


Figure 2.3 Packets exchanged when a TCP connection is closed.

2. The other end that receives the **FIN** performs the passive close. The received **FIN** is

acknowledged by TCP. The FIN is passed to the application as an end of file(after any data that may already be queued for the application to receive) and the receiver will not receive any further data from the sender.

3. When the received application closes its socket, the TCP sends FIN.
4. The TCP on the system that receives the FIN acknowledges the FIN.

TCP state transition diagram: The operation of TCP connection establishment and connection termination can be specified with a state transition diagram which is shown below

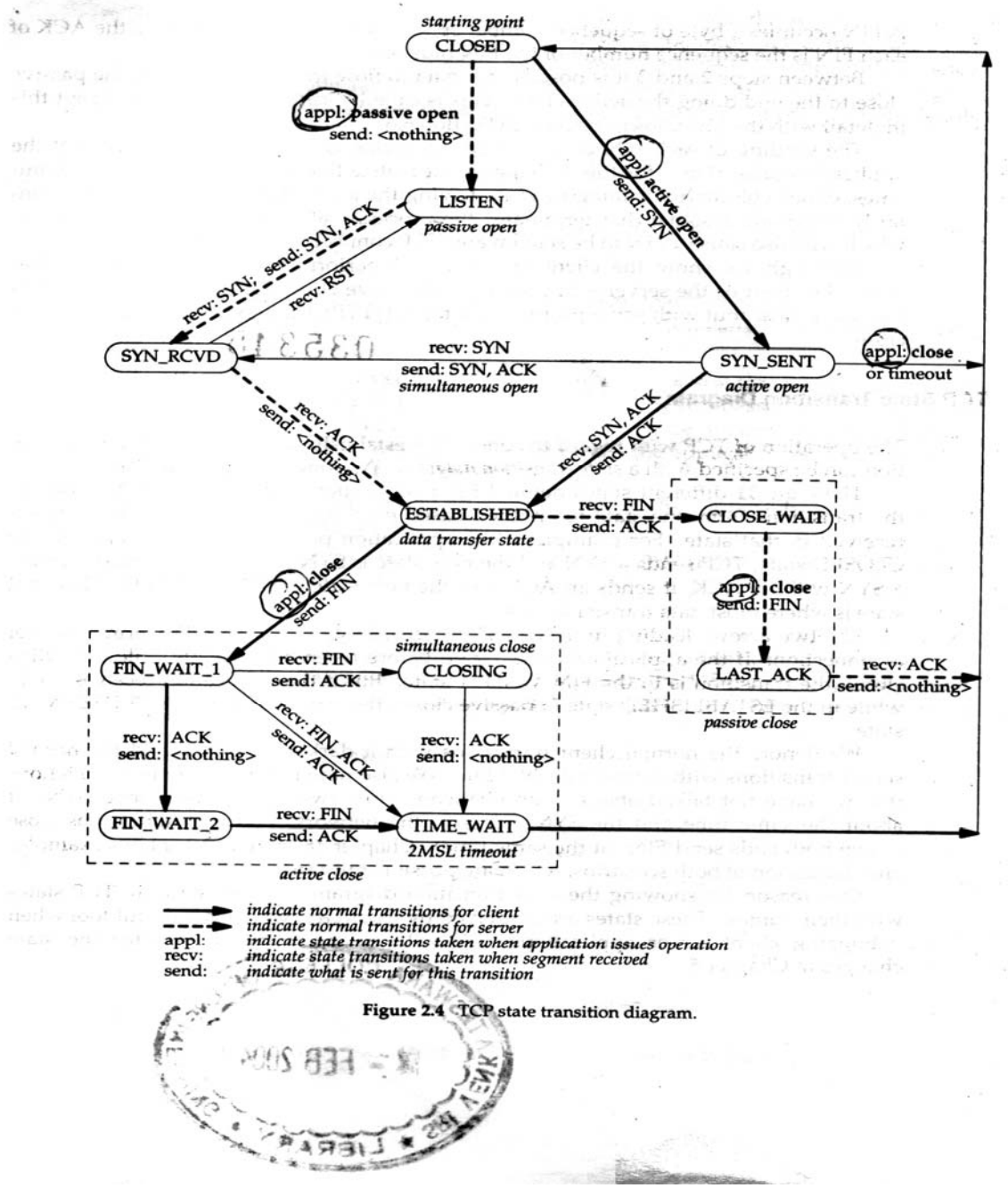


Figure 2.4 TCP state transition diagram.

Following figure shows the actual packet exchange that takes place for a complete TCP connection.

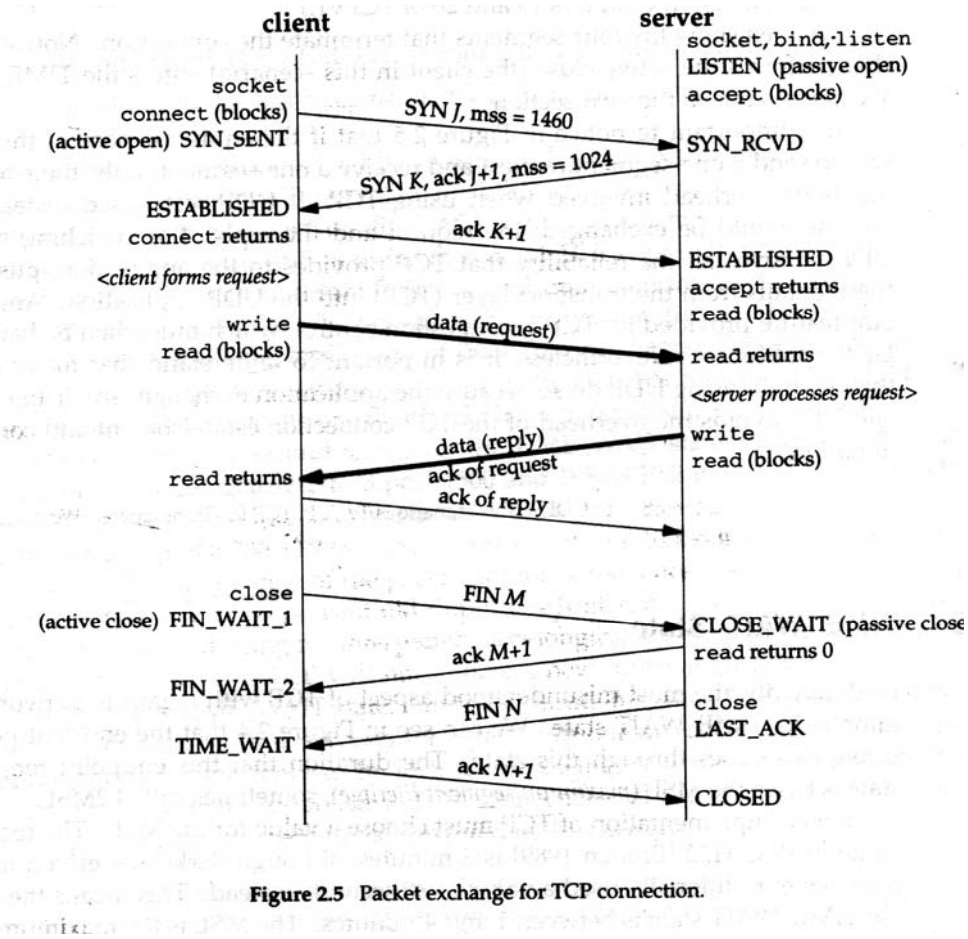


Figure 2.5 Packet exchange for TCP connection.

The client in this example announces an MSS(Maximum Segment Size) of 1460 and the server announces an MSS of 1024. Once the connection is established, the client forms a request and sends it to the server (fits in one segment). The server processes the request and sends a reply. The data segments transfer is shown in bold arrows The four segment shown terminate the connection. As it can be seen that the end that performs the *active close* enters the TIME_WAIT state.

Port Numbers: At any time multiple processes can use either UDP or TCP and both use 16 bit integer port numbers to differentiate these processes. Both TCP and UDP define a group of well known ports that identity services. Some of these are port 21 for FTP, TFTP is assigned UDP port 69 etc. Clients use ***ephemeral ports*** that is short lived ports. These port numbers are manually assigned by TCP or UDP to the client. The port numbers are divided into three categories

- **Well known ports** : 0 – 1023 These port numbers are controlled and assigned by the IANA. When possible same port number is assigned for both TCP and UDP as in the case of web server.
- **The Registered Ports**: 1024 – 49151 These are not controlled by IANA bit it registers and list the uses of these ports as a convenience to the community
- **Dynamic private ports** : 49152 – 65535 These are what we call as ephemeral ports.

Socket Pair: The two values that identify each endpoint, an IP address and a port number, are called a socket. The socket pair for a TCP connection is the 4 tuple that defined the two end points of the connection: the local IP address, local TCP port, foreign IP address, and foreign TCP port.

Socket Address Structure SAS:

This SAS is between application and kernal. An address conversion function translates between text representation of an address and binary value that makes up SAS. IPv4 uses inet_addr and inet_ntoa. But inet_pton and inet_ntop handle IPv4 and IPv6. Above functiuons are protocol dependent. However the functions starting with sock are protocol independent. Most socket functions require a pointer to a socket address structure as an argument.

IPv4 Socket AS: It is defined as follows:

include <netinet / in.h>

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

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[8]; /* unused */
};
```

sin_len, added in 4.3 BSD, is not normally supported by many vendors. It facilitates handling of variable length socket address structures.

Various data types that are commonly used are listed below:

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 8 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,normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netint/in.h>
in_port_t	TCP or UDP port normally uint16_t	<netinet/in.h>

Length field is never used and set. It is used within the kernal before routines that deal with socket address structures from various protocol families.

Four socket functions – **bind(), connect(), sendto(), sendmsg()** -pass socket address structures from application to kernal. All invoke **sockargs()** in Berkley derived implementation. This function copies socket address structures and explicitly set the **sin_len** member to the size of the structure that was passed. The other socket functions that pass socket address to the application from kernal **accept(), recvfrom(), recvmsg(), getpeername() and getsockname()** all set the **sin_len** member before returning to the process.

sin_port, sin_family and sin_addr are the only required for Posix.1g. ***sin_zero*** is implemented to keep the structure length to 16 byte.

Generic Socket Address Structure.

Socket address structures are always passed by reference when passed as an arguments to any of the socket functions.

int bind (int sockfd, struct sockaddr *, socklen_t);

But the socket function that accept these address structures as pointers must deal with any of these supported protocols. This calls for the any functions must cast the pointer to the protocol specific socket address structure to be a pointer to a generic socket address structure. For example

```
struct  sockaddr_in serv;
bind (sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

If we omit the cast, the C compiler generates a warning of the form incompatible pointer type.

Different socket address structures are :

IPv4 (24 bytes), IPv6 (24 bytes), Unix variable length and Data link variable length.

IPv6 SAS:

Defined by # include<netinet/in.h> header. The structure is shown below:

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t       sin6_len;      /* length of this struct (24) */
  sa_family_t   sin6_family;   /* AF_INET6 */
  in_port_t     sin6_port;     /* transport layer port# */
                               /* network byte ordered */
  uint32_t      sin6_flowinfo; /* priority & flow label */
                               /* network byte ordered */
  struct in6_addr sin6_addr;   /* IPv6 address */
                               /* network byte ordered */
};
```

Figure 3.4 IPv6 socket address structure: sockaddr_in6.

Important points to note are:

- The **SIN_LEN** constant must be defined if the system supports the length members for socket address structures.
- The IPv6 family is **AF_INET6**

- The members in this structure are ordered so that if the **sockaddr_in6** structure is 64 bit aligned, so is the 128 bit **sin6_addr** member.
- The sin6_flowinfo member is divided into three fields
  - The low order 24 bits are the flow label
  - The next 4 bits are the priority
  - The next 4 bits are reserved.

Comparison of socket address structure:

Following figure shows the comparison of the four socket address structures that are encountered.

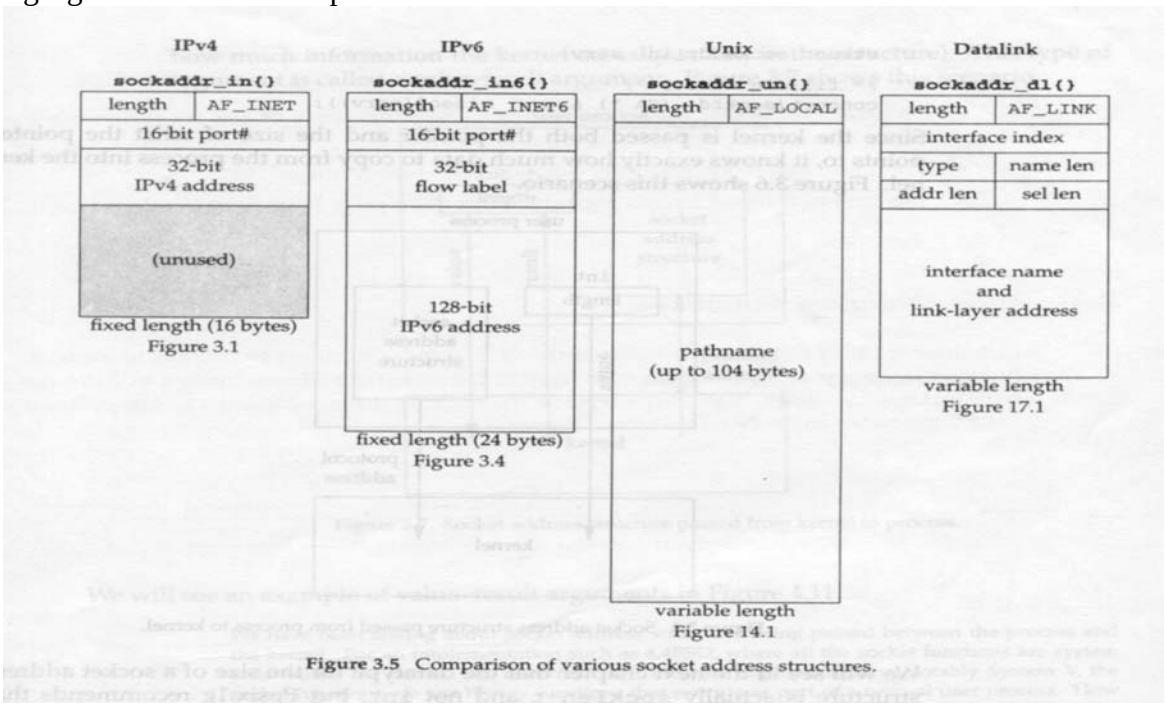


Figure 3.5 Comparison of various socket address structures.

Value Result Arguments: The socket address structure is passed to any of the socket function by reference. The length of the structure is also passed as argument to the function. But the way the length is passed depends on which direction it is being passed. From the process to the kernal or kernal to the process.

1. The three functions **bind(), connect() and sendto()** pass a socket address structure from the process to the kernal. One argument to these three function is the pointer to the socket address structure and another argument is the integer size of the structure.
   struct sockaddr_in serv;
   connect (sockfd, (SA *) & serv, sizeof(serv));

Since the kernal is passed both pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernal. Following figures shows this scenario:

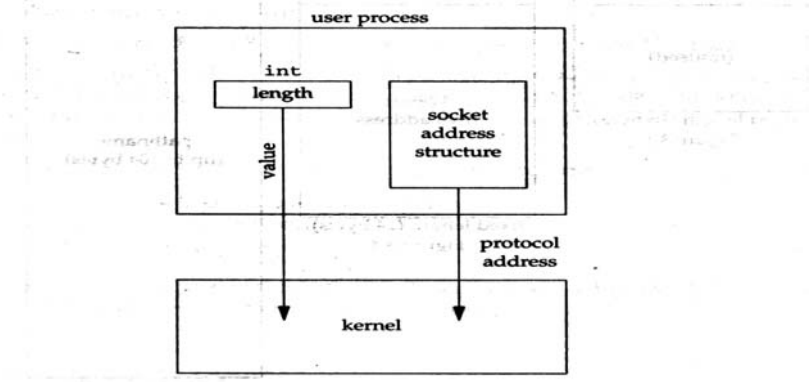


Figure 3.6 Socket address structure passed from process to kernel.

The four functions **accept(), recvmsg(), getsockname() and getpeername()** pass a socket address structure from kernal to the process, the reverse direction form the precious scenario. In this case the length is passed as pointer to an integer containing the size of structure as in

```
struct sockaddr_un cli; // Unix domain//
socklen_t    len;
len = sizeof (cli);
getpeername (unixfd,  (SA *) &cli, & len );
```

The reason that the size changes from an **integer to be a pointer to an integer** is because the size is both **value** when the function is called (it tells the kernal the size of the structure so that the kernal does not write past the end of the4 structure when filling it) and it is the **result** when the function results (It tells the process how much information the kernal actually stored in the structure). This type of argument is called **value – result arguments**.

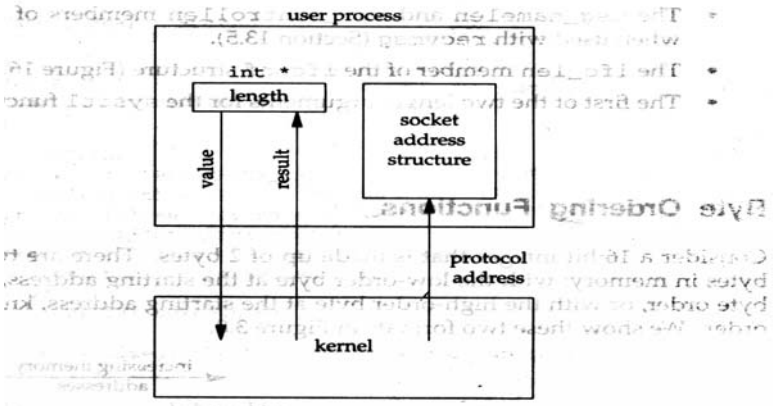


Figure 3.7 Socket address structure passed from kernel to process.

With variable length socket address structure, the value returned can be less than the maximum size of the structure.

Byte Order functions: There are two ways to store the 2 bytes in the memory. With the low order byte at the starting address known little endian byte order a or with high order byte at the starting address known as big endian byte order.

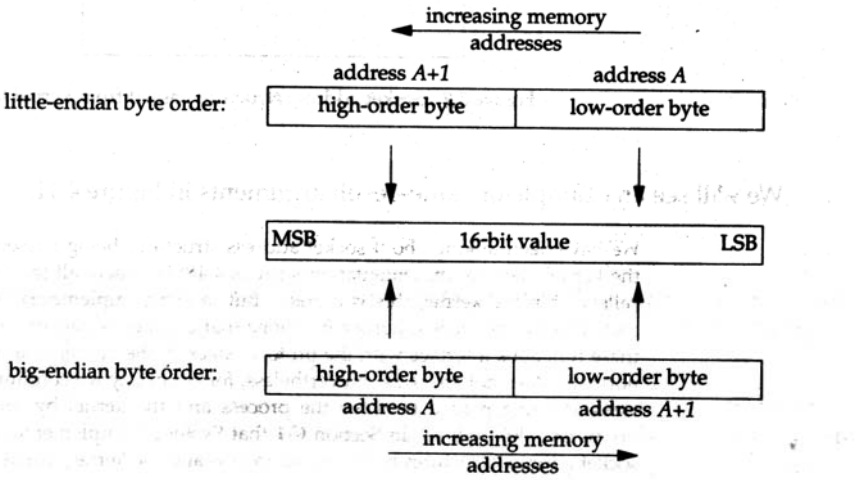


Figure 3.8 Little-endian byte order and big-endian byte order for a 16-bit integer.

The terms **little endian or big endian** indicate which **end of the multi byte** value , the little end or the big end is **stored at the starting address** of the value. Different systems use different orderings. For example, PowerPC of IB, sparc of Sun Soloaris, Happal of HP all use big endian while i386 PC of BSDi and i586 pc of Linux use little endian. The byte order used by a given system is known as the host byte order. As network programmer, one need to deal with the different byte orders. That is the sending and receiving protocol stack must agree on the order in which the bytes of these multibyte fields are transmitted. Internet protocol uses big endian byte ordering for these multibyte system.

Normally, the address structure may be maintained in the host byte order system. Then it may be converted into network byte order as per requirement. However, Posix.1g specifies that the certain files in socket address structure be maintained in network byte order . Therefore, there are function that convert between these two byte orders.

 # include <netinet/in.h>
uint16_t htons(uint16_t host16bitvalue);
uint32_t htons(uint32_t host32bitvalue);
Return value in network byte order.
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohs(uint32_t net16bitvalue);
Returns value in host byte order.

Appropriate function are called to convert a given value between the host and network byte order. On those systems that have the same byte order as the Internet protocols (big endian), these four functions are usually defined as null macros.

Byte Manipulation Functions:

There are two groups of functions that operate on multi byte fields, without interpreting the data, and without assuming that the data is a null terminated C string. We need these types of functions when dealing with sockets address structures as we need to manipulate fields such as IP addresses which can contain byte of 0, but these fields are not character strings. The functions beginning with str (for string), defined by including the < string.h> header, deal with null terminated C character strings.

The first group of functions whose name begin with **b (for byte)** are from 4.3. BSD. The second group of functions whose name begin with **mem (** for memory**)** are from ANSI C library.

First Berkeley derived functions are shown.

 # include <strings.h>

void bzero (void *dest, size_t nbytes);

void bcpy (const void *src, void * dest, size_t nbytes);
int bcmp (cost void * ptr1, const void *ptr2, size_t nbytes)

constant qualifier indicates that the pointer with this qualification, src, ptr1, ptr2 are not modified by the function.. That is memory pointed to by the cost pointer is read but not modified by the function.

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 byte strings . The return value is zero if the two byte strings are identical; otherwise it is nonzero.

Following are the ANSI C functions:

 # include <strings.h>
void memset (void *dest, int c, size_t len);
void memcpy (void *dest, const void * src, size_t nbytes);
int memcmp (const void * ptr1, const void *ptr2, size_t nbytes);
Returns 0 if equal, <0 or >0 if unequal.

memset () sets the specified number of bytes to the value in c in the destination, **memcpy()** is similar to bcopy () but the order of the two pointer arguments is swapped**.** ***bcopy*** correctly handles overlapping fields, while the behaviour of memcpy() is undefined if the source and destination overlap. **memmove**() functions can be used when the fields overlap. **memcpy()** compares two arbitrary byte strings and returns 0 if they are identical, if not, the return value is either greater than 0 or less than 0 depending whether the first unequal byte pointed to by ptr1 is greater than or less than the corresponding byte pointed to by ptr 2.

Address conversion functions: There are two groups of address conversion function that convert the Internet address between ASCII strings (readable form) to network byte ordered binary values and vice versa.

inet_aton( ), inet_addr( ), and inet_ntoa( ) : convert an IPv4 address between a dotted decimal string (eg 206.62.226.33) and it s 32 bit network byte ordered binary values

#include <arpa/inet.h>

int inet_aton (const * strptr, strut in_addr * addptr);

The first of these**, *inet_aton*( )** converts the C character strings pointed to by the ***strptr*** **into its 32 bit binary network byte ordered value** which is **stored** through the pointer ***addptr*.** If successful 1 is returned otherwise a 0.

in_addr_t inet_addr (const char * strptr);

in**et_addr( )** does the same conversion, returning the 32 bit binary network byte ordered value as the return value. Although the IP address (0.0.0.0 through 255.255.255.255) are al valid addresses, the functions returns the constant **INADDR_NONE** on an error.

This is deprecated and the new code should use **inet_aton** instead.

The function **inet_ntoa ( )** function converts a 32 bit binary network byte ordered IPv4 address into its corresponding dotted decimal string. The string pointed to by the return value of the function resides in static memory. This functions structure as arguments, not a pointer to a structure. (This is rare)

inet_pton ( ) and inet_ntop( ) functions:
These two functions are new with the IPv6 and work with both IPv4 and IPv6 addresses.
The letter **p** and **n** stands for **presentation and numeric**. Presentation format for an address is often ASCII string and the numeric format is the binary value that goes into a socket address structure.

 # include <arpa/inet.h>
int inet_pton (int family, const char *strptr, void *addrptr);
const char *inet_ntop (int family, cost void *addrptr, char *strptr, size_t len);

The family argument for both function is either **AF-INET or AF_ INET6**. If family is not supported, both functions return –1 with errno set to **EAFNOSUPPORT.**

The first functions tries to convert the string pointed to by strptr, storing the binary results through the pointer addrptr. IF successful, the return value is 1. If the input string is not valid presentation format for the specified family, 0 is returned.

inet_pton () does the reverse conversion from **numeric (addrptr) to presentation (strptr)**. The len argument is the size of the destination, to prevent the function from overflowing the caller's buffer. To help specify this size, following two definitions are defined by including the **<netinet/in.h>** header:

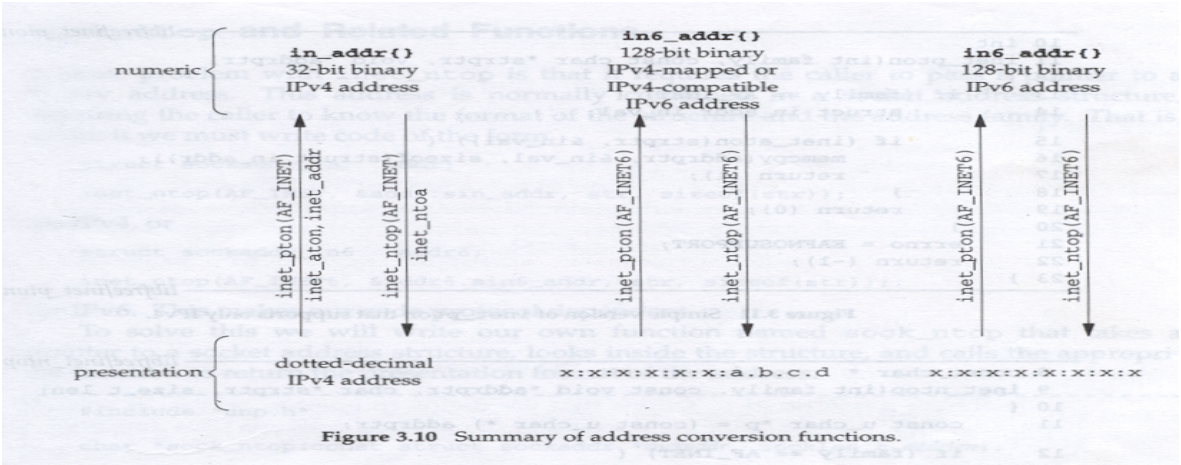


Figure 3.10 Summary of address conversion functions.

#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46

If LEN is too small to hold the resulting presentation format including the terminating null, a null pointer is returned and *errno* is set ot ENOSPC.

The ***strptr*** argument to ***inet_ntop*** cannot be a null pointer. The caller must allocate memory for the destination and specify its size. On success this pointer is the return value of the function. This is summarized in the following figure.

System calls used with sockets:

Socket calls are those functions that provide access to the underlying functionality and utility routines that help the programmer. A socket can be used by client or by a server, for a stream transfer (TCP) or datagram (UDP) communication with a specific endpoints address.

Following figure shows a time line of the typical scenario that takes place between client and server.

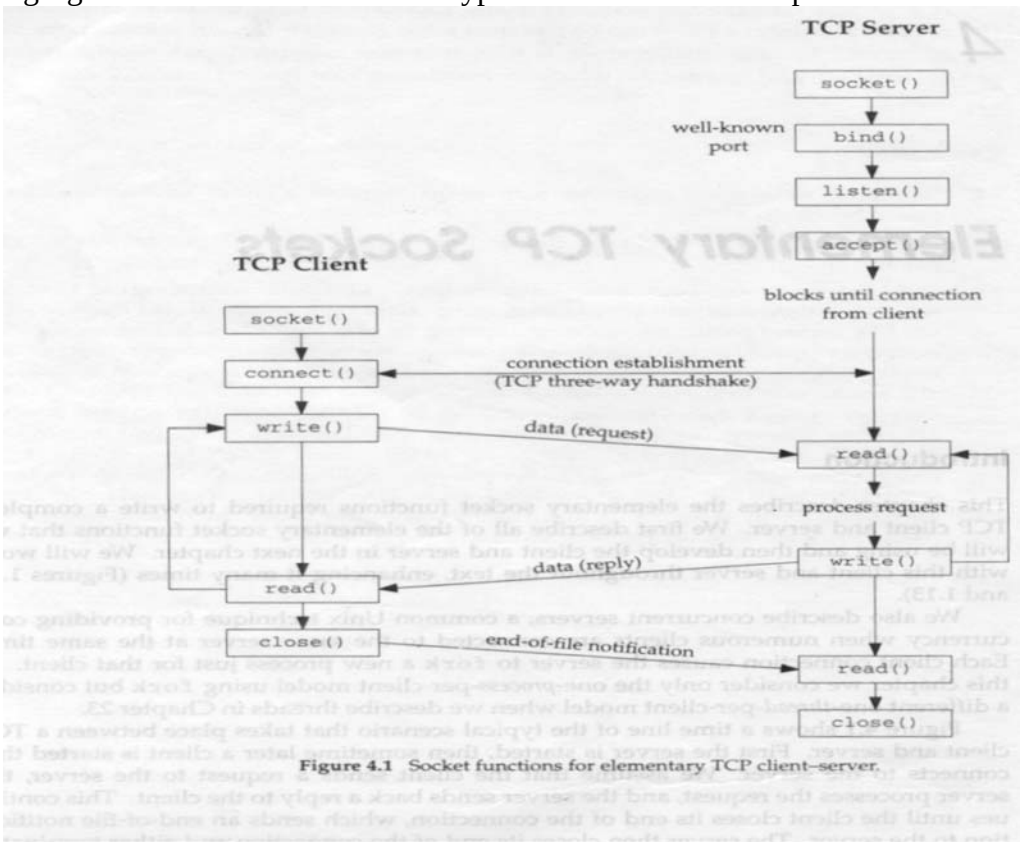


Figure 4.1 Socket functions for elementary TCP client–server.

First server is started, then sometimes later a client is started that connects to the server. The client sends a request to the server, the server processes the request, and the server sends back 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. The server then closes its end of the connections and either terminates or waits for a new connection.

socket function:

#include socket (int family, int type, int protocol);

returns negative descriptor if OK & –1 on error.

Arguments specify the protocol family and the protocol or type of service it needs (stream or datagram). The protocol argument is set to 0 except for raw sockets.

Family	Description		Type	Description
AF_INET	IPv4 protocols		SOCK_STREAM	Stream Socket
AF_INET6	IpV6 Protocols		SOCK_DGRAM	Datagram socket
AF_ROUTE	Unix domain protocol		SOCK_RAW	Raw socket
AF_ROUTE	Routing sockets			
AF_KEY	Key Socket			

Not all combinations of socket family and type are valid. Following figure shows the valid combination.

	AF-INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	YES		
SOCK_DGRAM	UDP	UDP	YES		
SOCK_RAW	IPv4	IpV6		YES	YES

connect Function : The connect function is by a TCP client to establish a active connection with a remote server. The arguments allows the client to specify the remote end points which includes the remote machines IP address and protocol port number.

include <sys/socket.h>

int connect (int sockfd, const struct sockaddr * servaddr, socklen_t addrelen)

returns 0 if ok -1 on error.

sockfd is the socket descriptor that was returned by the socket function. The second and third arguments are a pointer to a socket address structure and its size.

In case of TCP socket, the **connect()** function **initiates TCP's three way handshake**. The function returns only when the connection is established or an error occurs. Different type of **errors** are :

1. If the client TCP receives no response to its **SYN** segment, **ETIMEDOUT** is returned. This is done after the **SYN** is sent after, ***6sec, 24sec*** and if no response is received after a total period of ***75 seconds***, the error is returned.
2. In case for **SYN** request, a **RST** is returned (hard error), this indicates that no process is waiting for connection on the server. In this case **ECONNREFUSED** is returned to the client as soon the **RST** is received. RST is received when (a) a **SYN** arrives for a port that has no listening server (b) when **TCP** wants to abort an existing connection, (c) when **TCP** receives a segment for a connection does not exist.
3. If the **SYN** elicits an **ICMP** destination is unreachable from some intermediate router, this is considered a soft error. The client server saves the message but keeps sending **SYN** for the time period of 75 seconds. If no response is received, **ICMP** error is returned as **EHOSTUNREACH** or **ENETUNREACH**.

In terms of the TCP state transition diagram, ***connect()*** moves from the **CLOSED** state to the **SYN_SENT** state and then on success to the **ESTABLISHED** state. If the connect fails, the ***socket*** is no longer usable and must be closed.

Bind(): When a socket is created, it does not have any notion of end points addresses An application calls bind to specify the local endpoint address for a socket. That is the bind function assigns a local port and address to a socket..

#include <sys/socket.h>

int bind (int sockfd, const strut sockaddr *myaddr, socklen_t addrlen)

The second arguments is a pointer to a protocol specific address and the third argument is the size of this address structure. Server bind their well known port when they start. (A TCP client does not bind an IP address to its socket.)

listen Function:

The listen function is called only by **TCP** server and it performs following functions.

The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of TCP transmission diagram the call to listen moves the socket from the **CLOSED** state to the **LISTEN** state.

The second argument to this function specifies the maximum number of connections that the kernel should queue for this socket.

#include <sys/socket.h>

int listen (int sockfd, int backlog); returns 0 if OK -1 on error.

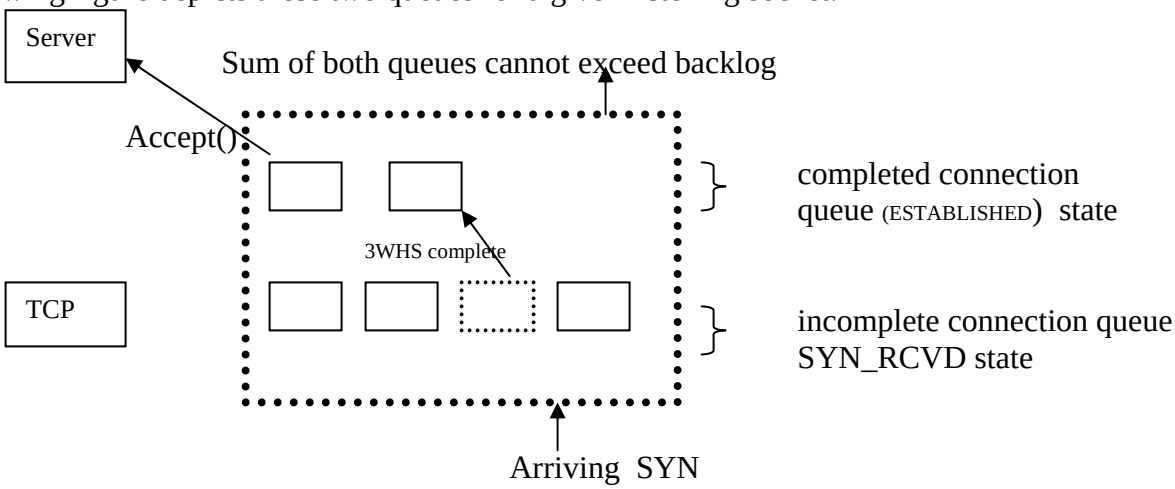
This function is normally called after both the socket and bind functions and must be called before calling the accept function.

The kernel maintains two queues and the backlog is the sum of these two queues. These are :

An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three way handshake. These sockets are in the **SYN_RECD** state.

A Completed Connection Queue which contains an entry for each client with whom three handshake has completed. These sockets are in the **ESTABLISHED** state.

Following figure depicts these two queues for a given listening socket.



The two queues maintained by TCP for a listening socket.

When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three way handshake. The server's SYN with an ACK of the clients SYN. This entry will remain on the incomplete queue until the third segment of the three way handshake arrives (the client's ACK of the server's SYN) or the entry times out. If the three way hand shake completes normally, the entry moves from the incomplete queue to the completed queue. When the process calls accept, the first entry on the completed queue is returned to the process or, if the queue is empty, the process is put to sleep until an entry is placed onto the

completed queue. If the queue are full when a client arrives, TCP ignores the arriving SYN, it does not send an RST. This is because the condition is considered temporary and the client TCP will retransmit its SYN with the hope of finding room in the queue.

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

 # include <sys/socket.h>
int accept (sockfd, struct sockaddr * cliaddr, socklen_t *addrlen) ;
return non negative descriptor if OK, -1 on error.

The *cliaddr* and addrlen arguments are used to return the protocol address of the connected peer process (the client). *addrlen* is a value-result argument before the call, we set the integer value pointed to by **addrlen* to the size of the socket address structure pointed to by *cliaddr* and on return this integer value contains the actual number of bytes stored by the kernel in the socket address structure. If *accept* is successful, its return value is a brand new descriptor that was automatically created by the kernel. This new descriptor refers to the TCP connection with the client. When discussing *accept* we call the first argument to accept the listening and we call the return value from a accept the connected socket

fork function:

fork is the function that enables the Unix to create a new process
#inlcude <unistd.h>
pid_t fork (void); ***Returns 0 in child, process ID of child in parent, -1 on error***

There are two typical uses of fork function:

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 normal way of working in a 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 process) calls exec function to replace itself with a the new program. This is typical for program such as shells.
3. ***fork*** function although called once, it returns twice. It returns once in the calling process (called the parent) with a return value that is process ID of the newly created process (the 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.
4. The reason ***fork*** returns 0 in the child, instead of parent's process ID is because a child has only one parent and it can always obtain the parent's process ID by calling ***getppid*** A parent, on the other hand, can have any number of children, and there is no way to obtain the process Ids of its children. If the parent wants to keep track of the process Ids of all its children, it must record the return values form ***fork.***

exec function :

The only way in which an executable program file on disk is executed by Unix is for an existing process to call one of the six ***exec*** functions. ***exec*** replaces the current process image with

the new program file and this new program normally starts at the main function. The process ID does not change. The process that calls the ***exec*** is the *calling process* and the newly executed program as the *new program*.

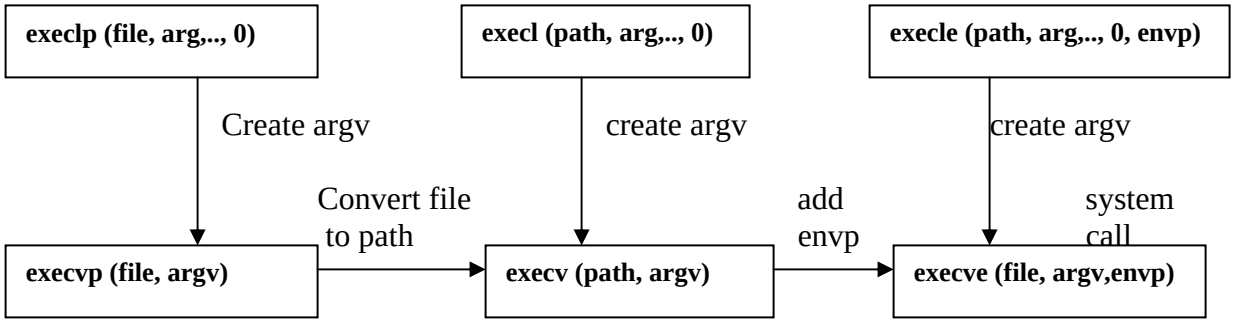
The differences in the six ***exec*** functions are:

a. whether the program file to execute is specified by a file name or a pathname.
b. Whether the arguments to the new program are listed one by one or reference through an array of pointers, and
c. Whether the environment of the calling process is passed to the new program or whether a new environment is specified.

```
#include <unistd.h>
int execl  (const char *pathname, const char arg 0, …/ (char *) 0 */);
int execv  (const char *pathname, char  *const  argv[ ]);
int execle (const char *pathname, const char *arg 0, ./ * (char *)0,char *const envp[] */);
int execve  (const char *pathname, char  *const arg [], char  *const envp[]);
int execlp  (const char *filename, const char arg 0, …/ (char *) 0 */);
int execvp  (const char *filename, char *const argv[]);
```

These functions return to the caller only if an error occurs. Otherwise control passes to the start of the new program, normally the main function.

The relationship among these six functions is shown in the following figure . Normally only ***execve*** is a system call within the kernal and the other five are library functions that call ***execve***.



1. The three functions in the top row 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 in the second row have an ***argv*** array containing the pointers to the argument strings. This ***argv*** array must contain a null pointer to specify its end, since a count is not specified.

1. The two functions in the left column specify a ***filename*** argument. This is converted into a ***pathname*** using current PATH environment variable. IF the **execlp (file, arg,.., 0)**
2. filename argument to ***execlp or execvp*** contains a slash (/) anywhaere in the string, the PATH variable is not used. The four functions in the right two columns specify a fully qualified ***pathname*** arguments.

3. The four functions in the left two column do no 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 in the right column specify an explicit environment list. The ***envp*** array of pointers must be terminated by a null pointer.

Concurrent Servers:

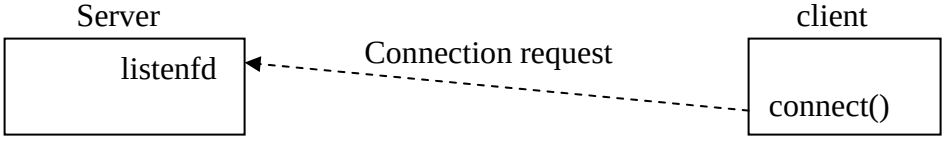
A server that handles a simple program such as daytime server is a ***iterative*** server. But when the client request can take longer to service, the server should not be tied upto a single client. The server must be capable of handling multiple clients at the same time. The simplest way to write a ***concurren***t server under Unix is to ***fork*** a child process to handle each client. Following program shows the **typical concurrent server**.

```
pid_t   pid;
int     listenfd,  connfd;

listfd = socket ( , , , );  /*fill in sockaddr_in with server's well known port*/
bind (listenfd, …);
listen (listenfd, LISTENQ);

for  (  ;  ; ) {
        connfd = accept (listenfd, …);
                if (  (pid = fork())== 0) {
                        close (listenfd);  /* child closed listening socket */
                        doit (connfd);   /* process the request */
                        close ( connfd);  /* done with the client*/
                        exit (0);         /* child terminates*/
                        }
close (connfd);         /* parent closes connected socket*/
}
```

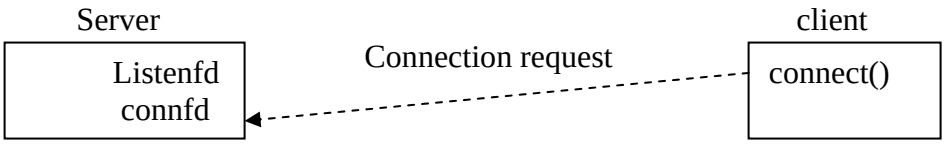
When a connection is established , ***accept*** returns, the server calls ***fork,*** and then the child process services the client (on ***connfd,*** the connected socket) and the parent process waits for another connection (on ***listenfd,*** the listening socket). The parent closes the connected socket since the child handles this new client.

In the above program, the function ***doit*** does whatever is required to service the client. When this functions returns, we explicitly ***close*** the connected socket in the child. This is not required since the next statement calls ***exit,*** and part of process termination is closing all open descriptors by the kernal. Whether to include this explicit call to ***close*** or not is a matter of personal programming taste.

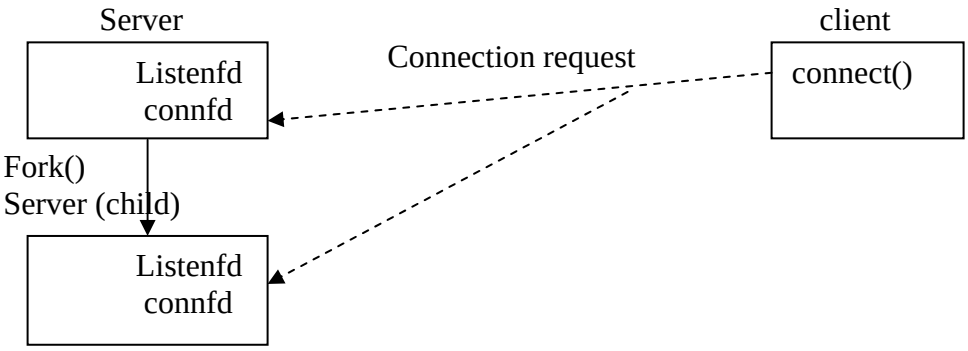
The connection scene in case of concurrent server is shown below:



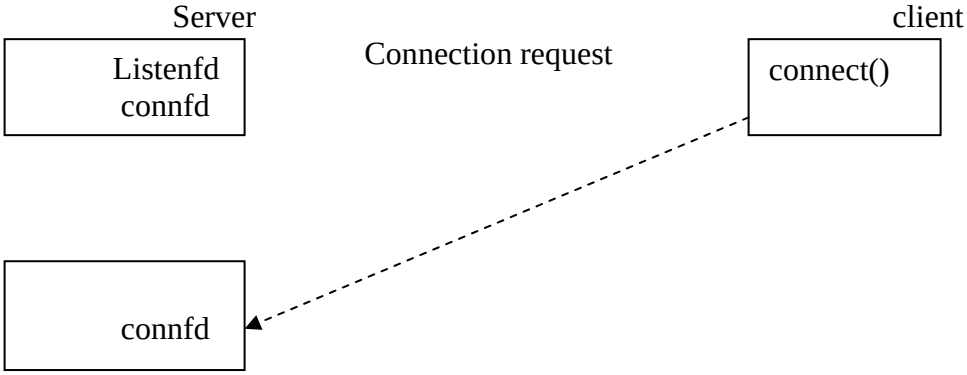
Status of client –server before call to accept().



Status of client server after return from accept().



Status of client server after fork returns.



Status of client server after parent and child close appropriate socket.

close() The normal Unix close() is also 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. Thst is , it can not be used as an argument to read or write.

getsockname () and getpeername():

These two functions return either the local protocol address associated with a socket or the foreign address associated with a socket.

#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *localadddr, socklen_t *addlen);

int getpeername(int sockfd, struct sockaddr *peeradddr, socklen_t *addlen);

both return 0 if OK and –1 on error.

These functions are required for the following reasons.

a. After connect successfully returns a TCP client that does not call **bind(), getsocketname()** returns the local IP address and local port number assigned to the connection by the kernel
b. After calling bind with a port number of 0, **getsockname()** returns the local port number that was assigned
c. When the server is ***exce***ed by the process that calls **accept(),** the only way the server can obtain the identity of the client is to call **getpeername().**

Model Question:

Short Question:

1. Draw Internet Protocol suit.
2. List the characteristics of UDP
3. List the characteristics of TCP
4. What do understand by Three Way Handshake in a TCP connection
5. List the packets that exchanged while TCP connection terminates.
6. Briefly describe Port numbers and its categories.
7. Define IPv4 Socket Address Structure
8. Define IPv6 Socket Address Structure
9. What are generic socket address structure?
10. What are Byte Order Function?
11. What are Byte Manipulation Functions?
12. What are Address Conversion functions?
13. Differentiate between IPv4 and IPv6 address system

Long Questions:

1. What do you understand by system calls used with sockets. Briefly describe any six of them.
2. Draw and briefly explain the state transition diagram of TCP.
3. Briefly describe concurrent servers.