LECTURE NOTES

ON

IT1352 – NETWORK PROGRAMMING AND MANAGEMENT

Ms. P. SIVAKAMI M.E.,

LECTURER

DEPT OF INFORMATION TECHNOLOGY

NPRCET

IT1352 – NETWORK PROGRAMMING AND MANAGEMENT

UNIT I ELEMENTARY TCP SOCKETS

Introduction to socket programming – Overview of TCP / IP protocols – Introduction to sockets – Socket address structures – Byte ordering functions – Address conversion functions – Elementary TCP sockets – Socket – Connect – Bind – Listen – Accept – Read – Write – Close functions – Iterative server – Concurrent server.

UNIT II APPLICATION DEVELOPMENT

TCP echo server – TCP echo client – POSIX signal handling – Server with multiple clients – Boundary conditions– Server process crashes– Server host crashes – Server crashes and reboots – Server shutdown – I/O multiplexing – I/O models – Select function – Shutdown function – TCP echo server (with multiplexing) – Poll function – TCP echo client (with multiplexing)

UNIT III SOCKET OPTIONS, ELEMENTARY UDP SOC SOCKETS Socket options – Getsocket and setsocket functions – Generic socket options – IP socket options – ICMP socket options – TCP socket options – Elementary UDP sockets – UDP echo server – UDP echo client – Multiplexing TCP and UDP sockets – Domain Name System – Gethostbyname function – IPV6 support in DNS – Gethostbyadr function – Getservbyname and getservbyport functions.

UNIT IV ADVANCED SOCKETS

IPV4 and IPV6 interoperability – Threaded servers – Thread creation and termination– TCP echo server using threads – Mutexes – Condition variables – Raw sockets – Raw socket creation – Raw socket output – Raw socket input – Ping program – Trace route program.

UNIT V SIMPLE NETWORK MANAGEMENT

SNMP network management concepts – SNMP management information – Standard MIB's – SNMP V1 protocol and practical issues – Introduction to RMON, SNMP V2 and SNMP V3.

TEXT OOKS

1. W. Richard Stevens, "Unix Network Programming Vol - I", 2nd Edition, Prentice Hall of India / Pearson Education, 1998.
2. William Stallings, "SNMP, SNMPV2, SNMPV3 and RMON 1 and 2", 3rd Edition, Addison Wesley, 1999.

REFERENCE

1. D. E. Comer, "Internetworking with TCP/IP Vol - III", (BSD Sockets Version), 2nd Edition, Prentice Hall of India, 2003.

UNIT I

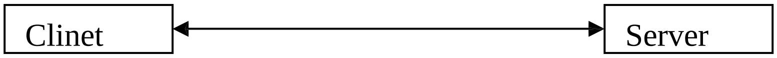
ELEMENTARY TCP SOCKETS

Introduction to socket programming – Overview of TCP / IP protocols – Introduction to sockets – Socket address structures – Byte ordering functions – Address conversion functions – Elementary TCP sockets – Socket – Connect – Bind – Listen – Accept – Read – Write – Close functions – Iterative server – Concurrent server.

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 with in the same Ethernet and the communication when LAN connected through WAN is shown below.

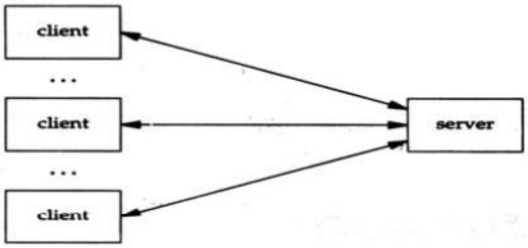


Figure 1.2 Server handling multiple clients at the same time.

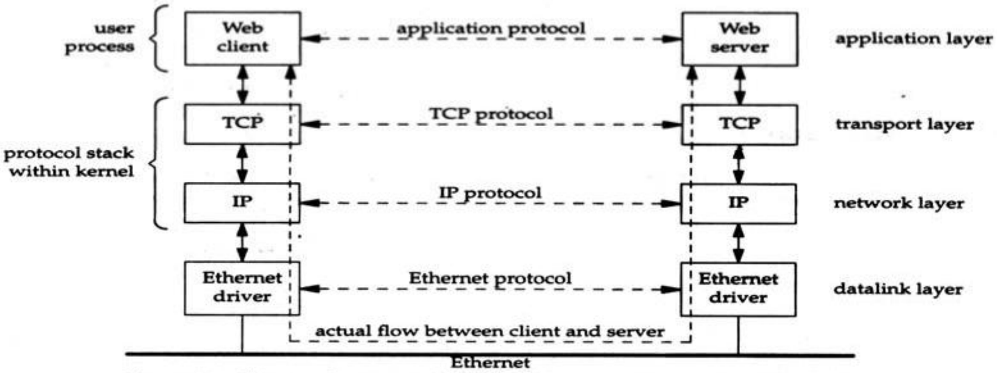


Figure 1.3 Client and server on the same Ethernet communicating using TCP.

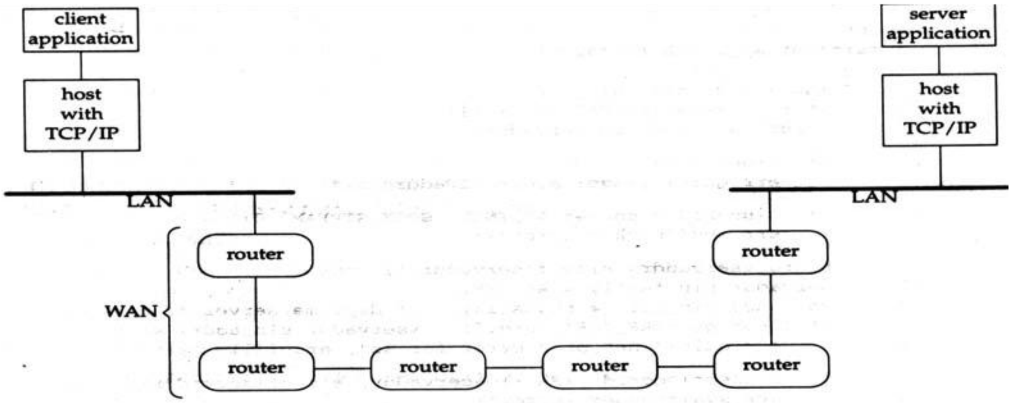


Figure 1.4 Client and server on different LANs connected through a WAN.

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.

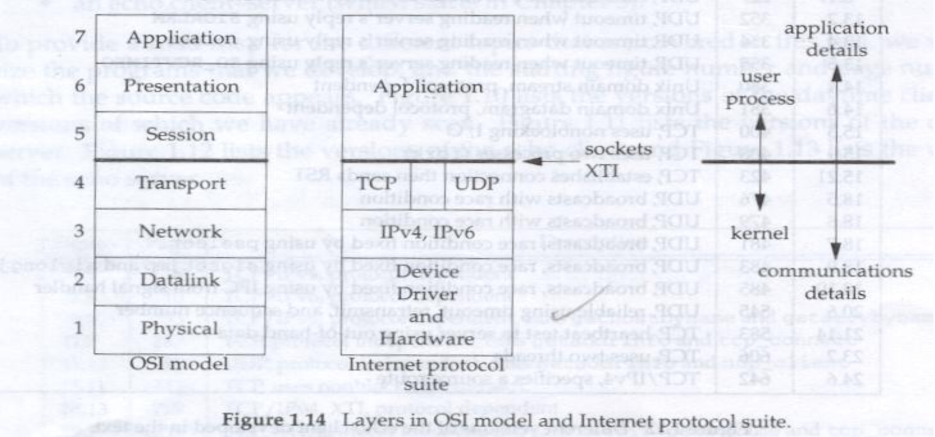


Figure 1.14 Layers in OSI model and Internet protocol suite.

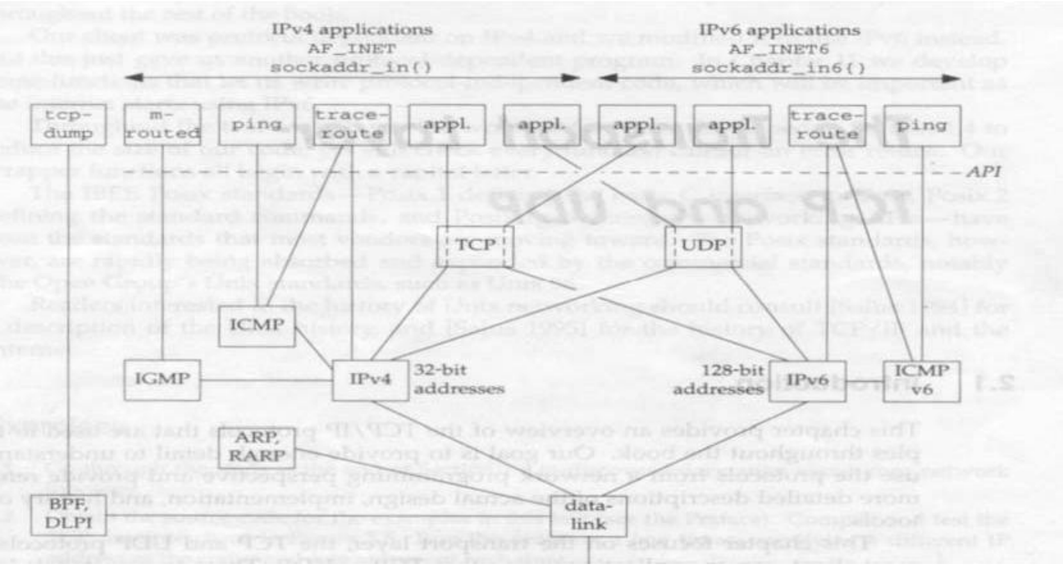
The upper three layers of OSI model are combined into a single layer called the application. This is the Web Client (Browser), Telnet client, the Web Server, FTP server or whatever application we are using. The transport layer that are chosen are TCP and UDP. As shown in the figure, it is possible to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

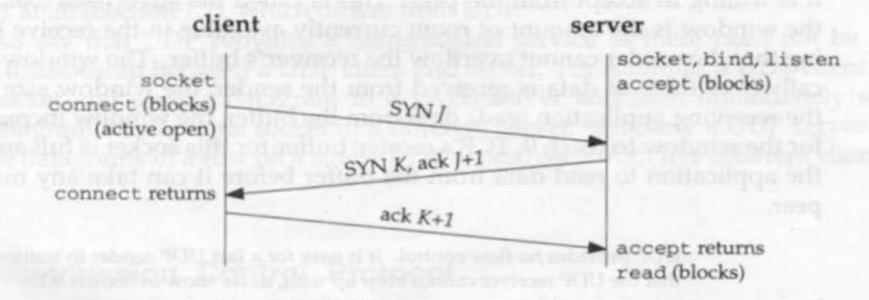


Figure 2.2 TCP three-way handshake.

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.

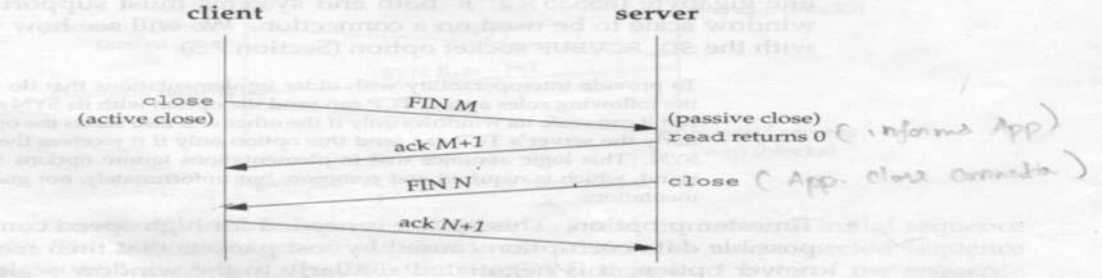


Figure 2.3 Packets exchanged when a TCP connection is closed.

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.

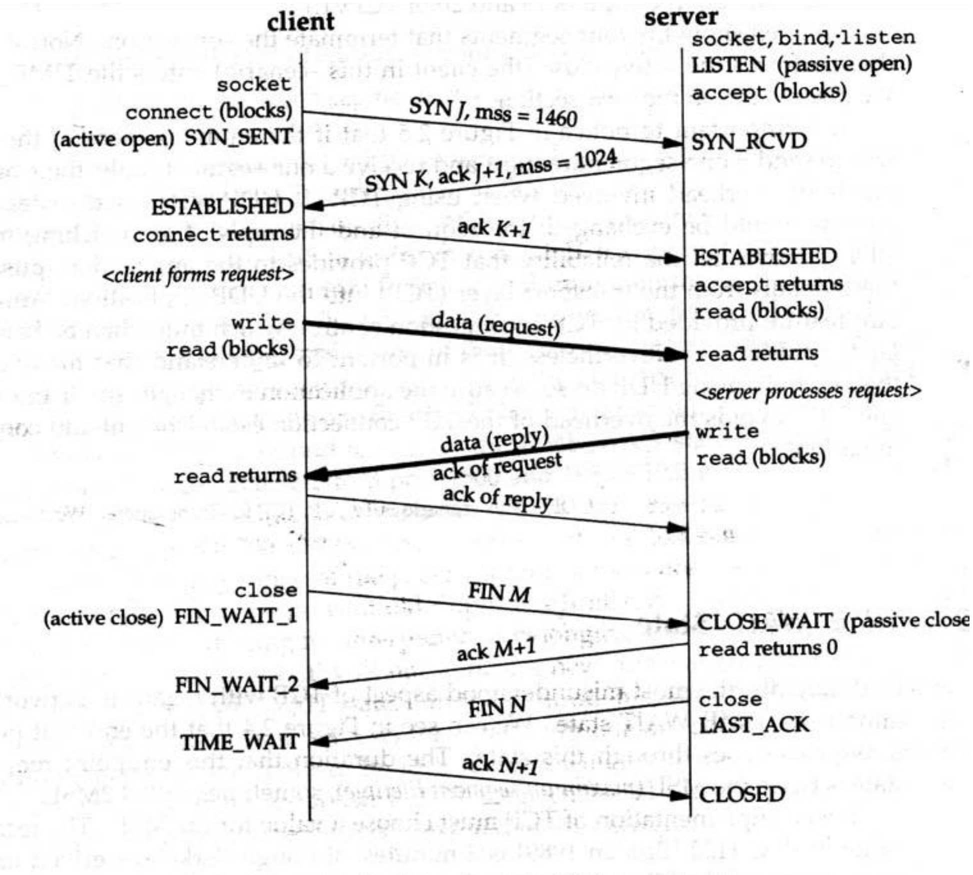


Figure 2.5 Packet exchange for TCP connection.

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.

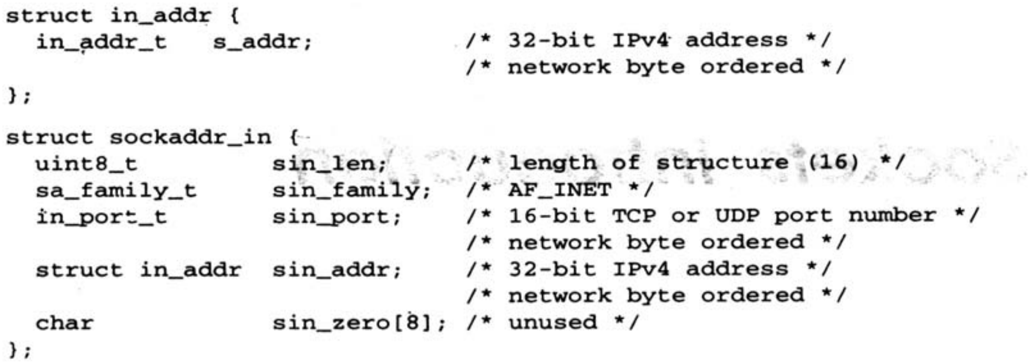
Socket Pair: The two values that identify each endpoint, an IP address and a port number, are called a socket. The socket pair for a TCP connection is the 4 tuple that defined the two end points of the connection: the local IP address, local TCP port, foreign IP address, and foreign TCP port.

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));
```

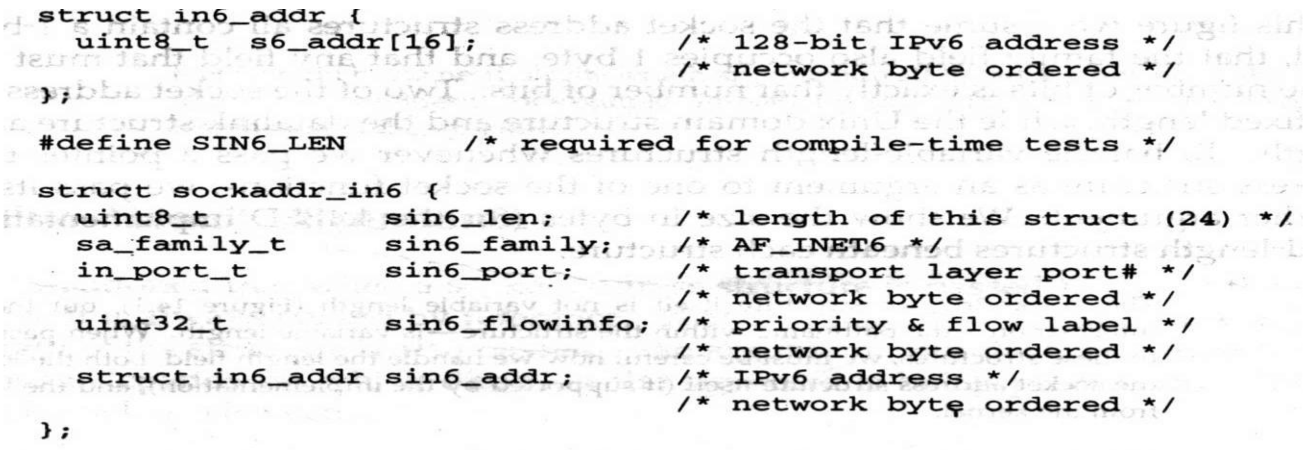
If we omit the cast, the C compiler generates a warning of the form incompatible pointer type.

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.

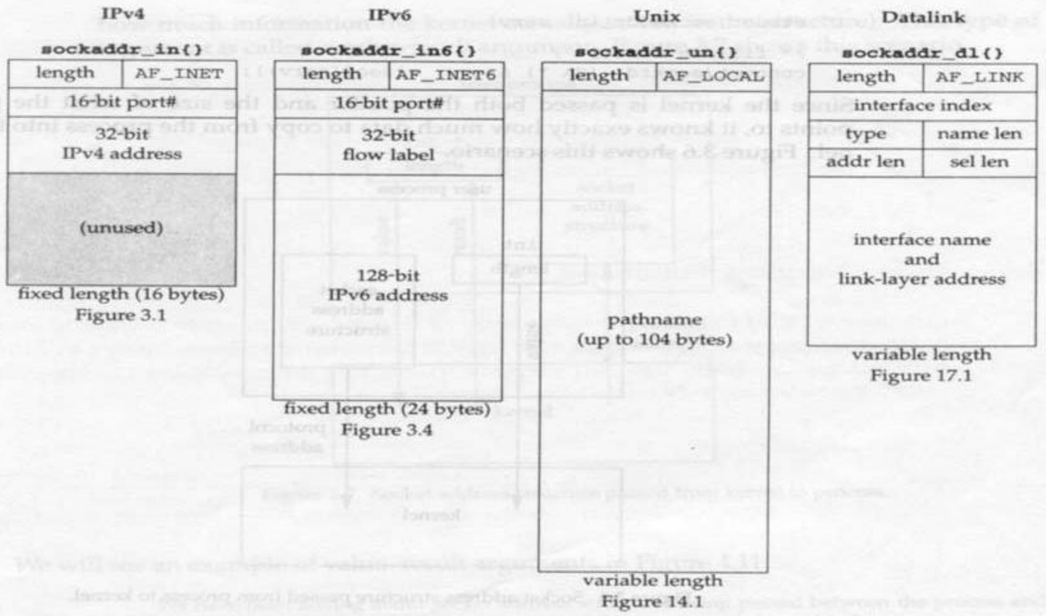


Figure 3.5 Comparison of various socket address structures.

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:

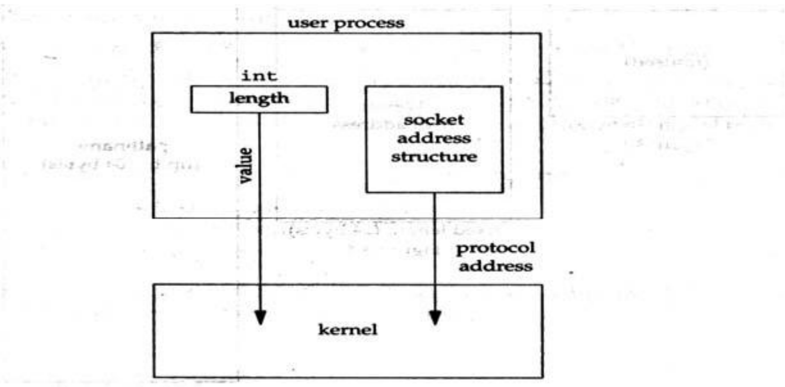


Figure 3.6 Socket address structure passed from process to kernel.

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**.

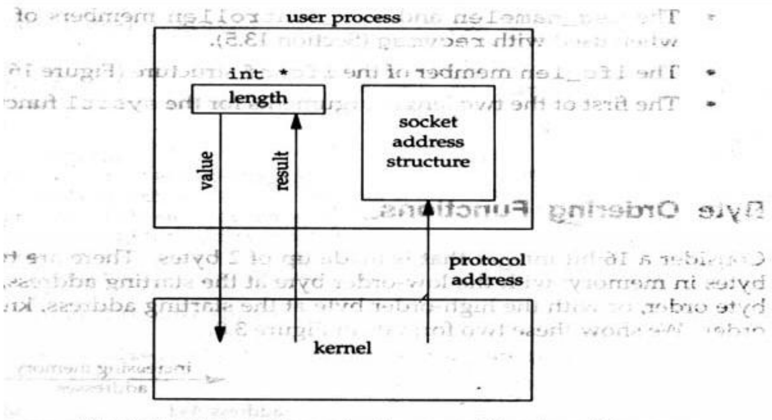


Figure 3.7 Socket address structure passed from kernel to process.

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.

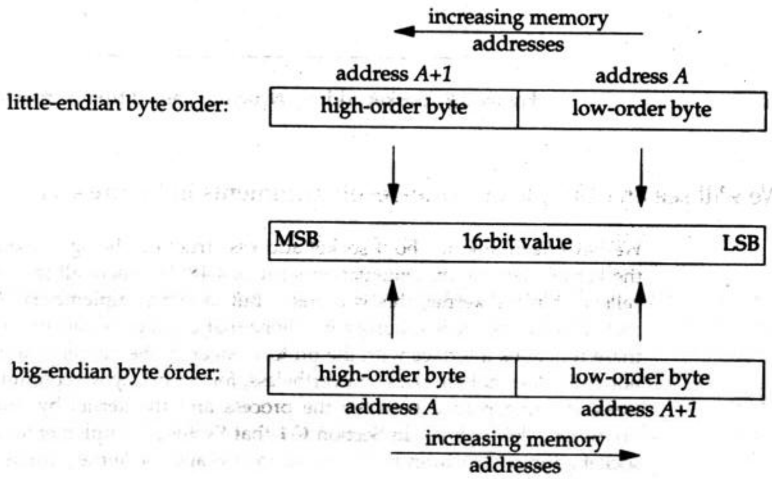


Figure 3.8 Little-endian byte order and big-endian byte order for a 16-bit integer.

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:

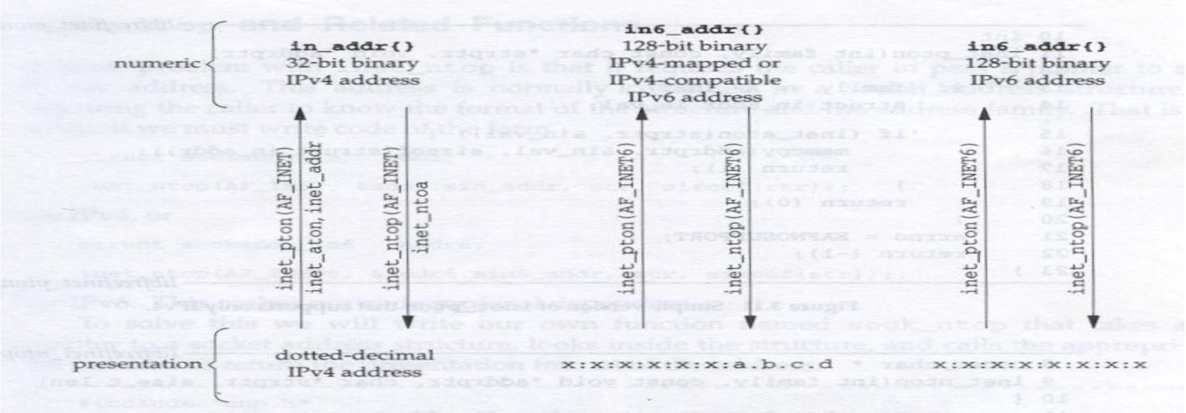


Figure 3.10 Summary of address conversion functions.

#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.

<u>System calls used with sockets:</u>

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.

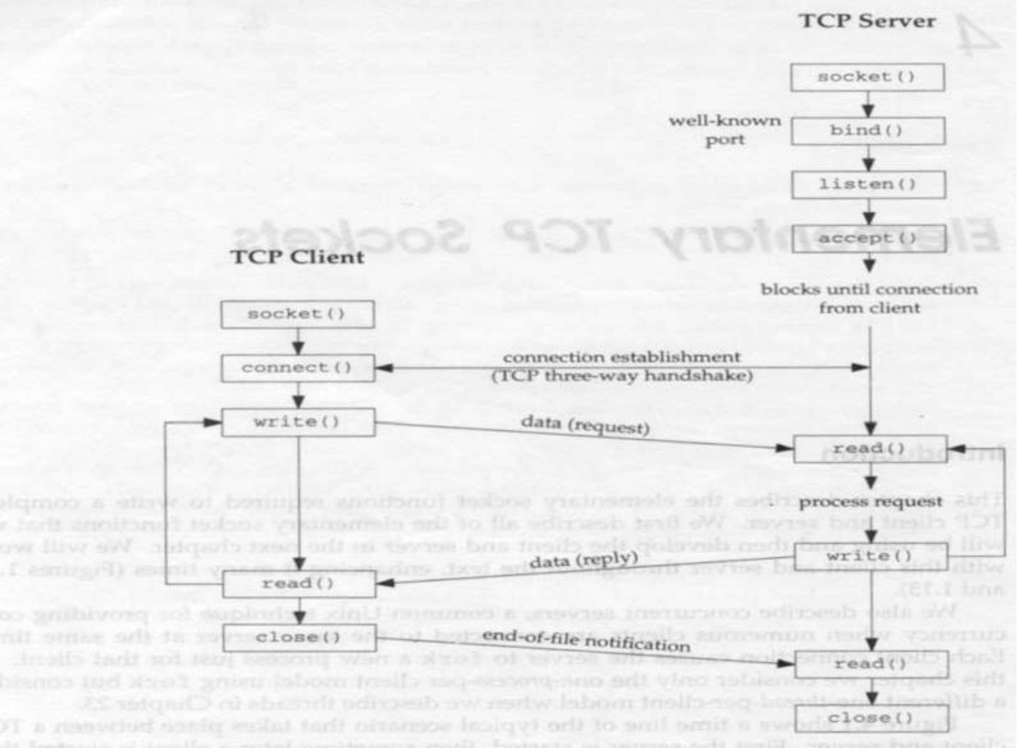


Figure 4.1 Socket functions for elementary TCP client–server.

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.

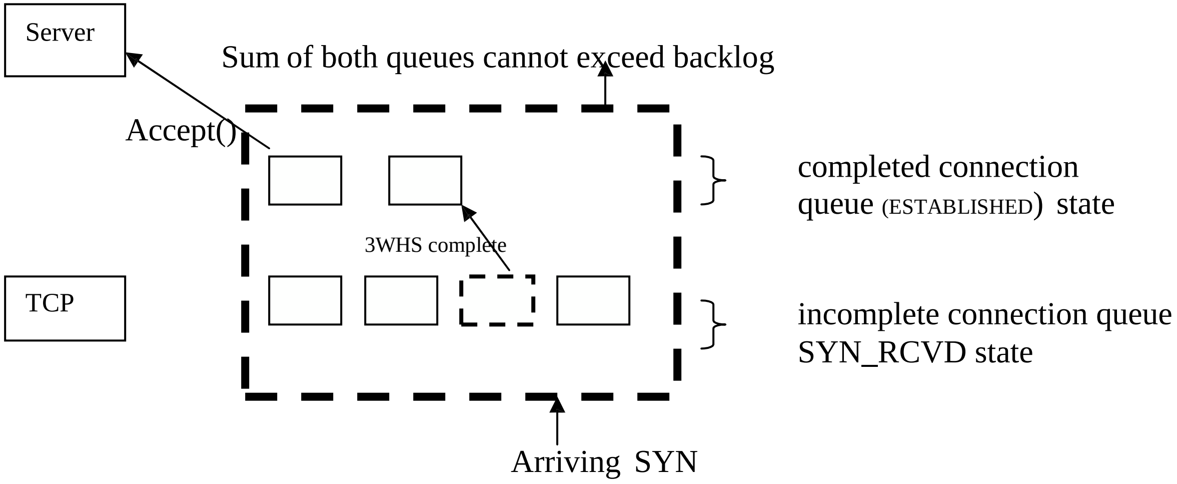
This function is normally called after both the socket and bind functions and must be called before calling the accept function.

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.


Server
Sum of both queues cannot exceed backlog
Accept()
completed connection
queue (ESTABLISHED) state
3WHS complete
TCP
incomplete connection queue
SYN_RCVD state
Arriving SYN

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.

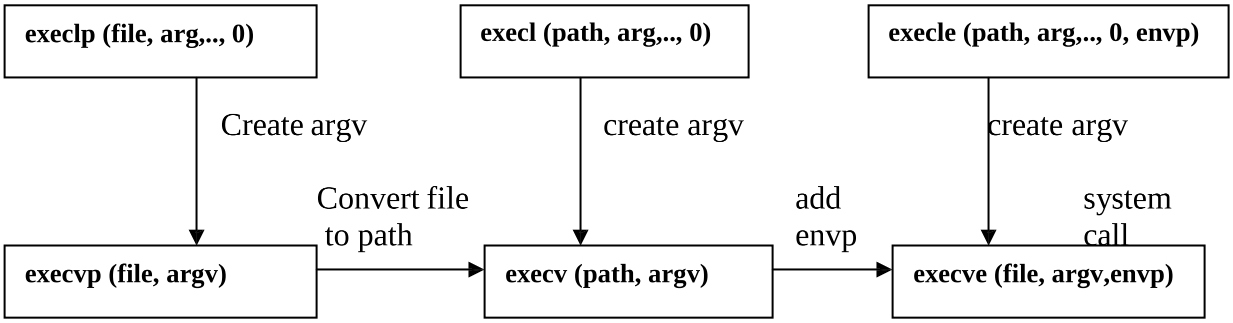
b. Whether the arguments to the new program are listed one by one or reference through an array of pointers, and

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/Iterative server:

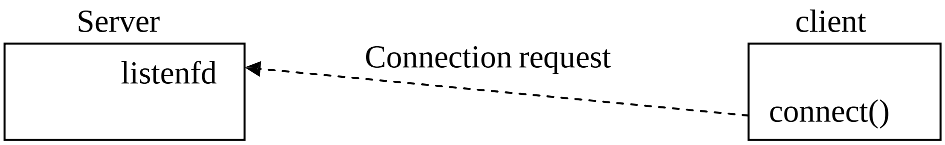
A server that handles a simple program such as daytime server is a ***iterative*** server. But when the client request can take longer to service, the server should not be tied upto a single client. The server must be capable of handling multiple clients at the same time. The simplest way to write a ***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*/
}
```

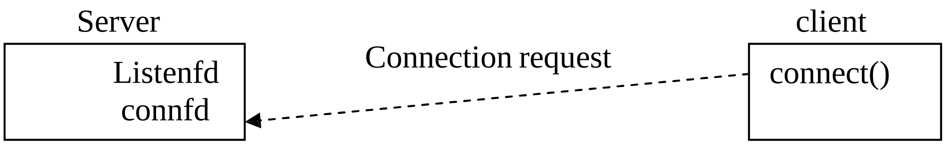
When a connection is established , ***accept*** returns, the server calls ***fork,*** and then the child process services the client (on ***connfd,*** the connected socket) and the parent process waits for another connection (on ***listenfd,*** the listening socket). The parent closes the connected socket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

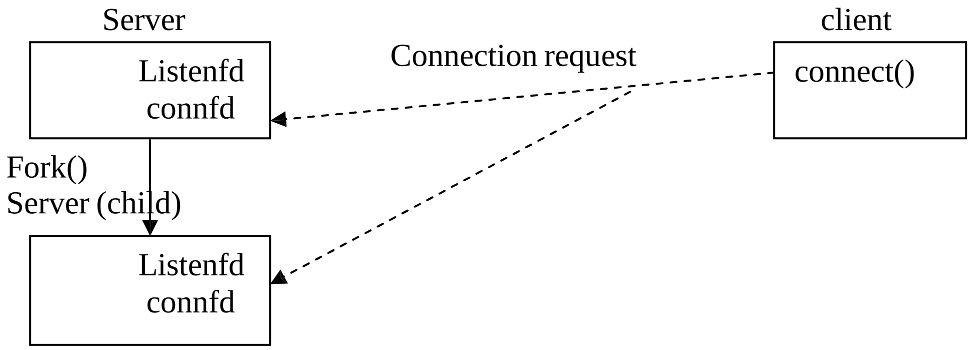
The connection scene in case of concurrent server is shown below:



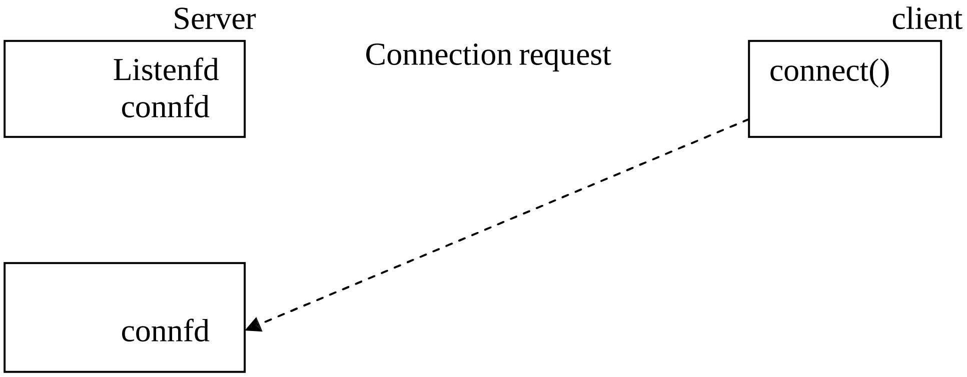
Status of client –server before call to accept().



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().**

Short Questions

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

Big Questions:

1. What do you understand by system calls used with sockets. Briefly describe any six of them(16).
2. Draw and briefly explain the state transition diagram of TCP.(16)
3. Briefly describe concurrent servers.(16)

UNIT II

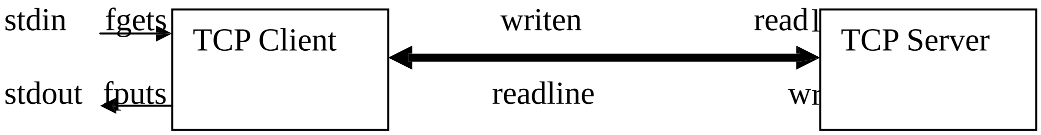
APPLICATION DEVELOPMENT

TCP echo server – TCP echo client – POSIX signal handling – Server with multiple clients – Boundary conditions– Server process crashes– Server host crashes – Server crashes and reboots – Server shutdown – I/O multiplexing – I/O models – Select function – Shutdown function – TCP echo server (with multiplexing) – Poll function – TCP echo client (with multiplexing)

UNIT-2

TCP CLIENT - SERVER COMMUNICATION

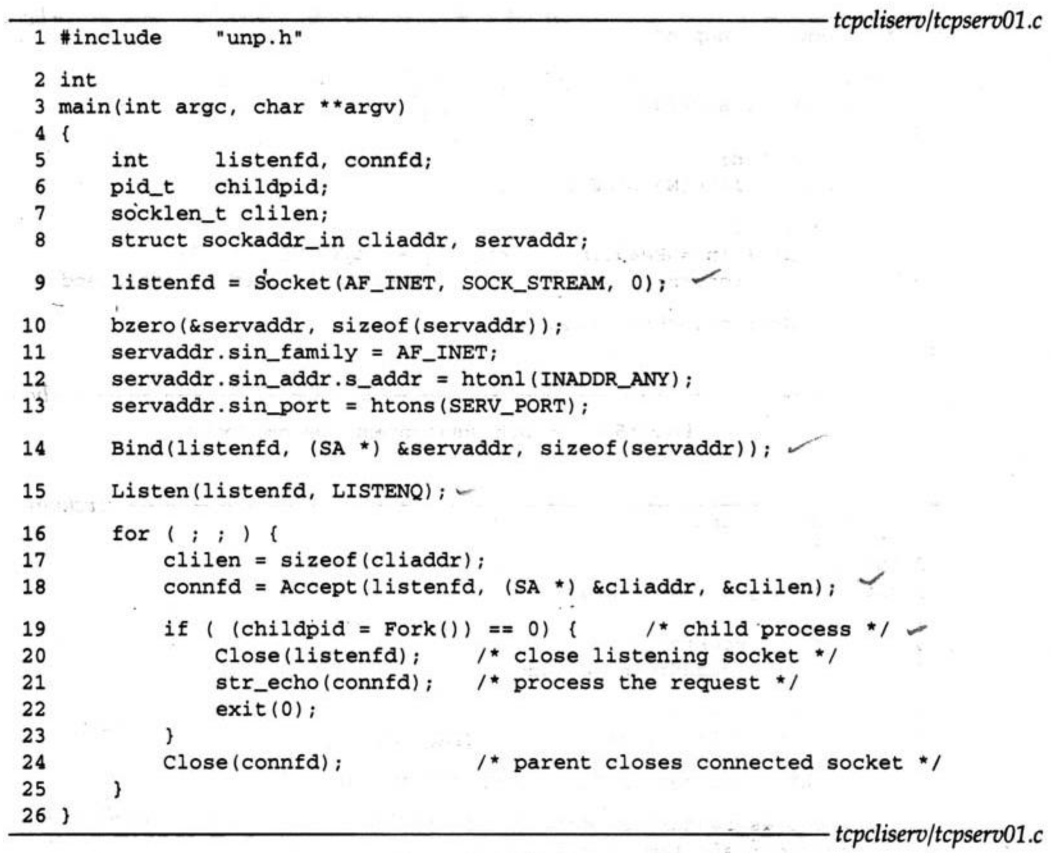
1. Client – Server communication involves reading of text (character or text) from the client and writing the same into the server and server reading text and client writing the same. Following picture depicts the same.



Simple Echo client server.

Functions ***fgets() and fputs()*** are from standard I/O library. And ***writen()*** ***and readline()*** are function created by the W Richard Stevans (WRS) (code given in the section 3.9)

The communication between client and server is understood by Echo client and Server. The Following code corresponds to Server side program.



```
                                                          tcpcliserv/tcpserv01.c
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd;
 6     pid_t   childpid;
 7     socklen_t clilen;
 8     struct sockaddr_in cliaddr, servaddr;

 9     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

10     bzero(&servaddr, sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
13     servaddr.sin_port = htons(SERV_PORT);

14     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

15     Listen(listenfd, LISTENQ);

16     for ( ; ; ) {
17         clilen = sizeof(cliaddr);
18         connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

19         if ( (childpid = Fork()) == 0) {    /* child process */
20             Close(listenfd);    /* close listening socket */
21             str_echo(connfd);   /* process the request */
22             exit(0);
23         }
24         Close(connfd);          /* parent closes connected socket */
25     }
26 }
                                                          tcpcliserv/tcpserv01.c
```

Figure 5.2 TCP echo server.

Line 1: It is the header created by the WRS which encapsulates a large number of header that are required for the functions that are referred.
Line 2 – 3: This the definition of the ***main()*** with command line arguments.
Line 5-8 : These are variable declarations of types that are used.
Line 9 : It is the system call to the **socket** function that returns a descriptor of type int.- in this case it is named as listenfd. The arguments are family type, stream type and protocol argument – normally 0)
Line 10: the function ***bzero()*** sets the address space to zero.

Line 11-12: Sets the internet socket address to wild card address and the server port to the number defined in **SERV_PORT** which is 9877 (specified by WRS). It is an intimation that the server is ready to accept a connection destined for any local interface in case the system is multi homed.

Line 14 : ***bind ()*** function binds the address specified by the address structure to the socket.

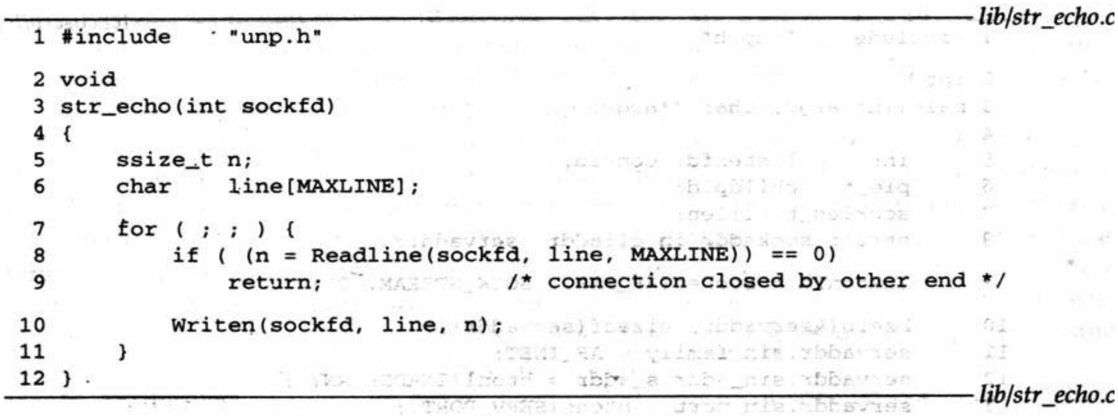
Line 15: The socket is converted into listening socket by the call to the listen()function

Line 17-18: The server blocks in the call to accept, waiting for a client connection to complete.

Line 19 – 24: For each client, ***fork()*** spawns a child and the child handles the new client. The child closes the listening socket and the parent closes the connected socket The child then calls ***str_echo ()*** to handle the client

2. TCP Echo Server : **str_echo** function

This is shown in the following figure. It shows the server processing for each client: reading the lines from the client and echoing them back to the client.



lib/str_echo.c

```
 1 #include    "unp.h"

 2 void
 3 str_echo(int sockfd)
 4 {
 5     ssize_t n;
 6     char    line[MAXLINE];

 7     for ( ; ; ) {
 8         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
 9             return;          /* connection closed by other end */

10         Writen(sockfd, line, n);
11     }
12 }
```

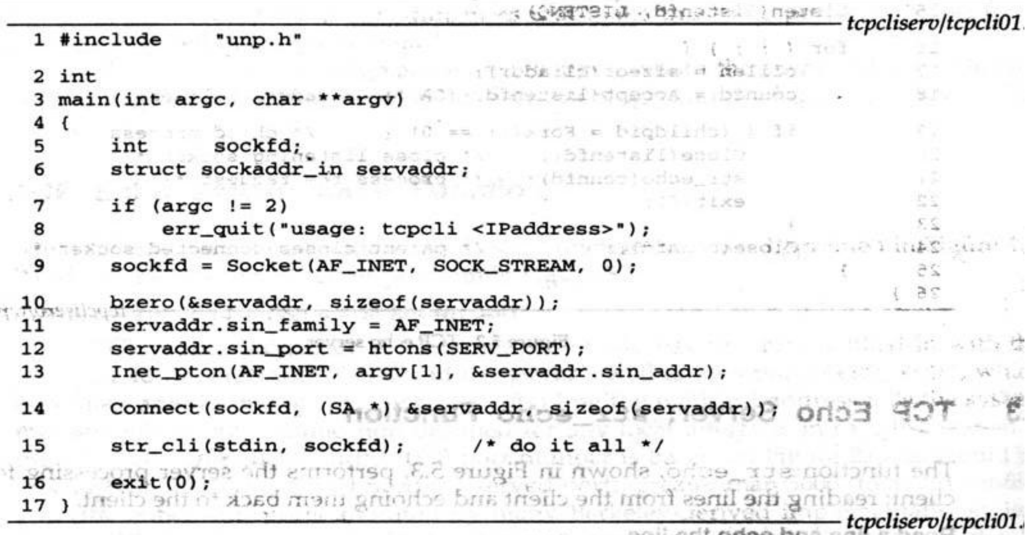
lib/str_echo.c

Figure 5.3 str_echo function: echo lines on a socket.

Line 6: MAXLINE is specified as constant of 4096 characters.

Line 7-11: ***readline*** reads the next line from the socket and the line is echoed back to the client by ***writen*** If the client closes the connection, the recept of client's FIN causes the child's readline to return 0. This causes the str_echo function to return which terminates the child.

TCP Echo Client : Main Function : Following code shows the TCP client main function.



tcpcliserv/tcpcli01

```
#include    "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd;
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit("usage: tcpcli <IPaddress>");
    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

    str_cli(stdin, sockfd);     /* do it all */

    exit(0);
}
```

tcpcliserv/tcpcli01.

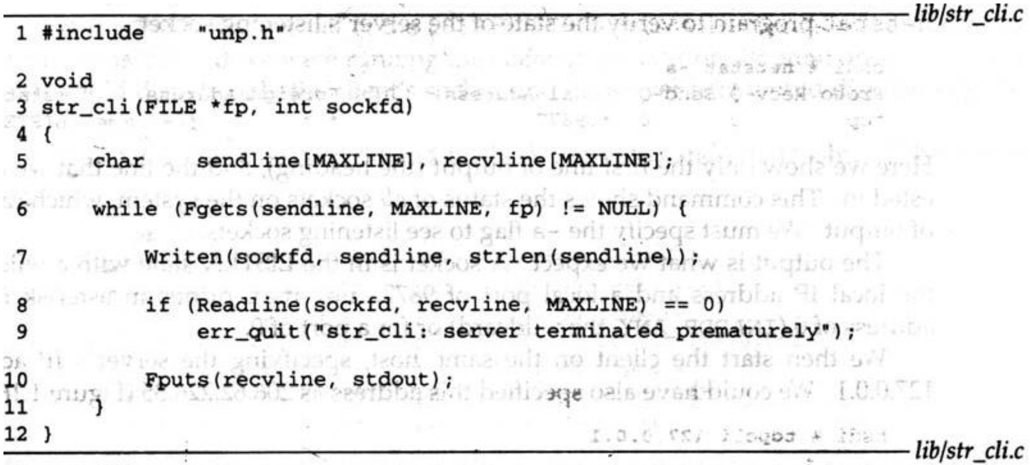
Figure 5.4 TCP echo client.

Line 9 – 13: A TCP socket is created and an Internal socket address structure is filled in with the server's IP address and port number. We take the server's IP address from the command line argument and the server's well known port (SERV_PORT) from the header.

Line 13: The function inet_pton () converts the argument received at the command line from presentation to numeric value and stores the same at the address specified as the third arguments.

Line 14 –1 5: Connection function establishes the connection with the server. The function str_cli () than handles the client processing.

TCP Echo Client : str _ cli function:



lib/str_cli.c

```
1 #include    "unp.h"

2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     char    sendline[MAXLINE], recvline[MAXLINE];

6     while (Fgets(sendline, MAXLINE, fp) != NULL) {

7         Writen(sockfd, sendline, strlen(sendline));

8         if (Readline(sockfd, recvline, MAXLINE) == 0)
9             err_quit("str_cli: server terminated prematurely");

10        Fputs(recvline, stdout);
11    }
12 }
```

lib/str_cli.c

Figure 5.5 str_cli function: client processing loop.

Above function handles the client processing loop. That is read a line of text from standard input, write it to the server, read back the server's echo of the line and output the echoed line to standard input.

Line 6-7 : ***fgets*** reads a line of text and ***writen*** sends the line to the server.
Line 8 – 10: readline reads the line echoed back from the server and fputs writes it to the standard output.

Signals – Introduction.

A signal is a message (an integer) sent to a process. The receiving process can try to ignore the signal or call a routine (a so-called *signal handler*). After returning from the *signal handler*, the receiving process will resume execution at the point at which it was interrupted. The system-calls to deal with signals vary between one Unix version and another

The following conditions can generate a signal(1):

- When user presses terminal keys, the terminal will generate a signal. For example, when the user breaks a program by CTRL + C key pair.
- Hardware exceptions can generate signals, too: Division by 0, invalid memory reference. Inexperienced programmers often get SIGSEGV (Segmentation Violation signal) because of an invalid address in a pointer.
- Processes can send signals to themselves by using kill(2) system call (If permissions allow).
- Kernel can generate signal to inform processes when something happens. For example, SIGPIPE will be generated when a process writes to a pipe which has been closed by the reader.

Signals can be sent using the `kill()` routine. The `signal()` and `sigaction()` routines are used to control how a process should respond to a signal.

Posix Signal Handling :

- Every signal has a ***disposition***, which is called the ***action*** associated with the signal. The disposition is set by calling the ***sigaction*** **function**.
- We have three choices for the disposition:

a) Whenever a specific signal occurs, a specific function can be provided. This function is called ***signal handler*** and the action is called ***catching*** the signal.

- The two signal SIGKILL and SIGSTOP cannot be caught – this is an exception.
- The function is called with a single integer argument that is the signal number and the function returns nothing as shown below:

```
const struct sigaction act;
 sigaction (SIGCHLD, &act, NULL)
```

- Calling *sigaction* and specifying a function to be called when the signal occurs is all that is required to catch the signal.

o For few signal like SIGIO, SIGPOLL, and SIGURG etc additional actions on the part of the process is required to catch the signal.

b) A signal a can be **ignore**d by setting its disposition to **SIG_IGN.** Again the two signals SIGKILL and SIGSTOP are exceptions.

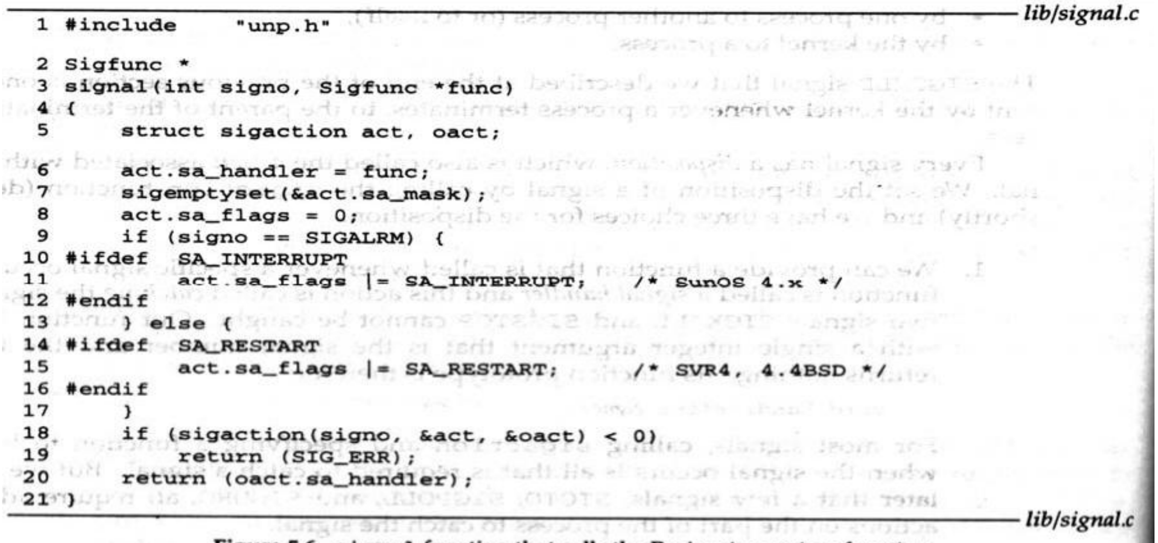
c) We can set the **default** disposition for a signal by setting its disposition to **SIG_DFL.** The default is normally to terminate a process on the receipt of a signal, with certain signal also generating a core image of the process in its current working directory. The signals whose default disposition is to be ignored are : SIGCHLD AND SIGURG(sent on arrival of out of band data.)

with appropriate settings in the `sigaction` structure you can control the current process's response to receiving a `SIGCHLD` signal. As well as setting a signal handler, other behaviours can be set. If

- `act.sa_handler` is `SIG_DFL` then the default behaviour will be restored
- `act.sa_handler` is `SIG_IGN` then the signal will be ignored if possible (`SIGSTOP` and `SIGKILL` can't be ignored)
- `act.sa_flags` is `SA_NOCLDSTOP` - `SIGCHLD` won't be generated when children stop.
- `act.sa_flags` is `SA_NOCLDWAIT` - child processes of the calling process will not be transformed into zombie processes when they terminate.

signal Function :

Posix way to establish the disposition of a signal is to call the **sigaction** function. However this is complicated at one argument to the function is a structure that we must allocate and fill in. An easier way to set the disposition for a signal is to call the **signal function.** The first argument is the **signal name** and the second arguments is the **pointer to a function** or one of the constants **SIG_IGN** or **SIG_DFE.** Normally, the define our own **signal function** that just calls the **Posix sigaction.**



lib/signal.c

```
 1 #include    "unp.h"

 2 Sigfunc *
 3 signal(int signo, Sigfunc *func)
 4 {
 5     struct sigaction act, oact;

 6     act.sa_handler = func;
 7     sigemptyset(&act.sa_mask);
 8     act.sa_flags = 0;
 9     if (signo == SIGALRM) {
10 #ifdef  SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;   /* SunOS 4.x */
12 #endif
13     } else {
14 #ifdef  SA_RESTART
15         act.sa_flags |= SA_RESTART;     /* SVR4, 4.4BSD */
16 #endif
17     }
18     if (sigaction(signo, &act, &oact) < 0)
19         return (SIG_ERR);
20     return (oact.sa_handler);
21 }
```

lib/signal.c

Figure 5.6 signal function that calls the Posix sigaction function.

line 2-3 call to the function when a signal occurs. It has pointer to signal handling function as the second argument

Line 6: Set Handler : The **sa_handler** member of the sigaction structure is set to the func argument
Line 7: Set signal mask to handler: POSIX allows us to specify a set of signals that will be blocked when our signal handler is called. Any signal that is blocked cannot be delivered to the process. We set the **sa_mask** member to the empty set, which means that no additional signals are blocked while our signal handler is running Posix guarantees that he signal being caught is always blocked while its handler is executing

Line 8 – 17: An optional flag SA_RESTART, if it is set, a system call interrupted by this signal will automatically restarted by the kernal.

Line 18 – 20: The function sigaction is called and then return the old action for the signal as the return value of the signal function.

Posix Signal semantics:

1. Once a signal handler is installed, it remains installed.
2. While a signal handler is executing, the signal being delivered is blocked. Further any additional signals that were specified in the sa_mask signal set passed to sigaction when the handler was installed are also blocked . When the **sa_mask** is set to empty set, no additional signals are blocked other than the signal being caught.
3. If a signal is being generated more times while it is blocked, it is normally delivered only one time after the signal is unblocked.
4. It is possible to selectively block and unblock a set of signals using the ***sigprocmask*** function. This protects a critical region of code by providing certain signals from being caught while the region of code is executing.

Handling SIGCHLD signals:

A zombie is a process that ha terminated whose parent is still running, but has not yet waited for its child processes. This will result in the resources occupied by the terminated process not returned to the system. If there are a lot of zombies in the system the system resources may run out.

The purpose of the zombie state is to maintain information about the child for the parent to fetch at some time later. The information are : the process ID of the child, its termination status and information on the resource utilization of the child (CPU time, memory etc). If a process terminates, and the process has children in the zombie state. The parent process ID of all the zombie children is set to 1 (the init process), which will inherit the children and clean them up.

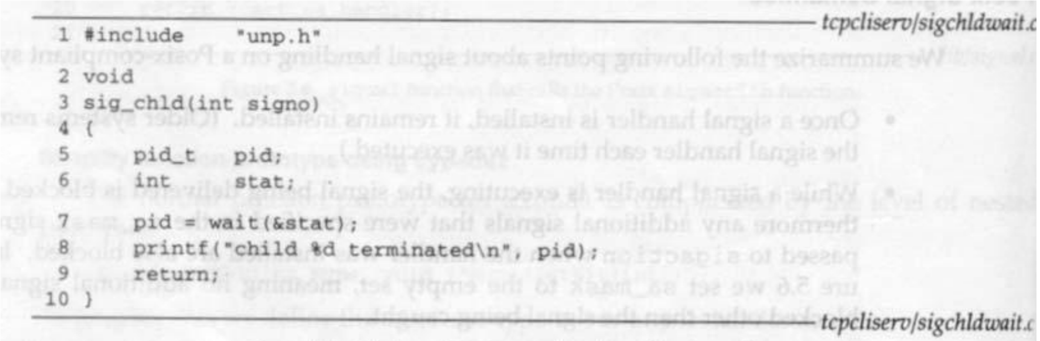
The zombie are not desired to be left. They take up space in the kernal and eventually we can run out of process. Whenever we fork children, we must wait, for them to prevent them from becoming zombies.

When a child terminates, the kernal generates a SGICHLD signal for the parent. The parent process should catch the signal and take an appropriate action. If the signal is not caught, the default action of this signal is to be ignored.

To handle a zombie, the parent process establishes a signal handler to catch SIGCHLD signal. The handler then invokes the function wait() or waitpid () to wait for a terminated child process and free all the system resources occupied by the process.

To do this we establish a signal handler called SIGCHLD and within the handler, we call wait. The signal handler is established by adding function call

Signal(SIGCHLD, sig_chld)



tcpcliserv/sigchldwait.c

```
#include    "unp.h"

void
sig_chld(int signo)
{
        pid_t   pid;
        int     stat;

        pid = wait(&stat);
        printf("child %d terminated\n", pid);
        return;
}
```

tcpcliserv/sigchldwait.c

Figure 5.7 Version of SIGCHLD signal handler that calls wait.

Interrupted System Calls:

There are certain system calls in the Client Server communication that are blocked while waiting for input. It can be seen that in the case of echo client server programme, in the server programme, the function ***accept ()*** is blocked while waiting for the call from a client. This condition is called ***slow system call*** . This can also occur in case of functional calls such as read(), write (), select(), open () etc. Under such condition, when a SIGCHLD is delivered on termination of a process, the sig_chld function executes (the signal handler for SIGCHLD), wait function fetches the child's pid and termination status. The signal handler then returns.

But as the signal was caught by the parent while parent was blocked in a *slow system call (*accept*)*, the kernal causes the accept to return an error of EINTR (interrupted system call). The parent does not handle this error, so it aborts. This is a potential problem which all slow system call (read, write, open, select etc) face whenever they catch any signal. This is undesirable.

In case of Solaris 2.5, the *signal function* provided in the C library does not cause an interrupted system call to be automatically restarted by the kernal. That is SA_RESTART flag that is set in signal function is not set by the signal function in the system library. Where as some other systems like, 4.4 BSD using the library version of signal function, the kernal restarts the interrupted system call and accept does not return an error.

Handling the Interrupted System Calls:

The basic rule that applies here is that when a process is blocked in a slow system call and the process catches a signal and the signal handler returns, the system call can return an error of EINTR. (Error interruption). For portability, when the programmes are written, one must be prepared for slow system calls to return EINTR. Even if some system supports the SA_RESTART flag, not all interrupted system calls may automatically be started. Most Berkeley-derived implementation never automatically restart.

To handle interrupted accept, we change the call to accept (in the server programme) in the following way:

```
for ( ; ; ) {
        clilen = sizeof(cliaddr);
        if ( ( connfd = accept (listenfd, (SA) & cliaddr, & clilen)) < 0) {
                if (errno == EINTR
                        continue;
                else
                        err_sys (“ accept error”);
                }
        }
```

IN this code, the interrupted system call is restarted. This method works for the functions read, write, select and open etc. But there is one function that cannot be restarted by itself. – connect. If this function returns EINTR, we cannot call it again, as doing so will return an immediate error. In this case we must call select to wait for the connection to complete.

Wait () and waitpid () functions:

pid_t wait(int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);

wait and waitpid both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (integer) is returned through statloc pointer. (Termination status are determined when three macros are called that examine the termination status and tell if the child terminated normally, was killed by a signal or is just the job control stopped. Additional macros tell more information on the reason.)

If there are no terminated children for the calling wait, but the process has one or more children that are still executing, then wait blocks until the first of the existing children terminate.

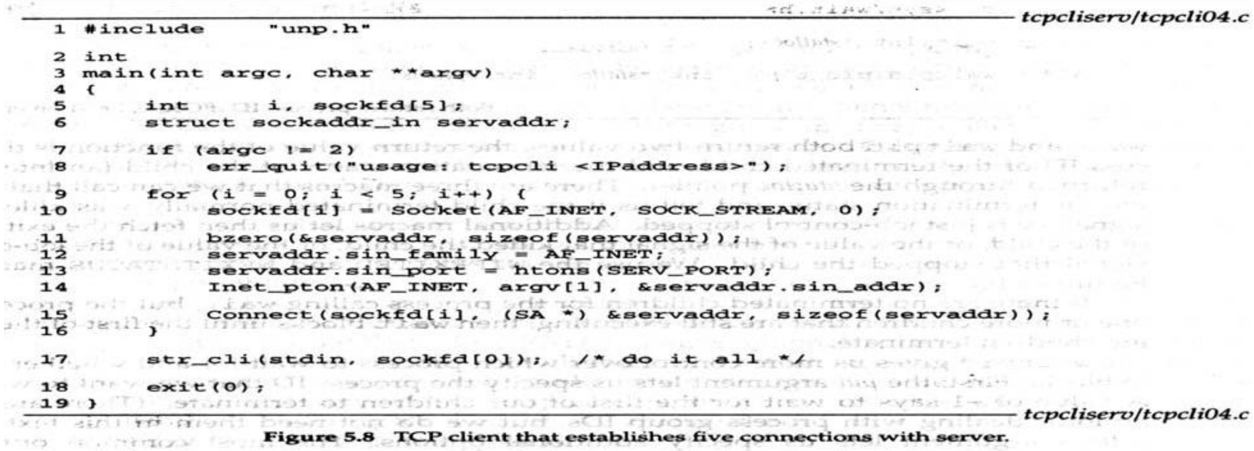
w***aitpid*** gives us more control over which process to wait for and whether or not to block. pid argument specify the process id that we want to wait for. a value of -1 says to wait for the first of our children to terminate. The option argument lets us specify additional options. The most common option is WNOHANG This tells the kernel not to block if there are no terminated children; it blocks only if there are children still executing.

The **wait** *pid* argument specifies a set of child processes for which status is requested. The waitpid() function shall only return the status of a child process from this set.

- If pid is equal to (pid_t) -1, status is requested for any child process. IN this respect, waitpid() is similar to wait().
- If pid is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If pid is 0, status is requested for any child process whose process groups.

Difference between wait and waitpid:

To understand the difference, TCP / IP client programme is modified as follows:



tcpcliserv/tcpcli04.c

```
#include    "unp.h"

int
main(int argc, char **argv)
{
    int     i, sockfd[5];
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit("usage: tcpcli <IPaddress>");

    for (i = 0; i < 5; i++) {
        sockfd[i] = Socket(AF_INET, SOCK_STREAM, 0);

        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(SERV_PORT);
        Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

        Connect(sockfd[i], (SA *) &servaddr, sizeof(servaddr));
    }

    str_cli(stdin, sockfd[0]);  /* do it all */

    exit(0);
}
```

tcpcliserv/tcpcli04.c

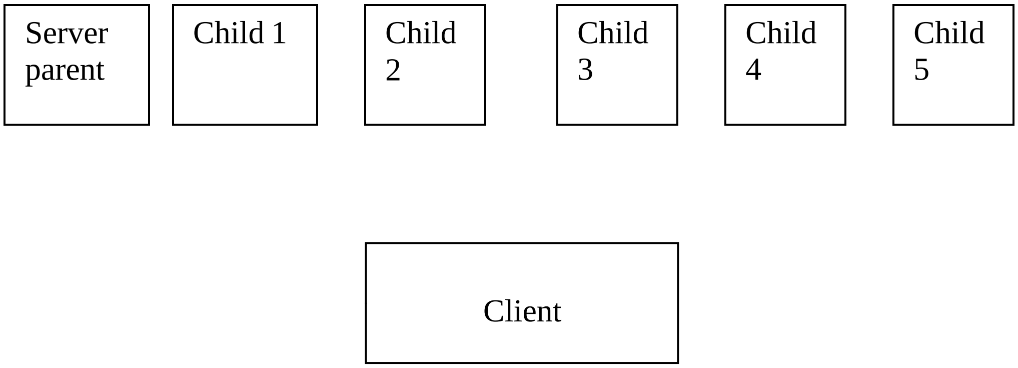
Figure 5.8 TCP client that establishes five connections with server.

The client establishes five connections with the server and then uses only the first one **(sockfd[0])** in the call to **str_cli**. The purpose of establishing multiple connections is to spawn multiple children from the concurrent server .

When the client terminates, all open descriptors are closed automatically by the kernel and all the five serve children terminate at about the same time. This causes five SIGCHLD signals to be delivered to the parent at about the same time, which we show in Figure below:

It is this delivery of multiple occurrences of the same signal that causes the problem that we are talking about. By executing ps, one can see that other children still exist as zombies.

Establishing a signal handler and calling *wait* form the signal handler are insufficient for preventing zombies. The problem is that all five signals are generated before the signal handler is executed, and the signal handler is executed only one time because Unix signals are not normally queued. Also this problem is non deterministic.



(If the client and server are on the same host, one child is terminated leaving four zombies. If we run the client and server on different hosts, the handler is executed twice once as a result of the first signal being generated and since the other four signals occur while the signal handling is executing, the handler is called once more This leaves three zombies. However depending on the timing of the FIN arriving at the server host, the signal handler is executed three or even four times).

The correct solution is to call waitpid instead of wait. Following code shows the server version of our *sig_ chld* function that handles SIGCHLD. Correctly. This version works because we call waitpid within the loop, fetching the status of any of our children that have terminated. We must specify the WNOHNG option; This tells waitpid not to block if there exists running children that not yet terminated. . In the code for wait , we cannot call wait in a loop, because there is no to prevent wait from blocking if there exits running children that have not yet terminated. Following version correctly handles a return EINTR from accept and it establishes a signal handler that called ***waitpid*** for all terminated children.

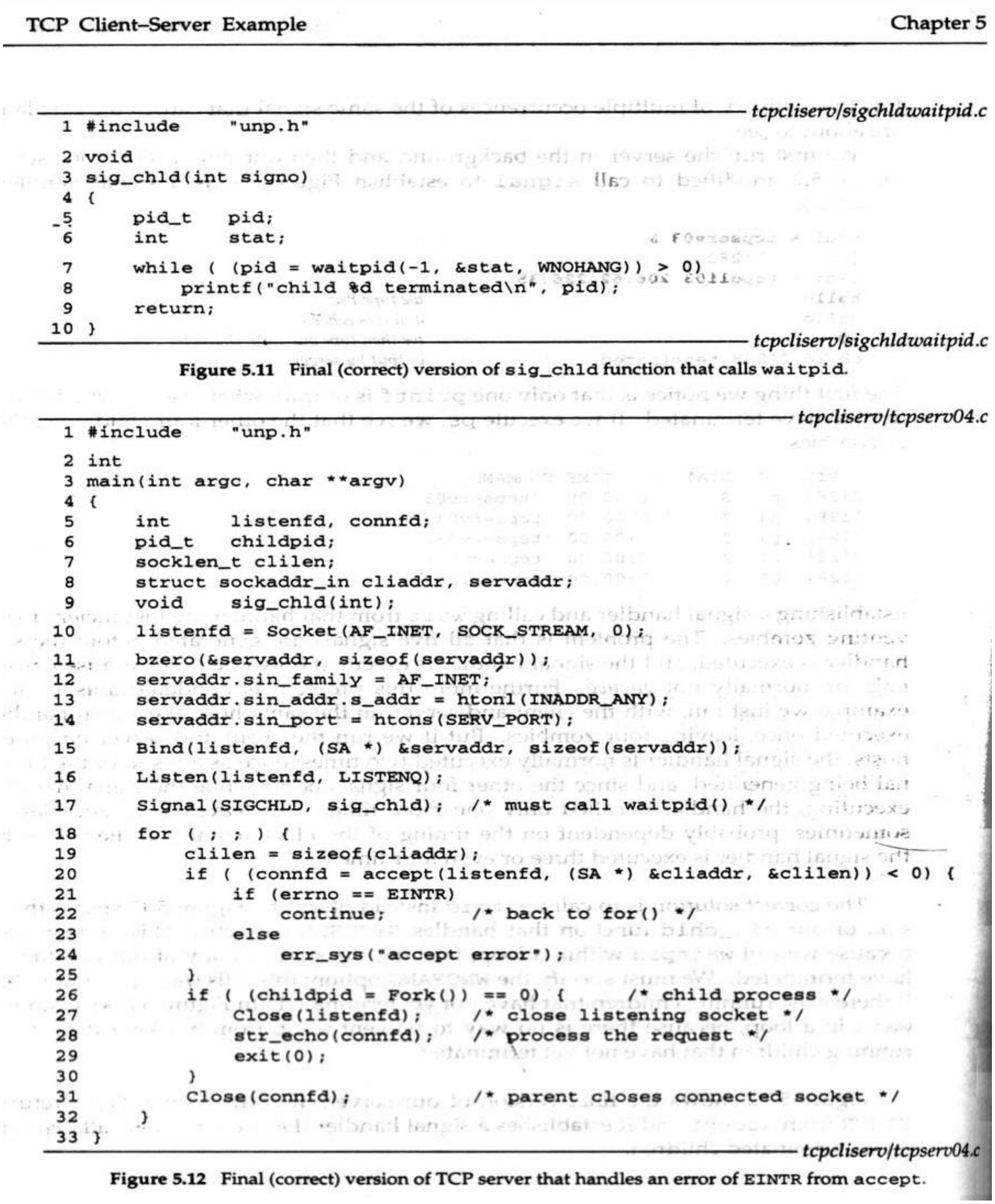
The following programme shows the implementation of waitpid(). The second part is the implementation of complete server programme incorporating the signal handler.

The three scenarios that we encounter with networking programme are :

We must catch SIGCHLD signal when forking child processes.

We must handle interrupted system calls when we catch signal.

A SIGCHLD handler must be coded correctly using waitpid to prevent any zombies form being left around



tcpcliserv/sigchldwaitpid.c

```
#include    "unp.h"

void
sig_chld(int signo)
{
    pid_t   pid;
    int     stat;

    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated\n", pid);
    return;
}
```

tcpcliserv/sigchldwaitpid.c

Figure 5.11 Final (correct) version of `sig_chld` function that calls `waitpid`.

tcpcliserv/tcpserv04.c

```
#include    "unp.h"

int
main(int argc, char **argv)
{
    int     listenfd, connfd;
    pid_t   childpid;
    socklen_t clilen;
    struct sockaddr_in cliaddr, servaddr;
    void    sig_chld(int);

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    Signal(SIGCHLD, sig_chld);  /* must call waitpid() */

    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
            if (errno == EINTR)
                continue;       /* back to for() */
            else
                err_sys("accept error");
        }
        if ( (childpid = Fork()) == 0) {    /* child process */
            Close(listenfd);    /* close listening socket */
            str_echo(connfd);   /* process the request */
            exit(0);
        }
        Close(connfd);          /* parent closes connected socket */
    }
}
```

tcpcliserv/tcpserv04.c

Figure 5.12 Final (correct) version of TCP server that handles an error of `EINTR` from `accept`.

.

Connection abort before *accept* returns :

Similar to interrupted system call, a non fatal error occurs when the client is reset after three handshake without transfer of any packet.

After the three way handshake, the connection is established and then the client TCP sends an RST (reset). ON the server side the connection is queued by its TCP waiting for the server process to call accept when the RST arrives. Some time later the server process calls accept.

What happens to aborted connection depends on the type of implementation . The Berkley derived implementation handles the aborted connection completely within the kernal and the server process never sees it. Most of the SVR4 (System V release 4) implementation returns an error to the process as the return from ***accept*** and the type of error depends on the implementation. Most implementation returns an ***errono*** EPROTO (protocol error) but posix .1g specifies that the return must be ECONNABORTED ("software caused connection abort"). The reason for this is that EPROTO is also returned when some fatal protocol related event occurs on the bytestream. Receiving the same error EPROTO by the server makes it difficult to decide whether to call ***accept*** again or not. IN case of ECONNABORTED error, the4 server ignores the error and just call accept again.

Termination of Server Process:

After starting client – server, the child process is killed at the server (kill the child process based on the its ID). This simulates the crashing of the server process, then what happens to client: The following steps take place.

1. We start the server and client on different hosts and type one line to the client to verify that all is OK. That line is echoed normally by the server child.
2. Identify the process ID of the server child and kill it. As part of the process termination, all open descriptors in the child are closed. This causes the FIN to by sent to the client and the client TCP responds with an ACK. This is the first half of the TCP connection termination.
3. The SIGCHLD signal is sent to the server parent and handled correctly.
4. Nothing happens at the client. The client receives the FIN from the sever TCP and responds with an ACK. But the problem is that the client process is blocked in the call to the ***fgets*** waiting for a line form the terminal.
5. When we type another line, str_cli calls written and the client TCP sends the data to the sever. This is allowed by TCP because the receipt of the FIN by the client TCP only indicates that the server process has closed its end of the connection and will not send any more data. The receipt of FIN does not tell the client that the server process has terminated (which in this case it has).
6. When the server TCP receives the data from the client, it responds with an RST since the process that had that socket open has terminated. We can verify that the RST is sent by watching the packets with ***tcpdump***
7. But the client process will not see the RST because it calls ***readline*** immediately a after the call to ***writen*** and ***readline*** returns 0 (end of line) immediately because of the FIN that was received inIT2351-NPM/ U-2/ 12 step 2. Our client is not expecting to receive an end of line at this point so it quits with an error message server terminated prematurely."
8. So when the client terminates (by calling err_quit), all its open descriptors are closed.

The problem in this example is that the client blocked in the call to fgets when the FIN arrives on the socket. The client is really working with the two descriptors - the socket and the user input - and instead of blocking on input from any one of the two sources, it should block on input from either source. This is the function of ***select*** and ***poll*** function .

SIGPIPE signal.:

When the client has more than one to write, what happens to it? That is when the FIN is received, the readline returns RST, but the second one is also written. The rule applied here is, when a process writes to a socket that has received an RST, the SIGPIPE signal is sent to the process. The default action of this signal is to terminate the process so the process must catch the signal to avoid involuntarily terminated.

If the process catches the signal and returns from the signal handler, or ignores the signal, the write operation returns EPIPE (error pipe)

```
        #include "unp.h"
void    str_cli (FILE *fp, int sockfd)
        {
                char sendline[MAXLINE], recvline[MAXLINE];
        while (fgets(sendline, MAXLINE, fp)!=null)
        {
                writen(sockfd, sendline, 1);
                sleep(1);
                        writen (sockfd, sendline +1, strlen (sendline)-1);
                if (readline()sockfd, recvline, MAXLINE==0)
                        err_quit ("str_cli: server terminated prematurely");
                fputs (recvline, stdout);
        }
}
```

in the above str_cli(), the writen is called two times: the first time the first byte of data is writen to the socket, followed by a pause of 1 sec, followed by the remainder of the line. The intention is for the first writen to elicit the RST and then for the second writen to generate SIGPIPE.

We start with the client, type in one line, see that line is echoed correctly, and then terminates the server child on the server host, we then type another line, but nothing is echoed and we just get a shell Prompt. Since the default action of the SIGPIPE is to terminate the process without generating a core file, nothing is printed by the Kornshell.

The recommended way to handle SIGPIPE depends on what the application what to do when this occurs. IF there is nothing special to do, then setting the signal disposition to SIG_IGN is easy, assuming that subsequent output operations will catch the error of EPIPE and terminate.

IF special actions are needed when the signal occurs, then the signal should be caught and any desired actions can be performed in the signal handler.

If multiple sockets are in use, the delivery of the signal does not tell us which socket encountered the error. IF we need to know which write caused the error, then we must either ignore the signal or return from the signal handler and handle **EPIPE** from ***write***

Crashing of Server Host:

Next scenario is to see what happens when the server host crashes. To simulate this we must run the client and server on different hosts. We then start server, start the client, type in a line to the client to verifyIT2351-NPM/ U-2/ 13 that the connection is up, disconnect the server host from the network, and type in another line at the client. This also covers the scenario of the server host being unreachable when the client sends data (some immediate router is down after the connection has been established).

The following steps take place.

1. When the server host crashes, nothing is sent out on the existing network connections. That is we are assuming the host crashes, and is not shut down by the operator
2. We type a line of input to the client, it is written by writen and is sent by the client TCP as a data segment. The client then blocks in the call to readline waiting for the echoed reply.
3. If we watch the network with tcpdump, we will see the client TCP continually retransmit the data segment, trying to receive ACK from the server. Berkley derived implementations transmit the date segments 12 times, waiting around 9 minutes before giving up. When the client finally gives up, an error is returned to the client process. Since the client is blocked in the call to readline, it returns an error. Assuming the server host had crashed and there were no responses

at all to the client’s data segments, the error is ETIMEDOUT. But if some intermediate router determine that the server was unreachable and responded with ICMP destination unreachable message, then error is either EHOSTUNREACH or ENETUNREACH.

To detect that the server is unreachable even before 9 minutes, place a time out call to readline. To find the crash of server even if client is not sending data actively, another technique is used which used SO_KEEPALIVE socket option

Shutdown of Server

When a Unix system is shutdown, the init process normally sends the SIGTERM signal to all processes (this signal can be caught), waits some fixed amount of time (often between 5 and 20 seconds), and then sends SIGKILL signal (which we cannot catch) to any process still running. This gives all running processes a short amount of time to clean up and terminate.

If we do not catch SIGTERM and terminate, our server will be terminated by SIGKILL signal.

When the process terminates, all the open descriptors are closed, and we then follow the same sequence of stepsdiscussed under “termination of server process”.

We need to select the select or poll function in the client to have the client detect the termination of the server process as soon it occurs.

I / O MULTIPLEXING : THE *select* AND *poll* FUNCTIONS

It is seen that the TCP client is handling two inputs at the same time: **standard input and a TCP socket.** It was found that when the client was blocked in a call to read(by calling readline function), and the server process was killed. The server TCP correctly, correctly sends a FIN to the client TCP, but since the client process is blocked reading from the standard input, it never sees the end – of file until it reads from the socket . What we need is the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e. input is ready to be read, or the descriptors is capable of taking more outputs). This capability is called I /O Multiplexing and is provided by the ***select*** and ***poll*** functions . There is one more Posix .1g variations called ***pselect.***

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

- When a client is handling multiple descriptors (normally interactive input and a network socket), I/O multiplexing should be used. This is the scenario that was described in the previous paragraph.
- It is possible, but rare, ofr a client to handle multiple sockets at the same time. We show an example of this using ***select*** _in the context of web client
- If a TCP server handles both a listening socket and its connected sockets, I / O multiplexing is normally used.
- IF a server handles both TCP and UDP, I/O multiplexing is normally used.
- If a server handles multiple services and perhaps multiple protocols, I/O multiplexing us normally used.

It is not restricted only to networking programme, it may be used in any nontrivial application as well.

I/O Models:

There are five I /O models in the Unix. These are:

a. Blocking I /O
b. Non blocking I / O
c. I/O Multiplexing (select and poll)
d. Signal driven I/O (SIGIO)
e. Asynchronous I/O (the Posix 1 aio_functions)

There are two distinct phases for an input operation.:

a. waiting for the data to be read and
b. copying the data from the kernel to the process.

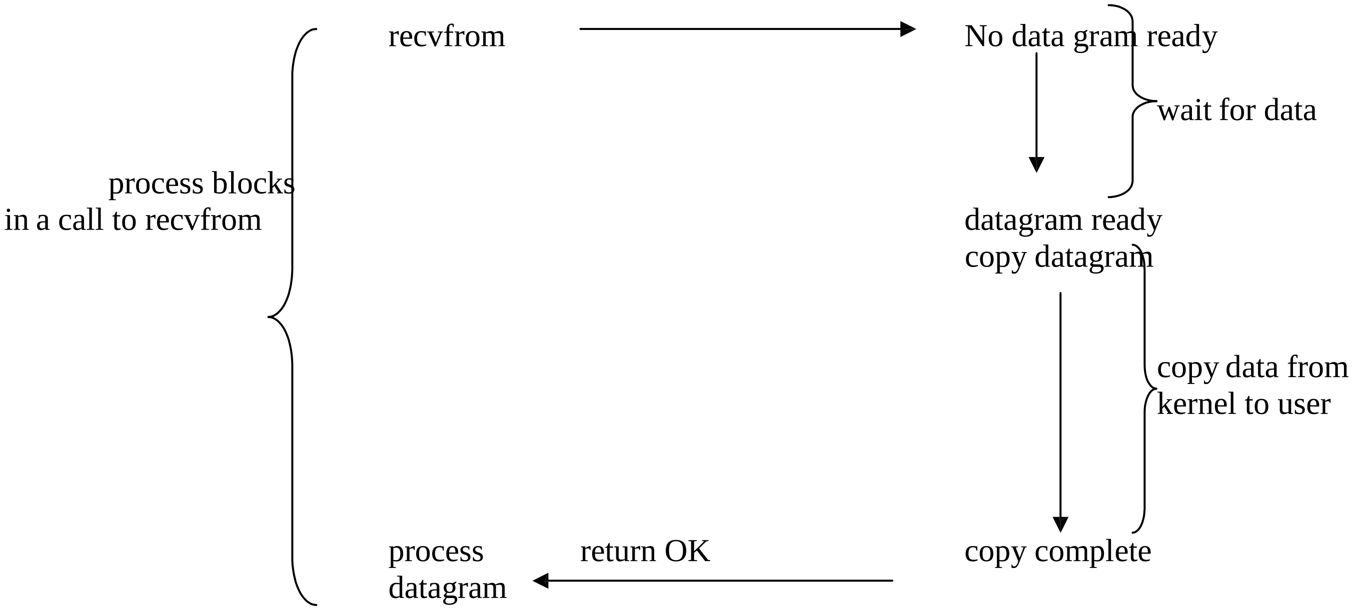
Fo ran input operation on a socket the first step normally involves waiting for the data to arrive on the network. When the packet arrives, it is copied into buffer within the kernel. The second step is copying this data from the kernel's buffer into our applications buffer.

Blocking I/O Model :

The most prevalent model for I/O is the blocking I/O model, which we have used for all our examples so far in the text.. BY default, all sockets are blocking. Using a datagram socket for our examples we have the scenario as shown below. In UDP the concept of data being ready to be read is simple because either an entire datagram packet is received or not.

IN this example ***recvfrom*** as a system call as it differentiated between our application and the kernel.

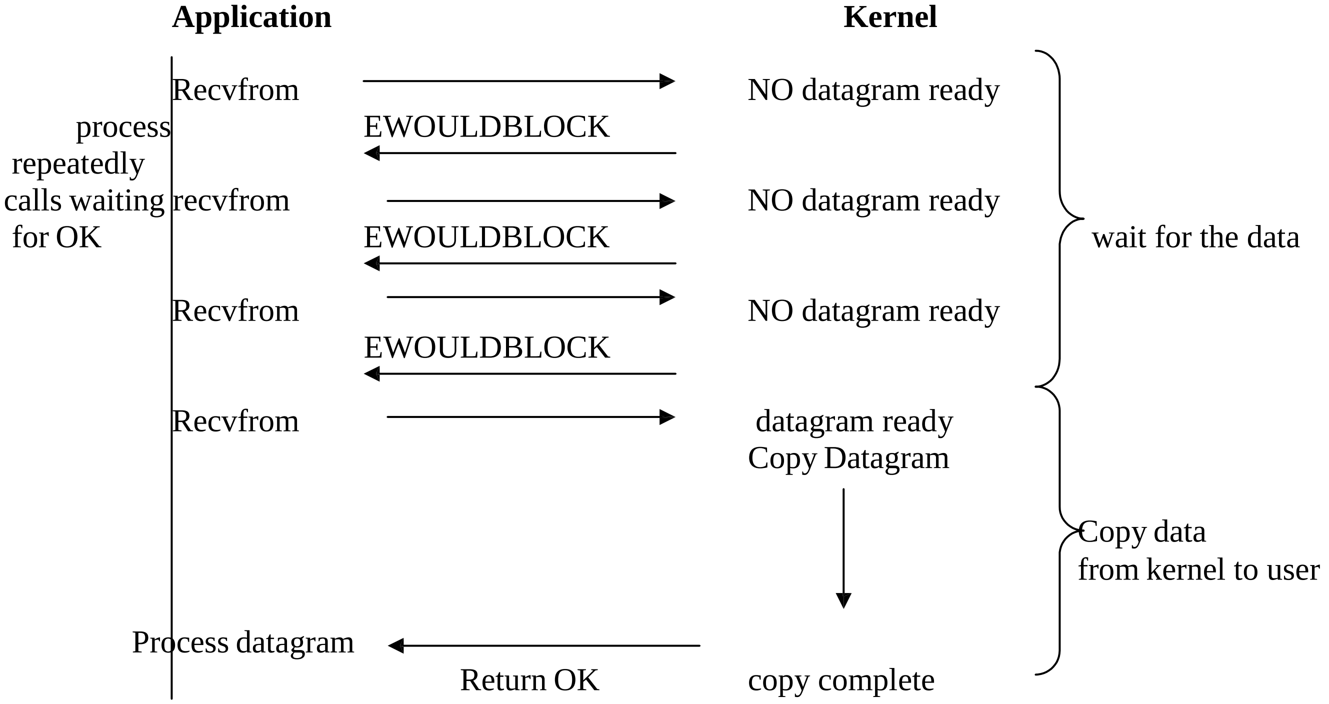
The process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. The most common error is the system call being interrupted by a signal. We say that our process is blocked the entire time from when it call recvfrom until it returns. When recvfrom returns OK, our application processes the datagram.



Blocking I/O Model.

Non Blocking I/O Model :

When the socket is set to non blocking, the kernel is told that "when I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep but return an error message instead." The following figure gives the details.

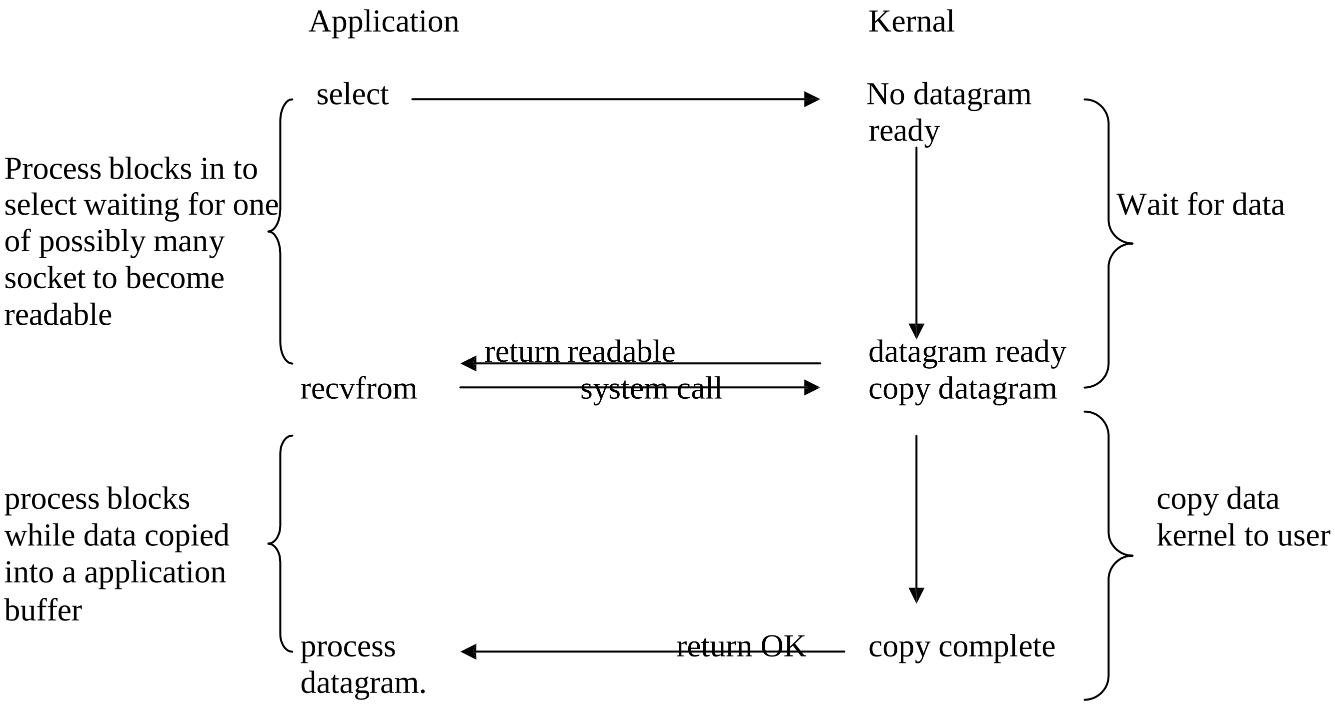


Non Blocking Model

During the first three times, when the ***recvfrom*** is called, there is no data to return, so the kernel immediately returns an error **EWOULDBLOCK.** Fourth time, when ***recvfrom*** is called, the datagram is ready, it is copied into our application buffer and the ***recvfrom*** returns OK. The application then process the data.

When the application puts the call ***recvfrom*** in a loop, on a non blocking descriptors like this, it is called polling. The continuation polling of the kernel is waste of CPU time. But this model is normally encountered on system that are dedicated to one function.

I / O Multiplexing Model :
IN this ***select*** or ***poll*** functions are called instead of ***recvfrom.*** The summary of I/O Multiplexing call is given in the following figure:



I / O Multiplexing Model

IN this the call to select is blocked waiting for the datagram socket to be readable. When select returns that the socket is readable, then the recvfrom called to copy the datagram into the application.

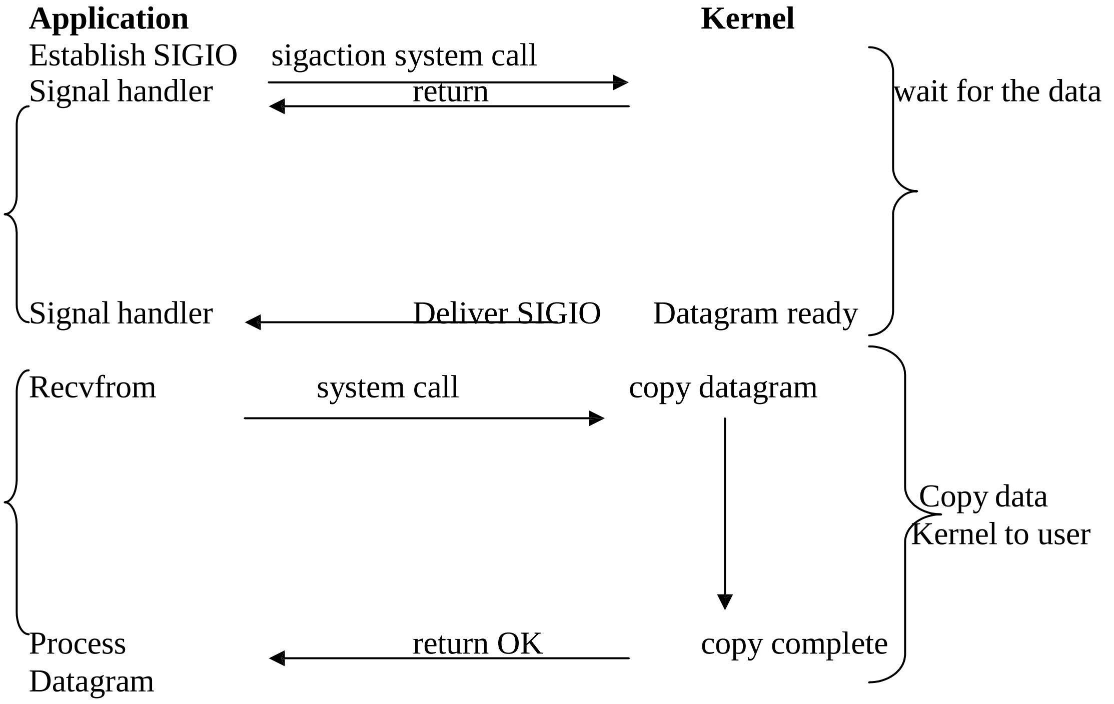
Compared to blocking model, the difference here is that the application can wait for more than one descriptors to be ready. The disadvantages is that select requires two system calls.

Signal Driven I/O Model

Signals are used to tell the kernel to notify applications with the **SIGIO** signal when the descriptor is ready. It is called **signal driven I/O** Model. The summary is shown in the following figure.

First enable the socket for the signal driven I/O and install a signal handler using the ***sigaction*** system call. The return from this system call is immediate and our process continuous, it is not block. When the datagram is ready to be ready, the SIGIO signal is generated for our process. We can either read the datagram from the signal handler by calling recvfrom and then notify the main loop that the data is ready to be process, or we can notify the main lop and let it read the datagram.

The advantage I this model is that we are not blocked while waiting for the datagram top arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the dta is read to process or that the datagram is ready to be read.

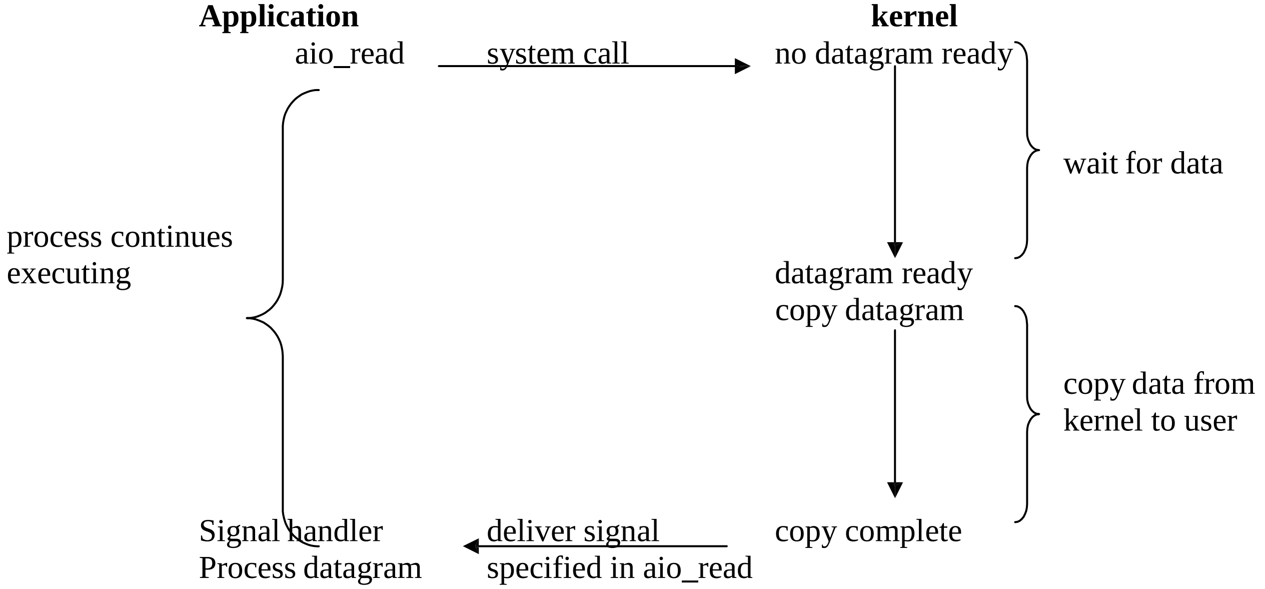


Signal Driven I/O Model

Asynchronous I/O Model

Asynchronous I/O are new with the 1993 edition of Posix 1g. In this the kernel is told to start operation and to notify when the entire operation (including the copy of the data from the kernel to our buffer) is over.

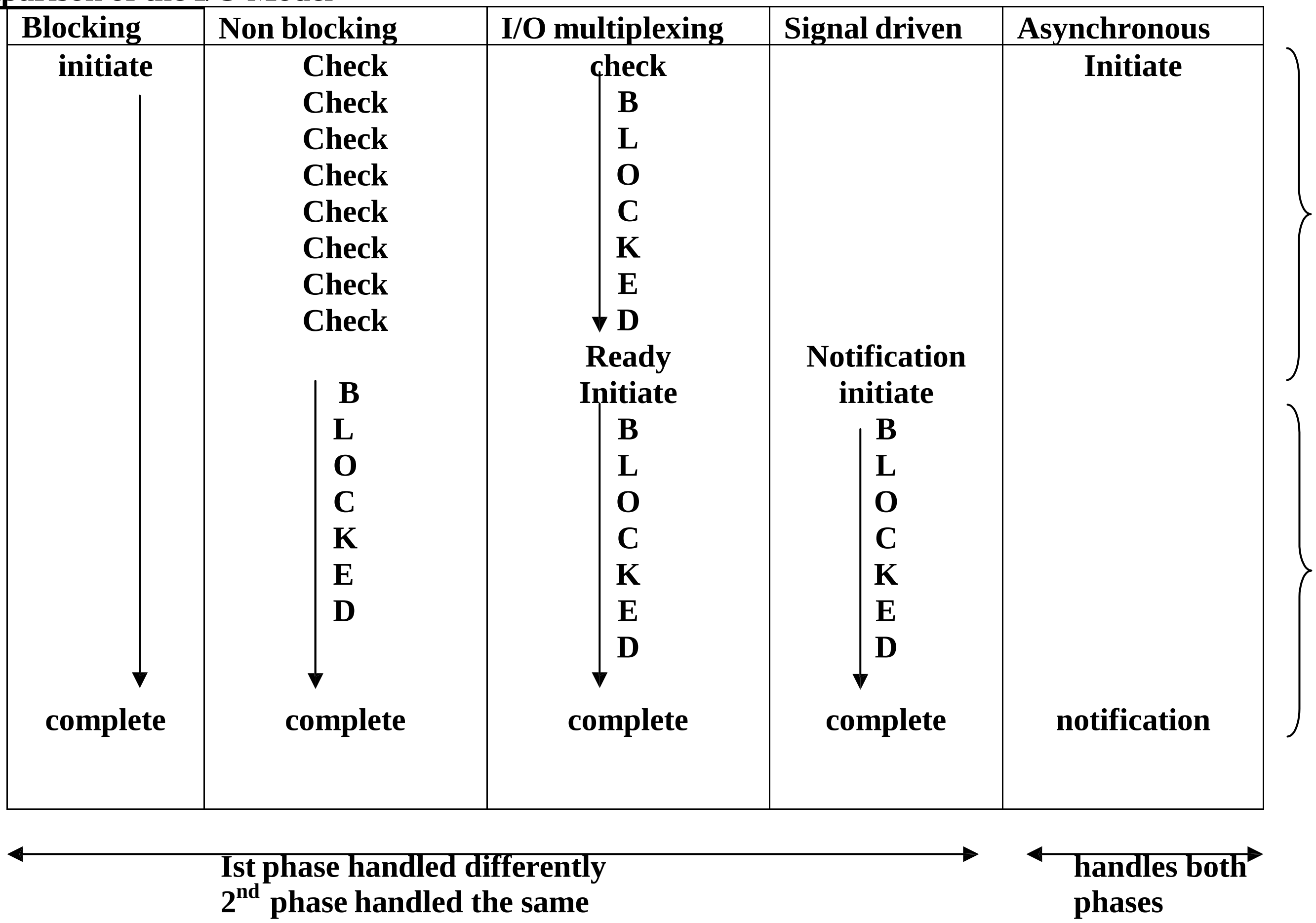
The main difference between this and the signal driven I/O model in the previous section is that with the signal driven I/O the kernel tells when the I/O operation can be initiated. But with asynchronous I/O, the kernel tells us when an I/O operation is complete. It is summarized as given below:



Asynchronous I/O Model

The function ***aio_read*** is called and the descriptor, buffer pointer, buffer size, file offset and how to notify when the entire operation is complete are passed to the kernel. The system calls returns immediately and our process is not blocked waiting for the I./O to complete. It is expected that the kernel will generate some signal when the operation is complete. The signal is generated only when the data is copied completely in the application buffer.

Comparison of the I/O Model



Blocking	Non blocking	I/O multiplexing	Signal driven	Asynchronous
initiate complete	Check Check Check Check Check Check Check Check BLOCKED complete	check BLOCKED Ready Initiate BLOCKED complete	Notification initiate BLOCKED complete	Initiate notification

Ist phase handled differently
2nd phase handled the same

handles both phases

The main difference between the four models is the first phase as the second phase in the first four models is the same. The process is blocked in a cal to ***recvfrom*** while the data is copied from the kernel to the caller's buffer. Asynchronous I/O however, handles both the phases and is different form the first four.

Synchronous vs asynchronous:

A synchronous I/O operation causes the requesting process to be blocked until that I?O operation is completes.

An asynchronous I/O operation does not cause the requesting process to be blocked.

Based on this definition, the first four are synchronous as the actual I/O operation blocks the process. Only asynchronous I/O model matches the asynchronous I/O definition.

Select Function :

The select function allows the process to instruct the kernel to wait for any one of the multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.

As an example, we can call select and tell the kernel to return only

- any of the descriptors in the set { 1,4,5} are ready for reading, or
- any of the descriptors in the set {2,7} are ready for writing, or

- any of the descriptors in the set {1,4 } have an exception condition pending or
- after 10.2. seconds have elapsed.

That the kernel is told in what descriptors we are interested in (for reading, writing or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets: any descriptor can be tested using select.

```
    #include <sys/select.h>
    #include <sys/time.h>
int select (int maxfdp1,  fd_set *readset, fd_set *writeset, fd_set *excepset, const struct timeval *timeout);
                    returns : positive count of ready descriptors, o on timeout, -1 on error.
```

Consider the final arguments: This 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;  /* micros seconds */
                }
```

There are three possibilities:

- **wait forever**: return only when one of the specified descriptors is ready for I/O . For this, we specify the timeout argument as a **null pointer**.
- **Wait upto 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 pointer to by the **timeout** argument.
- **Do not wait at all**: return immediately after checking the descriptors. This is called **polling**. To specify this, the timeout argument must point to a timeval structure and the timer value must be **zero**

In case of first two scenario, the wait is normally interrupted if the process catches a signal and returns from the signal handler.

The const qualifier on the timeout argument means it is not modified by select on return. For example, if we specify a time limit of 10 sec, and select returns before the timer expires, with one or more descriptors ready or with an error of EINTR, the timeval structure is not updated with the number of seconds remaining when the functions returns.

The three middle arguments readset, writeset and excepset specify the descriptors that we want the kernel to **test** for reading, writing and exception conditions.

The two exceptions conditions currently supported are:

- The arrival of out of bound data for a socket.
- The control status information to be read from the master side of pseudo terminal

The ***maxfdp1*** argument specifies the number of descriptors to be tested. Its value is the maximum descriptors to be tested plus one. The descriptors 0,1,2, through and including ***maxfdp1*** –1 are tested. The constant FD_SETSIZE defined by including <sys /select.h>, is the number of descriptors in the fd_set data type. It s value is often 1024, but few programs use that many descriptors. The ***maxfdp1*** argument forces us to calculate the largest descriptors that we are interested in and then tell the kernel this value. For 1,4,5 the ***maxfdp1*** is 6.

The design problem is how to specify one or more descriptors values for each of these three arguments. **Select** uses **descriptor sets**, typically an array of integers – a32 by 32 array- of 1024 descriptors set. The descriptors are programmed for initializing /reading / writing and for closing using the following macros:

```
        void FD_ZERO (fd_set   * fset)                /* clear all bits in fset */
        void FD_SET (int fd,  fd_set   * fset)        /* Turn on the bit for fd in  fset*/
        void FD_CLR (int fd,  fd_set   * fset)        /*turn off the bit for fd in fset
        void FD_ISSET (int fd,  fd_set   * fset)      /* is the bit for fd on in fset? */
```

For example to define a variable of type fd_set and then turn on the bits for descriptors, we write

```
        fd_set  rset;
        FD_ZERO (&rset);   /* initialize  the set; all bits to zet */
        FD_SET (1, &rset);  /* turn on bit for fd 1
        FD_SET (4, &rset );  /* turn on  bit for fd 4; */
```

If the descriptors are initialized to zero, unexpected results are likely to come.

Middle arguments to select, readset, writeset, or exceptset can be specified as null pointer, if we are not interest in that condition. The *maxfdp1* arguments specifies the number of descriptors to be tested. Its value is the maximum descriptors to be tested, plus one (hence the name maxfdp1). The descriptors 0,1,2 up through and including *maxfdp1 – 1* are tested.

Select function modifies the descriptor sets pointed by the readset, writset and excepset pointers. These three arguments are value-result arguments. When we call the function, we specify the values of the descriptors that we are interested in and on return the result indicates which descriptors are ready. We use FD_ISSET macros return to test a specific descriptor in an fd_set structure. Any descriptors that is not ready on return will have its corresponding bit cleared in the descriptors set.

str_cli Function :

The ***str_cli*** Function is rewritten using the select function as shown below:

```
# include “unp.h”
 void str_cli (FILE *fp, int socket)
        {
        int             maxfdp1;
        fd_set          rset;
        char    sendline[MAXLINE],          recvline[MAXLINE];
        FD_ZERO (&rset);
        for (  ;  ;)
                {
                FD_SET (fileno (fp), &rset);
                FD_SET (sockfd, &rset);
                maxfdp1 = max (fileno (fp), sockfd) + 1;
                select (maxfdp1, &rset, NULL, NULL, NULL);
                if (FD_ISSET(sockfd, &rset))
                        {       if (readline (sockfd, recvline, MAXLINE)==0)
                                err_quit (“str_cli: server terminated prematurely”);
                        fputs (recvline, stdout);
                        }
                if (FD_ISSET(fileno(fp), &rset))
                        {if (fgets (sendline,  MAXLINE, fp)==NULL)
                        return;
                        writen(sockfd, sendline, strlen (sendline));
                        }
                }
        }
```

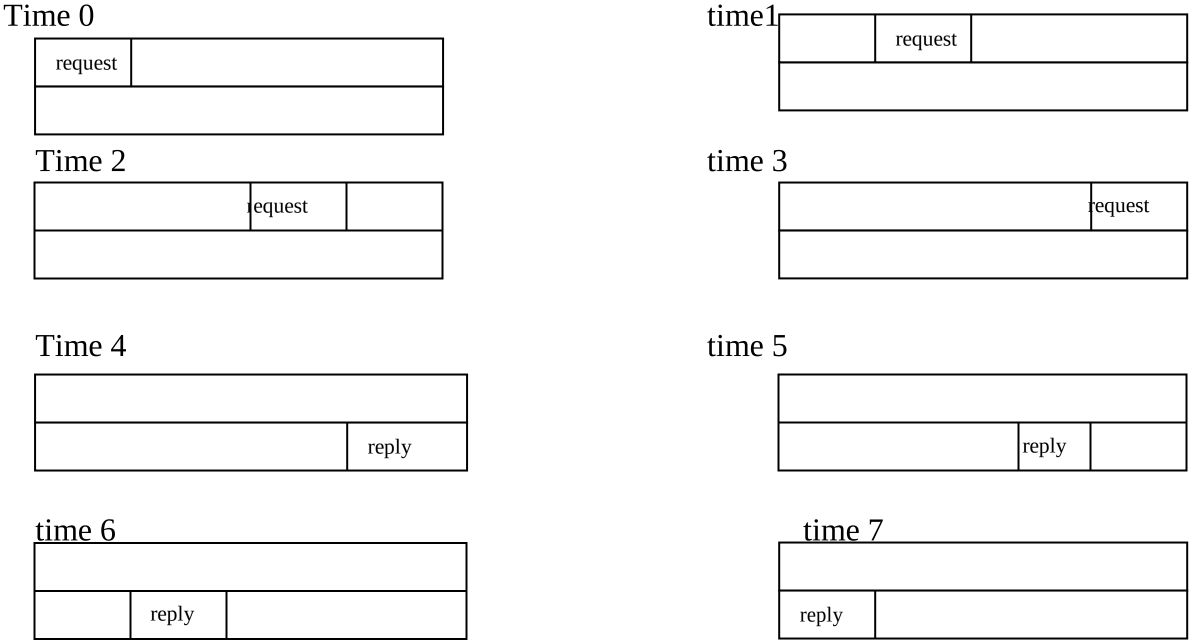
IN the above code the descriptors set is initialized to zero using FD_ZERO. Then, the descriptors, file pointer ***fp*** and socket ***sockfd*** are turned on using FD_SET. maxfdp1 is calculated from the descriptors list. **Select** function is called. IN this writeset pointer and exception set pointers are both NULL. Final time pointer is also NULL as the call is to be blocked until something is ready.

If on return from the select, socket is readable, the echoed line is read with ***readline*** and output by ***fputs.***

If the standard input is readable, a line is read by ***fget*** sand written to the sockets using ***writen.***
IN this although the four functions fgets, writen, readline and fputs are used, the order of flow within the function has changed. In this, instead of flow being driven by the call to fgets, it is driven by the call to select.

This has enhanced the robustness of the client.

Batch Input:
The echo client server, works in a stop and wait mode. That is , it sends a line to the server and then waits for the reply. This amount of time is one RTT (Round Trip Time) plus the server's processing time. If we consider the network between the client and the server as a full duplex pipe with requests from the client to server, and replies in the reverse direction, then the following shows the stop and wait mode.



The request is sent by the client at time 0 and we assume RTT of 8 units of time. The reply sent at a time 4 is received at time 7. It is assumed that there is no serving processing time and that the size of the request is the same as the reply. Also, TCP acknowledgment are ignored.

But as there is a delay between sending a packet and that packet arriving at the other end of the pipe, and since the pipe is full duplex, in this example we are only using one- eighth of the pipe capacity. This stop and wait mode is fine for interactive input, but since our client reads from standard input and writes to standard output, we can easily run our client in a batch mode. When the input is redirected as in client server example, the output file is always same.

To see what is happening in the batch mode, we can keep sending requests as fast as the network can accept them. The server processes them and sends back the replies at the same rate. This leads to the full pipe at time 7 as shown below:

Time 7

Request8	Request7	Request6	Request5
Reply 1	Reply 2	Reply 3	Reply 4

Time 8

Request9	Request8	Request7	Request6
Reply 2	Reply 3	Reply 4	Reply 5

Filling the pipe between the client and the sever : batch Mode;

Now to understand the problem with the **str_cli,** let the input file contains only nine lines. The last line is sent at time 8 as shown above. But we cannot close the connection after writing this request, because there are still other requests and rep0les in the pipe. The cause of the problem is our handling of end of file on input. The function returns to the main function, which then terminates. But in a batch mode, an end of file on the input does not imply. That we have finished reading from the socket. There might still requests on the way to the server or replies on the way to back from the server.

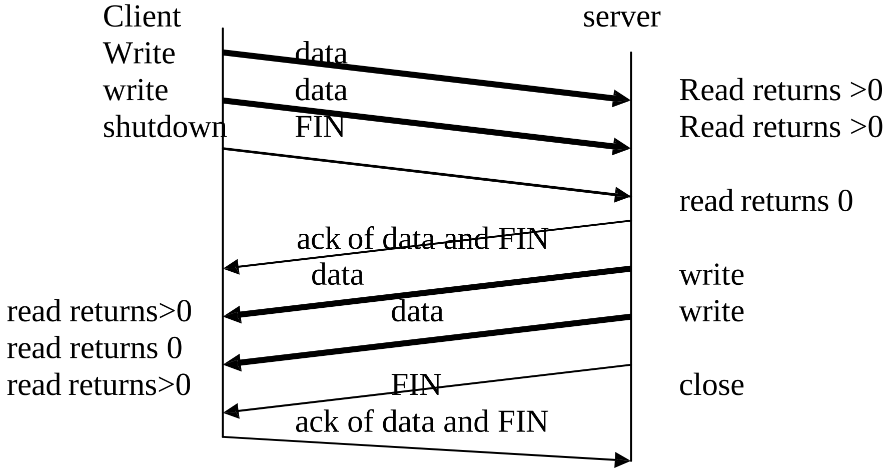
What we need is a way to close one half of the TCP connection. That is we want to send FIN to the server, telling it we have finished sending data, but leave the socket descriptors for reading. This is done with the ***shutdown*** function.

shutdown function.

The normal way to terminate a network function is to call the close function. But there are two limitations with close that can be avoided with the ***shutdown*** function.

1. Close decrements the descriptors' reference count and closes the socket only if the count reaches 0. With ***shutdown*** function, we can initiate TCP's normal connection termination sequence regardless of the reference count.

2. Close terminates both directions of data transfer, reading and writing. Since a TCP connections is full duplex, there are times when we want to tell the other end that we have finished sending, even though that end might have more data to send us. This is the scenario we encountered in the previous section with batch input to our str_cli function. Following figure shows the typical scenario



Syntax of shutdown () function

```
        int shutdown (int  sockfd, int howto);
                        Returns : 0 if OK, -1 on error.
```

The action of the function depends on the value of the ***howto*** argument.

SHUT_RD – The read half of the connection is closed. NO more data can be received on the socket and any data currently in the socket receive buffer is discarded. He process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP sockets is acknowledged and then silently dropped.

SHUT_WR : The write half of the connection is closed. IN the case of TCP, this is called half close. Any data currently in the socket

SHUT_RW: Read half and write half of the connections are both closed.

The following code gives the str_cli function that is modified by using shutdown function:

```
# include “unp.h”
 void str_cli (FILE *fp, int socket)
        {
        int                     maxfdp1, stdineof;
        fd_set                  rset;
        char    sendline[MAXLINE],              recvline[MAXLINE];
        stdineof = 0;
        FD_ZERO (&rset);
        for (  ;  ;)
                {
                if (stdineof ==0)
                        FD_SET (fileno (fp), &rset);
                FD_SET (sockfd, &rset);
                maxfdp1 = max (fileno (fp), sockfd) + 1;
                select (maxfdp1, &rset, NULL, NULL, NULL);
                if (FD_ISSET(sockfd, &rset))
                        {       if (readline (sockfd, recvline, MAXLINE)==0)
                                { if ( stdineof ==0 )
                                        return;
                                else
                                        err_quit (“str_cli: server terminated prematurely”);
                                }
                        fputs (recvline, stdout);
                        }
                if (FD_ISSET(fileno(fp), &rset))
                        { if (fgets (sendline,  MAXLINE, fp)==NULL)
                                {stdineof = 1;
                                shetdown (sockfd, SHUT_WR);
                                FD_CLR (fileno(fp), &rset);
                                continue;
                                }
                        writen(sockfd, sendline, strlen (sendline));
                        }
                }
        }
```

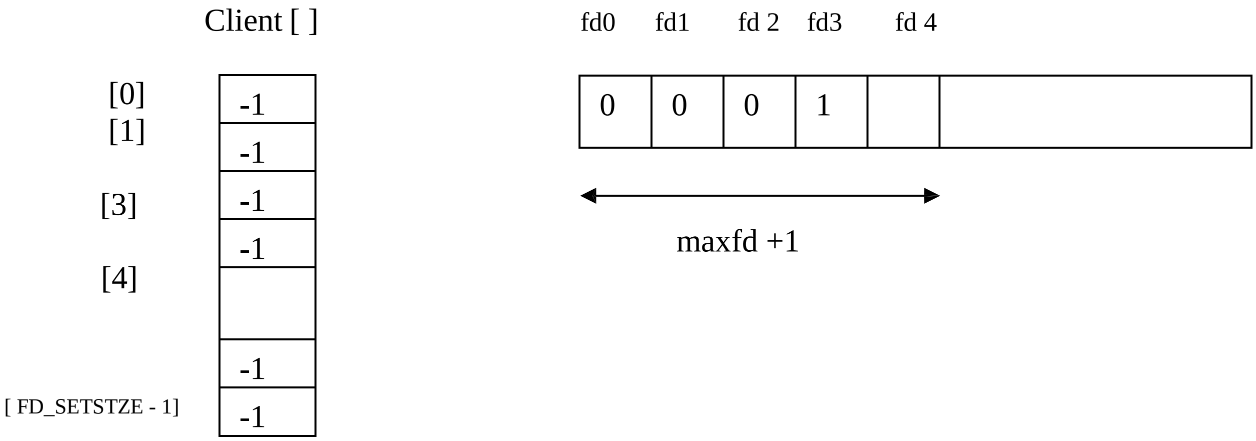
IN this ***stdineof*** in is a new flag that is initialized to 0. As long as this flag is 0, each time around the main loop we **select** on standard input for readability.

When we read the end of file on the socket, if we have already encountered an of file on standard input, this is the normal termination and the function returns. But if the end of file is not encountered on the standard input, the server process is prematurely terminated/

When we encounter the end of file on standard input, the new flag is set to 1 and we call shutdown with a second argument of SHUT_WR to send the FIN.

TCP Echo Server with IO multiplexing:

The TCP echo server can use the select function to handle any number clients, instead of forking one child per client. This needs to create two sets of data structures. These are : read descriptors set and connected socket descriptors set as shown below:



The server maintains only read descriptors set, which is shown above. When the server is started in the foreground, the descriptors 0, 1 and 2 are set to standard input, output and error. Therefore the available descriptors for the listening socket is 3.

Another client descriptors set is also shown. It is shows the number of the connected socket descriptors for each client. All elements of this array are initialized to –1.

The only non zero entry in the descriptors set is the entry of listening socket and the first argument to select will be 4.

When the first client establishes a connection with our server, the listening descriptors becomes readable and the server calls accept. The new connected descriptor returned by accept will be given, as per the assumption . From now on the server must remember the new connected socket in its client array and the connected socket must be added to the descriptor set. The updated data set is shown below.

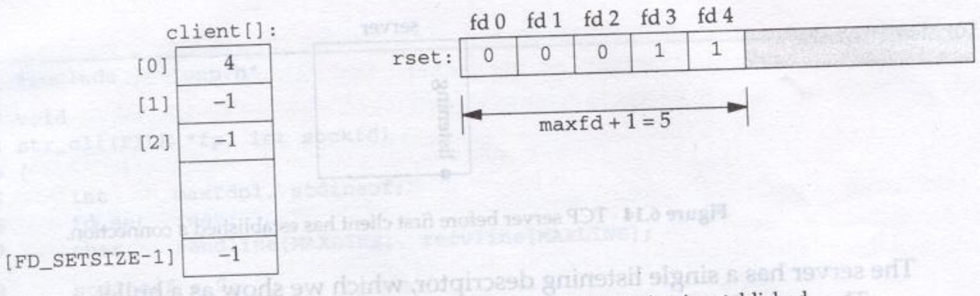


Figure 6.17 Data structures after first client connection is established.

Some time later a second client establishes a connection and we have the scenario as shown below:

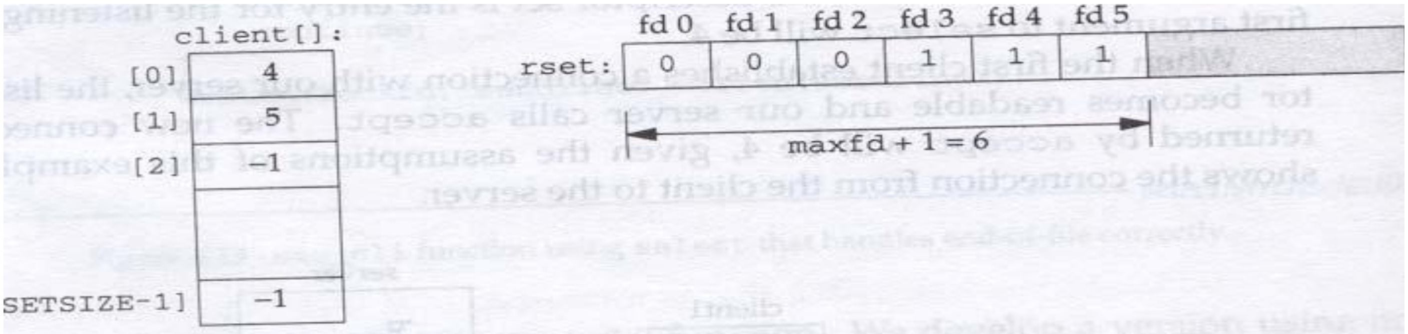


Figure 6.19 Data structures after second client connection is established.

The new connected socket which we assume as 5, must be remembered, giving the data structure as shown above.

When the first client terminates its connection, the client TCP sends FIN, makes descriptors in the server readable. When our server reads this connected socket, readline returns 0. We then close this colet and update our data structure accordingly. The value of the client[0] is set –1. and descriptors 4 in the descriptors is set to 0. The value of mxfd does not change.

That is as the client arrive we record their connected socket descriptor in the first available entry in the client array () the first entry with a value of –1. We must also add the connected socket to the descripto set. The variable maxi is the highest index in the current value of the first argument select. The only limit on the number of descriptors allowed for this process by the kernel. The first half server side programme is given below:

Short Questions

1. Explain TCP Echo server and client.
2. Define signal.
3. Explain signal function.
4. What is wait and Waitpid function?
5. What is the difference between wait and Waitpid function?
6. Explain crashing of server host.
7. Explain Shutdown of server host.
8. Explain the syntax of signal function.
9. Explain I/O multiplexing.
10. What are the scenarios used in I/O multiplexing applications?
11. What are the 5 basic I/O models available in UNIX?
12. State where POSIX function is used.
13. Define the two terms used in POSIX.
14. What are the possibilities of select function?
15. What are the three select descriptor arguments?
16. Difference between close function and shutdown function.
17. Difference between select function and pselect function.
18. Define poll function.
19. Difference between poll function and select function.
20. What are the three conditions handled with the socket?
21. What are the three classes of data identified by poll?

Big Questions

1. Write a TCP socket program to implement an Echo server/Echo client. (16)
2. Explain the following concept with suitable example. (16)

a) Shutdown function
b) Server host crashes
c) Input output models
d) Posix signal

3. Discuss the following scenario of server operations.

a) Crashing of server host (06)
b) Crashing and rebooting of server host (06)
c) Shutdown of server host (04)

4. Explain in detail about the various I/O models in Unix operating system. (16)
5. Explain in detail about

a) POSIX signal handling (08)
b) Boundary condition (08)

UNIT III

SOCKET OPTIONS, ELEMENTARY UDP SOC SOCKETS:

Socket options – Getsocket and setsocket functions – Generic socket options – IP socket options – ICMP socket options – TCP socket options – Elementary UDP sockets – UDP echo server – UDP echo client – Multiplexing TCP and UDP sockets – Domain Name System – Gethostbyname function – IPV6 support in DNS – Gethostbyadr function – Getservbyname and getservbyport functions.

UNIT-3

SOCKET OPTIONS

There are a number of options like – send buffer size, bypass table look up, get socket type, to name a few - that needs to be set. Also these needs to be got back also. Towards realizing this, there are ways to get and set socket options. These are:

- The ***getsockopt*** and ***setsockopt*** functions
- The ***fcntl*** functions and
- The ***ioctl*** functions

The options are different for IPv4, IPv6, TCP and for generic categories. These options are briefly discussed in this chapter.

Fcntl is the Posix way to set a socket for non blocking I/O, signal driven I/O and to set the owner of a socket. Ioctl is similar to fcntl but also has added functionalities that are not defined in the fcntl.

The syntax for the ***getsockopt*** and ***setsockopt*** is givenbelow:

#include <sys / socket.h>

int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t, *optlen);
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t, *optlen);
both return : 0 if OK –1 on error.

The sockfd refers to an open socket descriptor. The level specifies the code in the system to interpret the option. The general socket code, or some protocol specific (IPv4 or IPv6 or TCP etc) The optval is a pointer to a variable from which the new value of the option is fetched by setsockopt, or into which the current value of the option is stored by the getsockopt. The size of this variable is pecified by the final argument, as a value result for getsockopt.

The list of options that can be set and get by the socket options are listed in the fig 7.1 (page 179.) These options, in additions being classified based on the protocol, they are classified as either binary options (shown as flags) that enable or disable a certain features and other options that fetch and return specific values that we can either set or examine.

[Here are the functions for examining and modifying socket options. They are declared in `sys/socket.h`.

— *Function: int* ***getsockopt*** *(*int socket, int level, int optname, void *optval, socklen_t *optlen-ptr*)*

The `getsockopt` *function gets information about the value of option* optname *at level* level *for socket* socket.

The option value is stored in a buffer that optval *points to. Before the call, you should supply in* `*`optlen-ptr *the size of this buffer; on return, it contains the number of bytes of information actually stored in the buffer.*

Most options interpret the optval *buffer as a single* `int` *value.*

The actual return value of `getsockopt` *is* `0` *on success and* `-1` *on failure. The following* `errno` *error conditions are defined:*

`EBADF`

The socket *argument is not a valid file descriptor.*

`ENOTSOCK`

The descriptor socket *is not a socket.*

`ENOPROTOOPT`

The optname *doesn't make sense for the given* level.

Function: int ***setsockopt*** *(*int socket, int level, int optname, void *optval, socklen_t optlen*)*

This function is used to set the socket option optname at level level for socket socket. The value of the option is passed in the buffer optval of size optlen.]

The data type column shows the data types of what the ***optval*** pointer must point to for each option. The two braces notation {} is used to indicate a structure.

There are two basic types of options : binary options that enables or disables a certain features (flag), and options that fetch and return specific values that we can either set or examine values. The column labeled "flag" specifies if the option is a flag option. When calling ***getsockopt*** for these flag options, ***optval*** is an integer. The value returned in optval is zero if the option is disabled, or nonzero if the options is enabled. Similarly, ***setsocket*** requires a nonzero optval to turn the options on and a zero value to turn the option off. If the flag column does not contain a * then the options is to turn the options is used to pass a value of the specified datatype between user process and the system.

Level	Optname	Get	Set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	*	*	Permit Sending Of Broadcst Datagrams	*	Int
	SO_DEBUG	*	*	Enable debug tracing	*	int
	SO_ERROR	*		Get Pending Error And Clear		int
	SO_TYPE	*		Get Socket Type		int
	SO_LINGER	*	*	Linger on close if data to send		Linger{}
	SO_RCVTIMEO	*	*	receive time out		Timevalue{}
IPPRTO_IP	IP_TOS	*	*	Type Of Service And Precedence		Int
	IP_MULTICAST_IF	*	*	Specify outgoing interface		Int_addre{}
IP_PROTOICMPV6	ICMPV6_FILTER	*	*	Specify ICMPv6 message types to pass		Icmp6_ffilter
IPPROTO_IPV6	IPV6ADDFORM	*	*	Change address format of socket		Int
IPPROTO_TCP	TCP_KEEPALIVE	*	*	Seconds Between Keep alive Probes		Int

The code in 7.2, 7.3 and 7.4 provides a method to to find out if the given option is supported and if so to print the default value.

GENERIC SOCKET OPTIONS:

These socket options are protocol independent meaning that the protocol independent code within the kernel handles these and not particular module of any protocol. Some options apply to only certain types of sockets. For example, SO-BROADCAST socket options applied only to datagram socket although it is listed under GENERIC Sockets.

SO_BROADCAST Socket Option: This socket option enables or disables the ability of the process to send broadcast messages. Broadcasting is supported for only datagram sockets and only on net works such as ethernet, token ring etc but not point to point networks. This option controls whether datagrams may be broadcast from the socket. The value has type `int`; a nonzero value means "yes".

SO_DEBUG :The option is supported by TCP only. When enabled for a TCP socket, the kernal keeps track of all the packets sent or received by TCP for the socket.

SO_DONTROUTE Socket option: The option specifies that outgoing packets are to bypass the normal routing mechanism of the underlying ptotocol. With Ipv4, the packet is directed to the appropriate local interface, as specifried by the network and subnet protions of the destination address. If the local interface cannnot be determined from destination address, ENETUNREACH is returned.

SO_ERROR options: This option can be used with `getsockopt` only. It is used to reset the error status of the socket. When a errors occurs on a socket, the protocol module sets a variable named so_error for that socket to one of the standard unix values. The processs is immediately notified. The process can then obtain the values of so-error by fetching the SO_ERROR socket option. After the receipt, the so_error value is reset to 0 by the kernel.

SO_KEEPALIVE socket option: When the keep alsive socket option is set for a TCP socket, and if no data is exchanged across in either direction for two hours TCP automatically sends a keepalive probe to the peer. The probe is a
TCP segment to which the peer must respond. The possible three scenarios ae ;

- The peer responds with expected ACK. The application is no notified but the TCP sens another probe after 2 hours.
- The peer responds with RST which tells the local TCP that the peer host has crashed and rebooted.. The socket's pending error is set to ECONNRESET and the socket is closed.
- There is no response from the peer. TCP sends eight additional probes, 75 sec apart . If there is no response within 11 min a dn 15 se after first probe, the socket is sent with ETIMEOUT and the socke tis closed.

The purpose of this option is to detect if the peer host crashes. IF the peer host crashes, its TCP will send FIN across the connection which can easily detect with select.

SO_LINGER Socket Option :

The option specifies how the close function operates for a connection oriented protocol (TCP). 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 top the peer.

The SO_LINGER socket option lest us change this default. This option requires the folowing structure to be passed between the user process and the kernel. It is is defined by including <sys/socket.h>

```
Struct linger {
        Int l_onoff;   /*0=off, nonzero = on */
        Int l_linger ;  /* liner time, posix 1g specifies uits as sec*/
        }
```

Calling setsockopt leads to one of the following three scenarios depending on the values of the two structure.

1. if l_onoff is 0, the option is turned off. The value of l_linger is ignored and the previously discussed TCP defaults applies. Close returns immediately.

2. IF l_onoff is nonzero and l_linger is 0, TCP aborts the connection when it is closed. That is TCP discards any data still remaining in the socket send buffer and an RST to the peer.
3. IF l_onoff is nonzero and l_linger is nonzero, the kernel will linger when the socket is closed. Thst is if ther is any data still remaining in the socket send buffer, the porcess is put to sleep until either a) all the data is send and acknowledged by the peer. B) the linger time expires.

Let us understand when the close on a socket returns, given the various scenarios that we have seen so far.

Assue that the client writes data to the socket and then calls close as shown in the folowing figure. When the clidnt data arrives, the server is temporarily busy, so the data is added to the socket receive buffer by its TCP. Similarly the next segment, the client's FIN is also addded to the socket receive buffer. But by default, the client's close returns immediately.

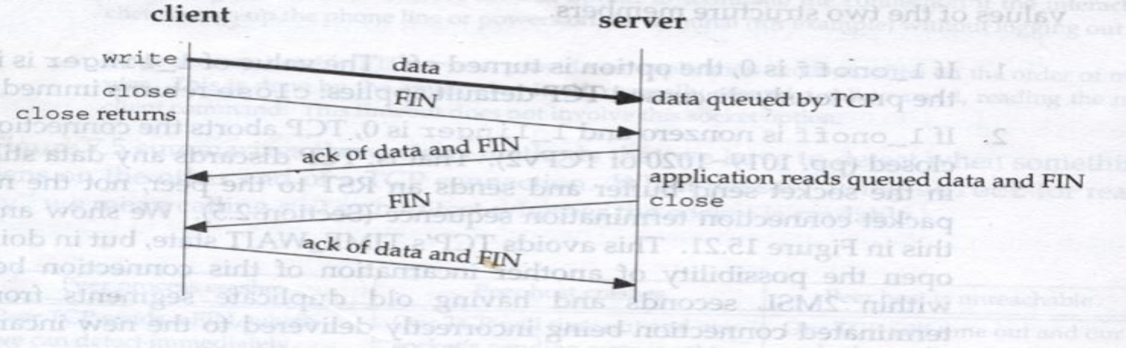


Figure 7.6 Default operation of close: it returns immediately.

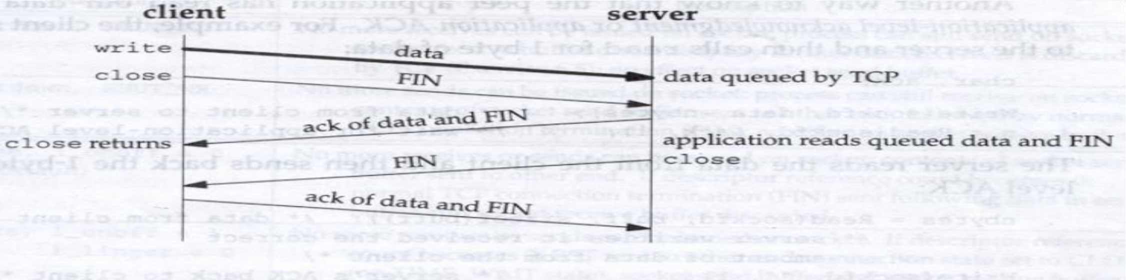


Figure 7.7 close with SO_LINGER socket option set and l_linger a positive value.

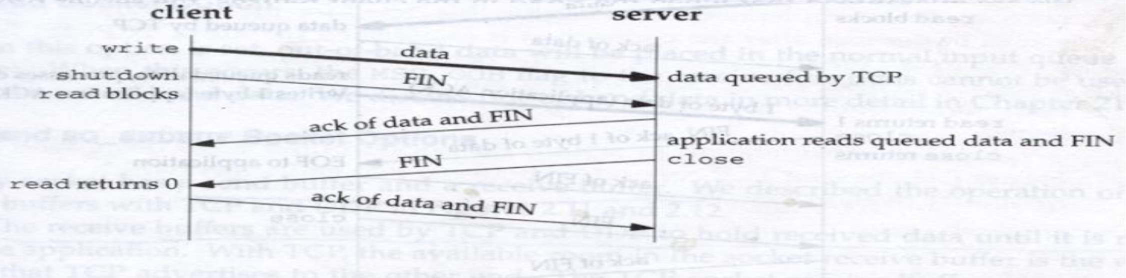


Figure 7.8 Using shutdown to know that peer has received our data.

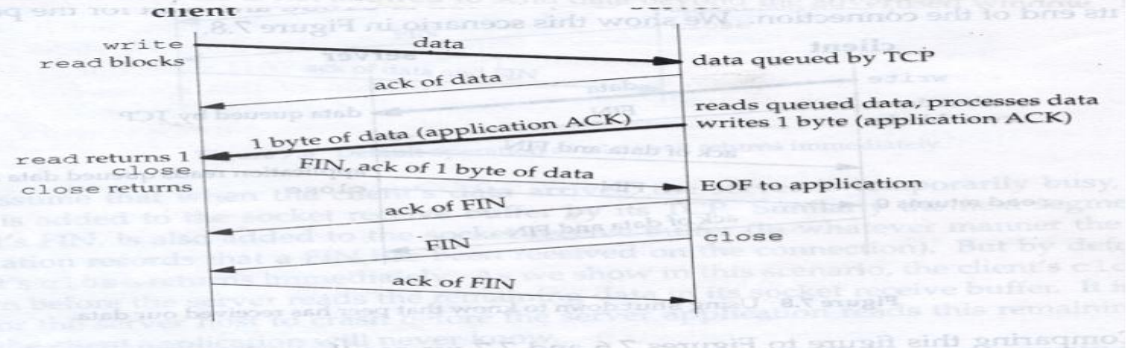


Figure 7.9 Application ACK.

SO-OOBINLINE socket option:

When This option is set, out of band data will be placed in the normal input queue. Whne this occurs, the MSG_OOB flag to the receive functions cannot be used to read the out of band data.

SO_RCVLOWAT and SO_SNDFLOWAT Socket Options:

Every socket has a receive low mark and a send low water mark. These are used in select function. These two socket opitons les us to change these options.

Receive low water mark is the amount of data that must be in the socket receive buffer for select function to be readable. If defalut to 1 for a TCP and UDP sockets. The send low water mark is the amount of available space that must exist in the socket send buffer for a select function to return writable. This low watermarl bnormally defaults to 2048 for TCP sockets.

SO_CVTIMEO AND SO_NDTIMO socket options: These two socket options allow us to place a timeout on socket receives and sends. Notice that the arguments to the two sockopt functionsis a pointer to a timeval structure, the same one used with select. This lets specify the timeout in seconds and microseconds.

The receive timeout afffects the five input functions : read, readv, recv, recvfrom and recvmsg. The send timeout affects the five output functions : write, writev,send, sendto, and sendmsg.

16.12.2 Socket-Level Options

— Constant: int **SOL_SOCKET**

Use this constant as the *level* argument to `getsockopt` or `setsockopt` to manipulate the socket-level options described in this section.

Here is a table of socket-level option names; all are defined in the header file `sys/socket.h`.

`SO_REUSEADDR`

This option controls whether `bind` (see Setting Address) should permit reuse of local addresses for this socket. If you enable this option, you can actually have two sockets with the same Internet port number; but the system won't allow you to use the two identically-named sockets in a way that would confuse the Internet. The reason for this option is that some higher-level Internet protocols, including FTP, require you to keep reusing the same port number.

The value has type `int`; a nonzero value means “yes”.

`SO_KEEPALIVE`

This option controls whether the underlying protocol should periodically transmit messages on a connected socket. If the peer fails to respond to these messages, the connection is considered broken. The value has type `int`; a nonzero value means “yes”.

`SO_DONTROUTE`

This option controls whether outgoing messages bypass the normal message routing facilities. If set, messages are sent directly to the network interface instead. The value has type `int`; a nonzero value means “yes”.

`SO_LINGER`

This option specifies what should happen when the socket of a type that promises reliable delivery still has untransmitted messages when it is closed; see Closing a Socket. The value has type `struct linger`.

— Data Type: **struct linger**

This structure type has the following members:

`int l_onoff`
This field is interpreted as a boolean. If nonzero, `close` blocks until the data are transmitted or the timeout period has expired.
`int l_linger`
This specifies the timeout period, in seconds.

`SO_BROADCAST`
This option controls whether datagrams may be broadcast from the socket. The value has type `int`; a nonzero value means "yes".

`SO_OOBINLINE`
If this option is set, out-of-band data received on the socket is placed in the normal input queue. This permits it to be read using `read` or `recv` without specifying the `MSG_OOB` flag. See Out-of-Band Data.The value has type `int`; a nonzero value means "yes".

`SO_SNDBUF`
This option gets or sets the size of the output buffer. The value is a `size_t`, which is the size in bytes.

`SO_RCVBUF`
This option gets or sets the size of the input buffer. The value is a `size_t`, which is the size in bytes.

`SO_STYLE`
`SO_TYPE`
This option can be used with `getsockopt` only. It is used to get the socket's communication style. `SO_TYPE` is the historical name, and `SO_STYLE` is the preferred name in GNU. The value has type `int` and its value designates a communication style; see Communication Styles.

`SO_ERROR`

This option can be used with `getsockopt` only. It is used to reset the error status of the socket. The value is an `int`, which represents the previous error status.

http://www.unet.univie.ac.at/aix/aixprggd/progcomc/toc.htm

SO_RCVBUF /SO_SNDBUF

- Integer values options - change the receive and send buffer sizes.
- Can be used with STREAM and DGRAM sockets.
- With TCP, this option effects the window size used for flow control – must be established before connection is made.

SO_REUSEADDR

Boolean option: enables binding to an address (port) that is already in use.

- Used by servers that are transient - allows binding a passive socket to a port currently in use (with active sockets) by other processes.

Can be used to establish separate servers for the same service on different interfaces (or different IP addresses on the same interface).

- Virtual Web Servers can work this way.

KEEPALIVE socket Option: The purpose of this option is to detect if the peer host crashes. This option is usually used by servers, although client can also use this option. Server uses this option because they spend most of their time blocked wiating for the input across the TCP connection, that is waiting for the client request. But if the client host crashes, the server process will never know about it and it will wiat continuosly for input data that can never arrriver. This is called half open connection. The keep alive option will detect these half open connections and terminates them.

3. If there is no response from the peer to the keepalive probe, TCP sends eight additional probes, 75 sec apart trying to elicit response. TCP will give up if ther is no response within 11 minutes and 15 seconds after sending the first probe. If there is no response at all to TCPs keepalive probes, the socket's pending error is sent to ETIMEDOUT and the socket is closed. But if the socket receives an ICMP response to one of the keepalive probes, the socket corresponding error is returned instead such as EHOSTUNREACH error.

SO_LINGER Socket option: This option specifies how the close function operates for a connection oriented protocol (that is for TCP) By default, close function returns immediately with ack, but if there is any data still remaining the in the socket sedn buffer, the system will try to deliver the data to the peer.

The SO_LINGER socket option les us change this default. This option requires the folowing structure to be passed between the user process and the kernel. It is defined as

```
Struct linger {
        Int l_onoff; /* 0 for Off and Nonzero for ON*/
        Int l_linger;  /*Linger time Posix 1g specification as seconds*/
        }
```

Calling setsocket leads to one of the folowing threee scenarios depending on the values of the two structure memebrs.

1. If l_onoff is 0, the option is turned off. The value of l_linger is ignored and the preveiously discussed TCP default applies: close returns immediately.
2. If l_onoff is non zero and l_linger is 0, TCP aborts the connection when it is closed. That is TCP discards any data still remaining in the socket send buffer amd sends an RST to the peer. (Not the four step termination sequence) This avoids the TCP's TIME_WAIT state.
3. If l_onoff is nonzero and l_linger is non zero, then the kernel will linger when the socket is closed . That is, if ther is any bdata stilol remaining in the socket send buffer, the process is put to sleep until either i) all the data is sent and acknolodged by the peer TCP ii) the linger time expires

   When using this feature of the SO-LINGER option, it is important for the application to check the return value from close because if the linger time expires before the remianing data is sent and acknowledged, claose returns EWOULDBLOCK and any remaining data in the send buffer is discarded.

SO_RCVBUF and SO_SNDBUF Socket Option:

The receive buffer are used by the TCP and UDP to hold received data until it is read by the application. With TCP, the available room in the socket receive buffer is the window that TCP avertises to the other end. Hence the the peer sends only that amount of data and any data beyond that limit is discarded. In case of UDP, the bufffer size is not advertised hence, any data that do not fit ito the buffer, are dropped However, the abovementioned socket options allows one to change the default sizes. The default values for the TCP and UDP differ for different implementation. It is normally 4096 for TCP and send buffer for UDP is 9000 and 40000 bytes for receive buffer.

SO_REUSEADDR and **SO_REUSEPORT:**

The SO_REUSEADDR serves four purposes :

This option allows a listening server to start and bind its well known port even if previously established connections exists that use this port as their local port. As the server is in listening state, when connection request comes from a client, a child process is spawned to handle that client. Wqith this listening server terminates. Once again when the listening is restarted buy calling socket, bind and listen, the call to bind fails because the listening server is trying to bind a port that is part of exisitng connection. But if the srver is sets the SO_REUSEADDR socket option between the calls to socket and bind, the latter function will succeed.

This allows multiple instances of the same service to be started on the same port as long as each instance binds a different local IP address. It is common of a a site to host multiple http servers using the IP alias techniques. If the primary address is 198.69.10.2 and it has two aliases as 198.69.10.129 and 198.69.10.128. Three HTTP servers are started. Whne the first connection request comes, the server binds the call with 198.69,10.128 and a local port of 80. The second request is connected to 198.69.10.129 provided the SO_REUSEADDR is set before the call to bind.
Simlarly for the final http server also.

It also allows a single process to bind the same port to multiple socket as slong as each bind specifies a different loacl IP address It also allows completely duplicate binding: a bind of an IP address and port, when the same IP address and port are already bound to another socket. This happens with the protocol that support multicasting (UDP)

SO_ TYPE socket Option: This option returs the socket ytpe. The integer value returned is a value such as SOCK _ STREAM or SOCK_DGRAM.

SO_USELOOPBACK Socket Option: When this option is set, the socket receives a copy of everything send on the socket.

IPv4 Socket options: The level of this options are IPPROTO_IP.

IP_HDRINC Socket Option : If this socket is set for a raw socket, we must buid our own IP header for all datagrams that we send on the raw socket. Normally kernel builds the headers for datagrams sent on raw socket. But for some applications, build there own IP address to override that IP would place into certain header fields. (Traceroute).

IP_OPTIONS Socket Options: Setting this option allows us to set the IP option in the Ipv4 header. This requires intimate knlowledge of the format of the IP options in the IP header.

IP_RECVDSTADDR Socket Options :

This socket options causes the destination IP address of a received UDP datqgram to be returned as ancillary data by recvmsg.

IP_RECVIF Socket Options :

This socket option causes the index of the interface on which a UDP datagram is received to be returned as an ancillary data by recvmsg.

IP_TOS Socket Option :

This options lets us set the type of field service in the IP header for a t
CP or UDP socket. The different value to which this o[tions can be set are given below:

Constant	description

IPTOS_LOWDELAY	minimize delay.
IPTOS_THROUGHPUT	maximize throughput
IPTOSRELIABILITY	maximize reliability.
IPTOS_LOWCOST	Minimize cost.

For telnet login shoul;d specify IPTOS_LOWDEAY while the data portion of an FTP transfer should specify IPTOS_ THROUGHPUT

IP_TTL Socket Option :

With this option we can set and fetch the default TTL (time to live field) that the system will use for a given socket (*64 for TCP and 255 for UDP)

ICMPv6 Socket Option : The level is of **IPPROTO_ICMPV6**

ICMP6_FILTER: This option lets us fetch and set an icmp6_filter structure that specifies which of the 255 possible ICMPV6 message types are passed to the process on a raw socket.

IPv6 Socket Options:

These are processed by IPv6 and have a level of IPPROTO_IPV6.

IPV6_ADDRFORM Socket Option:

This option allows a socket to be converted from IPv4 to IPv6 or vice versa.

IPV6_CHECKSUM Socket Option: This socket option specifies the byte offset into the user data of where the checksum field is located. If this value is nonnegative, the kernel computes and store the checksum for all outgoing pockets. And verify the received checksum on input, discarding packets with invalid checksum. If the value is set to -1, the kernel will calculate and store the checksum for outgoing packets.

IPV6_DSTOPTS Options : This options lets any received IPv6 destination options are to be returned as ancillary data by ***recvmsg.***

IPV6_HOPOPTS: Setting this option specifies that the received IPv6 hop by hop options are to be returned as ancillary data by recvmsg.

IPV6_HOPLIMIT : Setting this option specifies that the received hop limit field be returned as ancillary data by **recvmsg.**

IPV6_NEXTMSG : This is a not a socket option but the type of an ancillary data object that can be specified to ***sendmsg.*** This opjet specifies the next hop address for a datagram as a socketaddress structure.

IPV6 _PKTINFO: Setting this option specifies that the following two pieces of information about a received IPv^ datagram ae to be returned as ancillary data by ***recvmsg.***

IPV6_PKTOPTIONS: Most of the IPv6 socket options assume a UDP socket with the information being passed between the kernel and the application using ancillary data with ***recvmsg and sendmsg.*** A TCP socket fetches and stores these values using the IPV6_PLTOPTIONS socket options.

IPV6_RTHDR: Setting this options specifies that a received IPV6 routing header is to be returned as ancillary data by ***recvmsg.***

IPV6_UNICAST_HOPS : This IPv6 option is similar to the IPv4 IP_TTL socket option. Setting the socket option specifies the default hop limit for outgoing datagram sent on the socket, whcile fetching the socket options returns the value for the hop limit that the kernel use fo the socket. To obtain actual hop limit field, IPV^_HOPLIMIT socket option is used.

TCP SOCKET OPTIONS: The leve is IPPROTO_TCP
TCP_KEEPALIVE socket options : It specifies the idle time in seconds for the connections before TCP starts sending keepalive probes. Default value is 7200 sec (2 hours) This is effective when SO_ ***KEEP ALIVE option is enabled.***

TCP_MAXRT Socket Options : It specifies the amount of time in seconds before a connections is broken once TCP starts transmitting data. A value of 0 means touse the system default. And a value of –I means to rettransmit forever. If a positive value is specified, it may be rounded up to the implementations next transmission time.

TCP_MAXSIZE Socket Option: This socket option allows us to fetch or set the maximum segment size for a TCP connection. Often it is the MSS value announced by the peer process.

TCP_NODELAY Socket Option :
If this option is set, TCP's Nagle's algorithm is disabled.
Nagle's Algorithm: The algorithm states that if a given connection has outstanding data (that is, data that our TCP has sent and for which it is currently awaiting an ack), then no small packet will be sent on the connection until the existing data is acknowledged. Small packet is any size less than MSS. The purpose is to prevent a connection from having multiple packets outstanding at any time. This situation becomes more prominenet in a WAN with Telnet.

Consider the example where in we type Hello to Telnet client. Let this take 250 ms between each letter (as shown below) The RTT to the server and back is 600 ms and the server immediately sends the echo of the character. We assume that ACK of the client character is sent back to the client along with the character echo and we ignore the ACKs that the client sends for the server echo. Assuming Nagle's algorithm disabled, we have the following figure:

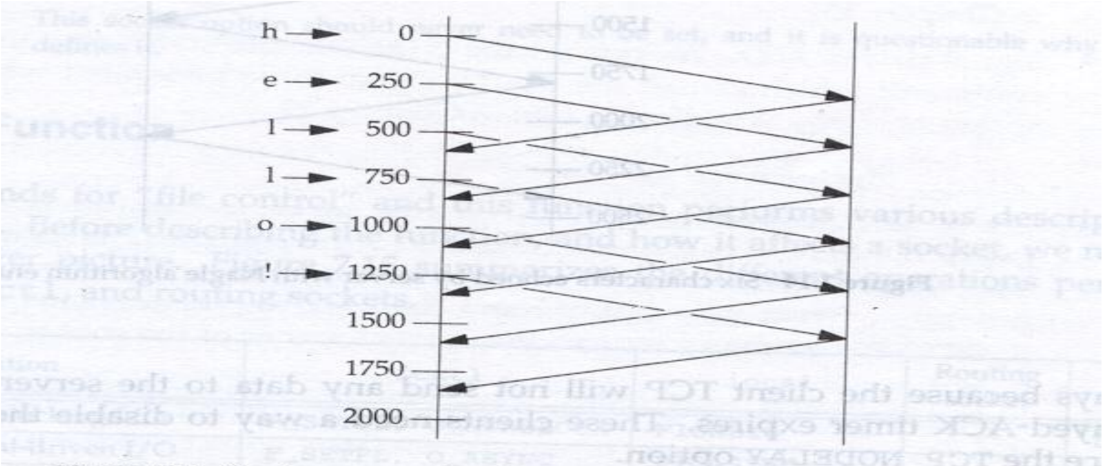


Figure 7.13 Six characters echoed by server with Nagle algorithm disabled.

IN this each packet is sent in a packet itself.

But if the Nagle's algorithm is enabled (default), we have the six packets as shown in the following figure:

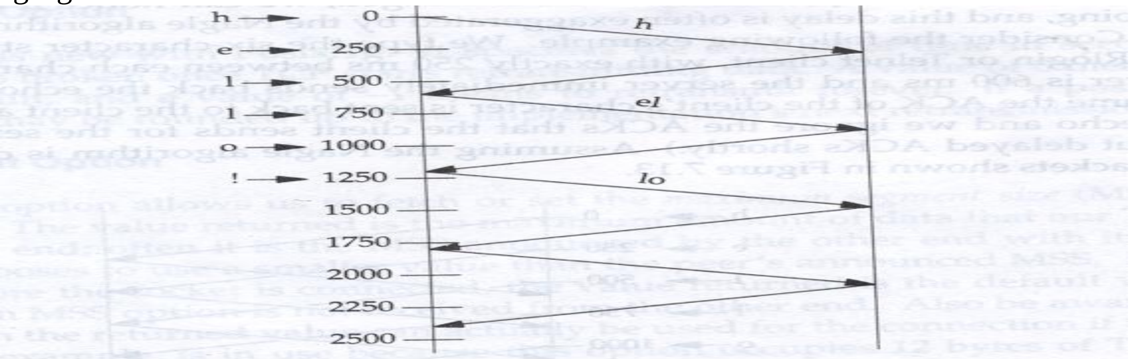


Figure 7.14 Six characters echoed by server with Nagle algorithm enabled.

The first packet is sent as a packet by itself., but the next two characters are not sent, since the connection has small packet outstanding. At time 600 ms when the ACK of the first packet is received, along with the echo of the first character, theset two characters are sent. Until this packet ACKed at time 1200, no more small packets are sent.

When the Nagles algorithm oftern interacts with another TCP algorithm called 'delayed ACK' algorithm. This algorithm causes the TCP to not send an ACK immediately whne it receives data; instead TCP will wait some small amount of time and only then send ACK. The hope is that in this small time (50 – 200 ms) there will be more data to be sent back to the peer and the ACK can piggy back on the data saving the TCP segment. This is the normally case with Telnet. Therefore, they wait for the echoed data to piggy back.

In all these cases, the TCP_NODELAY socket option is set.

TCP_STDURG socket option: If this is set, then the the urgent pointer will point to the data byte sent with the MSG_OOB flag

fcntrl Function : This stands for file control. This control performs various deascriptors control operations. Following table summarised some of the key file operations. These are preferred way under Posix 1g.

Opeations	Fcntl
Set socket for non blocking I/O	F-SETFL, O_NONBLOCK
Set socket for signal driven I/O	F_SETFL, O_ASYNC
Set socket owner	F-SETOWN
Get socket owner	F-GETOWN

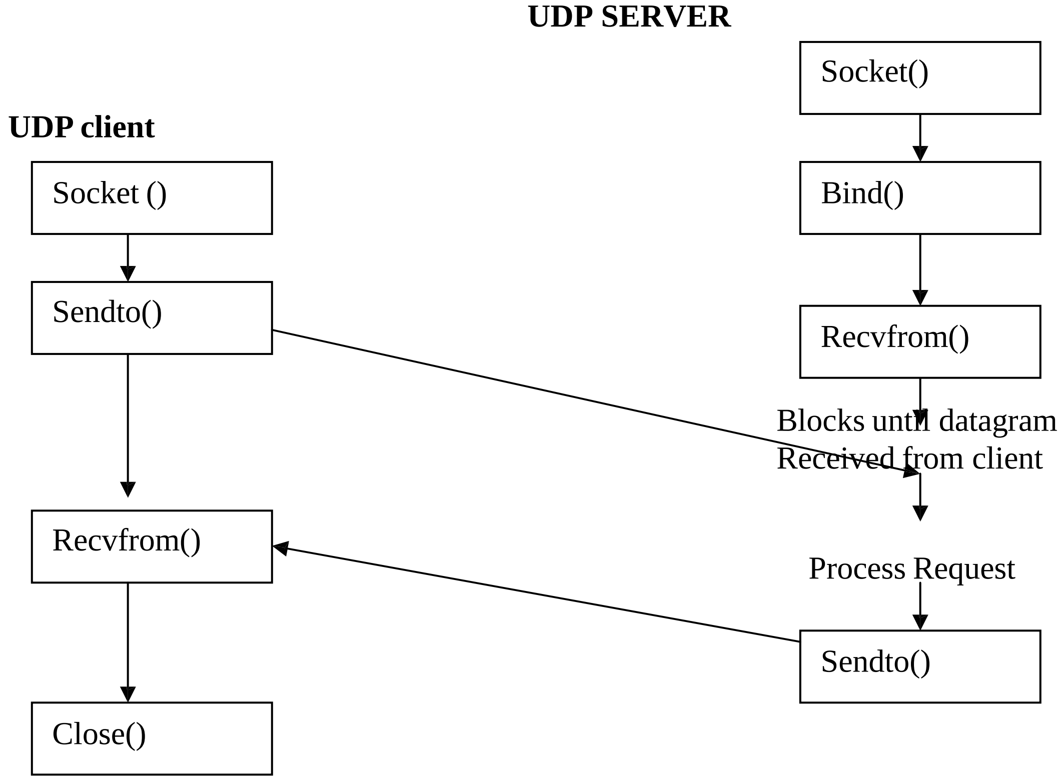
These features are provided in the following way.

- Non blocking I/O: We can set the O_NONBLOCK file status flag using the F_SETFL command to set a socket nonblocking.
- Signal Driven I/O : We can set the O_SYNC file status by using F-SETFL command which caused the SIGIO signal to be generated when the status of a socket changes.
- The F_SETDOWN command lets us set the socket owner to receive the SIGIO and SIGURG signals. SIGIO is generated when the signal driven I/O is enabled for a socket and the latter is generated with new out of band data arrives for a socket. The F-GETOWN command returns the current owner of the the socket.

```
#include <fcntl.h>
int fcntl (int fd, int cmd, …)  returns: depends on cmd if OK, -1 on error.
```

ELEMENTS OF UDP SOCKET

The significant difference between TCP and UDP applications are in the Transport layer. UDP is connectionless, unreliable, datagram protocol. However, there are needs for such requirements in applications such as DNS, NFS and SNMP. Typical functions calls of UDP client server are shown below:



IN this, client does not establishes connection with server, rather it sends datagram using send to function along with destination address. Similarly, the server does not accept connection from a client, instead the server just calls 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.

Recvfrom and sendto functions :

```
#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, struct const sockaddr  *to,
socklen_t *addrlen)
```

sockfd, buff and nbytes are identical to the first three arguments for read and write: descriptors, pointer to buffer to read into or write from, and number of bytes to read or write.

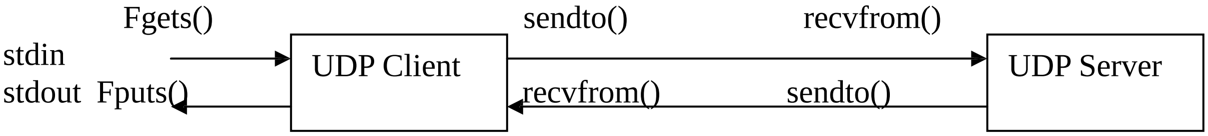
The flags are meant are normally used with recvmsg and sendmsg functions. IN this function they are defaulted to the value of 0. The to argument for the sendto is a socket address structure containing protocol address (IP and port) of where data is to be sent. The recvfrom functions fills in the socket address structure 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 the addlen.

Final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value result - argument.)

The final two argument of recvfrom is similar to the two argument to accept : the contents of the socket address structure upon return tell us who sent the datagram (in case of UDP)or who initiated the connection in the case of TCP. Both function return the length of the data that was read or written as the value of the funtion.

In the case of writing a datagram of length 0 is OK. IN UDP, this means, 20 byte length of IP header of IPV4, 8 byte UDP header and no data. It is accepted unlike TCP where in a 0 is consider as EOF.

UDP Echo Server : Main function: The function call of UDP client and server is shown in the following figure.



The main server program is shown below:

```
#include    "unp.h"

int
main(int argc, char **argv)
{
      int                             sockfd;
      struct sockaddr_in      servaddr, cliaddr;

      sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

      bzero(&servaddr, sizeof(servaddr));
      servaddr.sin_family      = AF_INET;
      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
      servaddr.sin_port        = htons(SERV_PORT);

      Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

      dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
}
```

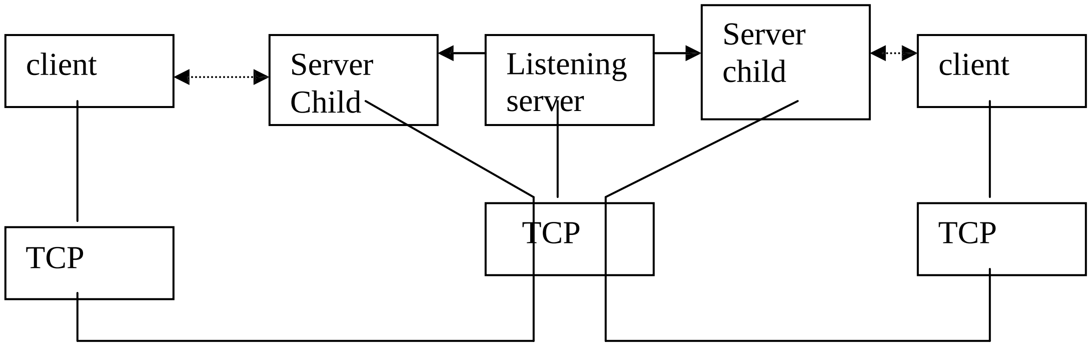
IN this UDP socket is created by giving SOCK_DGRAM . The address for the bind is given as INADDR_ANY for multihomed server. And the const SERVER_PORT is the well known port.

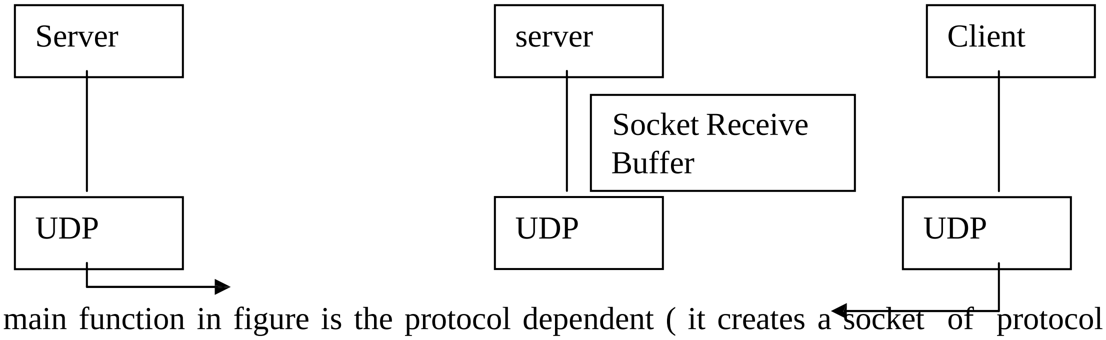
UDP Echo Server : dg_echo function: The programme is given below:

```
 void
dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
      int n;
      socklen_t   len;
      char        mesg[MAXLINE];

            for ( ; ; ) {
            len = clilen;
            n = recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
            sendto (sockfd, mesg,n,0,pcliaddra, len);

      }
}
```

The important details to consider are:

1. The function never terminates (exit (0) is not called) as it is connection less protocol
2. The main function is iterative server not concurrent. There is no call tofokr, so a single server process handles any and all clients.
3. There is implied queuing takes place in the UDP layer for the socket. Each datagram that is arrived is received in a buffer from where the recvfrom receives the next datagram in FIFO manner. The size of the buffer may be changed by changing the value of SO_RCVBUF





The main function in figure is the protocol dependent (it creates a socket of protocol AF_INET) and allocates and initializes an IPv4 socket address structure. But the dg_echo function is protocol independent as it is provided with the socket address structure and its length by the main function. It is the recvfrom that fills in this structure with the IP address and prot number of the client and as the same pointer is passed to sendto as the destination address.

UDP Echo Client :

udpcliserv/ udpcli01.c Page No: 216

```
#include        "unp.h"

int
main(int argc, char **argv)
{
        int                                     sockfd;
        struct sockaddr_in      servaddr;
        if (argc != 2)
                err_quit("usage: udpcli <IPaddress>");
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(SERV_PORT);
        Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
        sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
        dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
        exit(0);
}
```

An IPv4 socket address structure is filled in with the IP address and port numbers of the server. This structure is passed on to dg_cli.

A UDP socket is is created and the function calls dg_cli.

Dg_cli Function

```
#include        "unp.h"

void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
        int             n;
        char    sendline[MAXLINE], recvline[MAXLINE + 1];

        while (Fgets(sendline, MAXLINE, fp) != NULL) {

                sendto (sockfd, sendline, strlen(sendline, 0, pservaddr, servlen);

                n = recvfrom(sockfd, recvline, MAXLINE,0,NULL,NULL);

                recvline[n] = 0;        /* null terminate */
                Fputs(recvline, stdout);
        }
}
```

There are four steps in the client processing loop. Read a line from the standard input using fgets, send the line to the server using sendto, read back the server’s echo using recvfrom and print the echoed line to standard output using fputs

With a UDP socket, when the first time the call to sendto() is given, the port is bound to it. Also similar to server, client can also call for bind() to tie up its port.

In the recvfrom(), a null pointer is specified in the fifth and sixth argument. Which means that we are not interest in knowing who has send the datagram. This may result in error in reception of datagram, as the same host or any other host may send datagram which is likely to be mistaken.

Similar to dg_echo(), dg_cli() is also protocol independent. In this the main function allocates and initializes a socket address structure of some protocol type and then passes pointer to this structure along with its size.

Lost Datagrams:

Above client server example is not reliable. If the client datagram is lost, (may be dropped at any router), the client will block its recvfrom() forever waiting for a server reply that will never arrive. Similarly if the client datagram arrives at the server but the server's reply is lost, the client will once again block the recvfrom(). A time out and a appended number is normally sent with the datagram which is sent back as acknowledgment with reply. This will assure whether the datagram is received or not. This facility is provided many UDP servers.

Verifying the Received Response : It is seen from the above example that any process that knows the clients ephemeral port number could datagram to our client and these will be interpreted with the normal server replies. It can be avoided by comparing the IP address and port number of the recvfrom() (reply received from server), with that of the IP address and port number to which it was sent earlier. This requires creating a socket address structure in which the return address structure is returned and compared with the sent address structure. This is shown in the following example

udpcliserv/dgcliaddr.c **Page No : 219**

```
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
        int                         n;
        char                sendline[MAXLINE], recvline[MAXLINE + 1];
        socklen_t           len;
        struct sockaddr     *preply_addr;
        prpeply_addr = malloc(servlen);
        while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
        Len = servlen;
        n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
        if (len != servlen|| memcmp (pservaddr, preply_addr, len)!= 0)
        {printf (reply from %s (ignored)\n"),sock_ntop (preply_addr, len);
          continue;
        }
        Recvline[n] = 0;    /* null terminate */
        Fputs(recvline, stdout);
        }
}
```

First the, client main function is changed to use the standard echo server. **Servaddr.sin_port = htons(SERV_PORT)** changed to **servaddr.sin_port = htons (7).** Malloc() function allocates the byte size to the structure. With this arrangement, the received address captured in preply_addr is compared with the one that was sent in sendto(). First a comparison is made of its length and then memcopy compares then socket address itself is compared. This will assure if the server address is same.

However, even this may not work if the server is multi homed where in it is likely to allocate different IP address as the server has not bound an IP address to its socket and it chooses any IP Address. Hence this may also fail.

One solution is for the client to verify the responding host's domain name instead of its IP address by looking up the server's name in the DNS, given the IP address returned by recvfrom().

Another solution is for the server to create one socket for every IP address that is configured on the host , bind that IP address to the socket, use select across all these sockets (waiting for one to be readable) and then reply from the socket that is readable. Since the socket used for the reply was bound to the IP address that was the destination address of the client request, this guarantees that the source address for the reply is the same as the destination address of the request.

Server not Running :

If the client sends a datagram to a server that is not yet started, what happens to it? The recvfrom() will be blocked for ever waiting for a reply from the server. When the call to sendto() is given by passing the massage to it, the function returns OK (size of message to application) which means that datagram is put in the output interface queue for further transmission. Where as the datagram traveling to server that is not switched on, is responded with an ICMP port unreachable error. As this error takes more time to return (equal to RTT), this error is known as asynchronous error - not synchronous with the error causing source, which in this case is sendto().

This error is not returned to UDP socket at the client. Hence, the recvfrom() is left open. The asynchronous error can be returned only when the socket have been connected which requires a call to connect() by the client.

connect() with UDP:

As it was seen that asynchronous errors are not returned to UDP sockets unless the socket has been connected. It is possible to give a call to connect() in UDP also. But it does involved few changes that are listed below:

1. When connect() is called, the destination IP address and port number of the server host is provided. Hence there is no requirement to pass these values in sendto(). Hence, the functions sendto() is replaced with write() or send() functions. This assumes that anything written is automatically send to the server address specified in the connect function.

2. Similarly, recvfrom() is also not used instead recv() or read() functions are used. The only datagram returned by the kernel for an input operation on a connected UDP socket are those arriving from protocol address specified in the connect. Datagram destined to the connected UDP sockets local protocol address but arriving from a protocol address other than the one to which the socket was connected, are not passed to the connected socket. This limits the connected UDP socket to exchanging datagram with one and only peer.

3 Asynchronous errors are returned to the process for a connected UDP socket. Also an unconnected UDP sockets does not receive any asynchronous error.

It can be said that UDP client or server can call connect() only if that process uses UDP socket to communicate with exactly one peer.

Calling connect () Multiple Times for a UDP socket:

A process with a connected UDP socket can call connect() function again for that socket to either

- Specify a new IP address and p ort or to,
- Unconnect the socket.

Unlike in the case of TCP, where in the connect() is called only once, in the case of UDP, it can be called again.

To unconnect, the connect is called by setting the family members of the socket address structure (sin_family for IPv4 and sin6_family for IPv6) to AF_UNSPEC. This might return EAFNOSUPPORT error but it is accepted.

Performance:

When an application calls sendto() on an unconnected UDP socket, Berkeley derived kernels temporarily connect the socket, and datagram and then un connect the socket . Calling sendto for two datagram on unconnected UDP socket then involves the following six steps:

- Connect the socket.
- Output the first datagram.
- Unconnect the socket.
- Connect the socket.
- Output the second datagram
- Unconnect the socket.

When the application knows that it will be sending multiple datagram to the same peer, it is more efficient to connect the socket explicitly by calling connect() function and then call write() function as many times. As shown below:

- Connect the socket/
- Output the first datagram and
- Output the second datagram.

IN this case the kernel copies only the socket address structure containing the destination IP address and port one time, versus two times when sendto is called twice.

The dg_cli function with call to connect function is given below:

```
#include        "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{       int             n;
        char    sendline[MAXLINE], recvline[MAXLINE + 1];
        Connect(sockfd, (SA *) pservaddr, servlen);
        while (Fgets(sendline, MAXLINE, fp) != NULL) {
                Write(sockfd, sendline, strlen(sendline));
                n = Read(sockfd, recvline, MAXLINE);
                recvline[n] = 0;            /* null terminate */
                Fputs(recvline, stdout);
        } }
```

The changes are the new call to connect and replacing the calls to sendto() and recvfrom() with calls to write() and read(). This functions are still protocol independent.

TCP and UDP Echo Server using Select ():

Following example combines the concurrent TCP echo server with iterative UDP echo server into a single server using select function to multiplex the TCP and UDP socket.

```
/* include udpservselect01 */
#include        "unp.h"
int
main(int argc, char **argv)
{       int                     listenfd, connfd, udpfd, nready, maxfdp1;
        char                    mesg[MAXLINE];
        pid_t                   childpid;
        fd_set                  rset;
        ssize_t                 n;
        socklen_t               len;
        const int               on = 1;
        struct sockaddr_in      cliaddr, servaddr;
        void                    sig_chld(int);
        /* 4create listening TCP socket */
        listenfd = Socket(AF_INET, SOCK_STREAM, 0);
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family      = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port        = htons(SERV_PORT);
        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
        Listen(listenfd, LISTENQ);
                /* 4create UDP socket */
        udpfd = Socket(AF_INET, SOCK_DGRAM, 0);
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family      = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port        = htons(SERV_PORT);
        Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));
/* end udpservselect01 */
/* include udpservselect02 */
        Signal(SIGCHLD, sig_chld); /* must call waitpid() */
        FD_ZERO(&rset);
        maxfdp1 = max(listenfd, udpfd) + 1;
        for ( ; ; ) {
                FD_SET(listenfd, &rset);
                FD_SET(udpfd, &rset);
                if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
                        if (errno == EINTR)
                                continue;               /* back to for() */
                        else
                                err_sys("select error");
                }
                if (FD_ISSET(listenfd, &rset)) {
                        len = sizeof(cliaddr);
                        connfd = Accept(listenfd, (SA *) &cliaddr, &len);
                                if ( (childpid = Fork()) == 0) {        /* child process */
                                Close(listenfd);        /* close listening socket */
                                str_echo(connfd);       /* process the request */
                                exit(0);
                        }
                        Close(connfd);                  /* parent closes connected socket */
                }
                if (FD_ISSET(udpfd, &rset)) {
                        len = sizeof(cliaddr);
                        n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);
                        Sendto(udpfd, mesg, n, 0, (SA *) &cliaddr, len);
                } }}
/* end udpservselect02 */
```

Create listening TCP socket

A listening TCP socket is created that is bound to the server's well known port. We set the SO_REUSEADDR socket option in case of connections exist on this port.

Create a UDP socket

A UDP socket is also created and bound to the same port. Even though the same port is used for the TCP and UDP sockets, there is no need to set the SO_REUSEADDR socket option before this call to bind because TCP ports are independent of UDP ports.

Establish a signal handler for SIGCHLD:

Establised the signal handler SIGCHLD because TCP connections will be handled by a child process.

Prepare for Select:

A descriptor set is initialized for select and maximum of two descriptors for which the select waits.

Call select:

We call select waiting only for readability on the listening TCP socket or readability on the UDP socket. Since our sig_chld handler can interrupt our call to select, we handle an error of EINTR

Hane the new client:

We accept a new client connection when the listening TCP socket is readable, fork a child and call our str_echo in the child.

Handle arrival of datagram.

If the UDP socket is readable, a datagram has arrived. We read it with recvfrom and send it back to the client with sendto().

Summary:

Converting echo-client server to use UDP instead of TCP was simple. But the features provided by TCP are missing: detecting lost packet and retransmitting, verifying responses and so on.

UDP socket can generate asynchronous errors that is errors that are reported some time after the packet was sent. IN TCP, these error are always reported to application but not in UDP

UDP has no flow control. But this is not a big restriction as the UDP requirement are built for request – response application.

gethostname () :

I n the application, as it is easy to input human readable DNS name instead of IP address, there is a need to convert the host names into IP address format. This is achieved by the function **gethostname ().** When called, if successful, it returns a pointer to a **hostent structure** that contains all the **IPv4** address or all **IPv6** address for the host.

The syntax is given below:

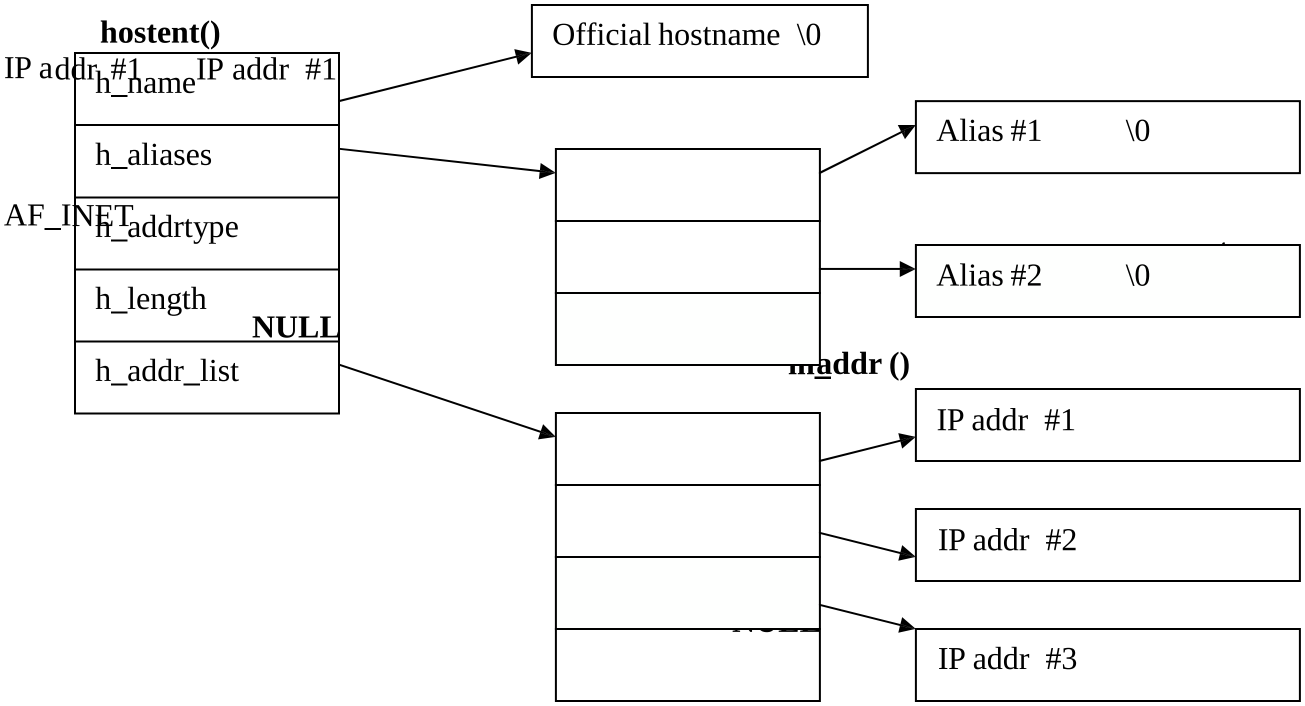
```
          #include <netdb.h>
          struct hostent *gethostbyname (const char * hostname);
                   returns nonnull pointer if OK , NULL on error with h_errorno set.
```

The non null pointer returned by this function points to the following **hostent** structure:

```
          Struct hostent {
                   char *h_name; /* official canonical) name of host*/
                   char  **h_aliases ;    /* pointer to array of pointers to alias names*/
                   int      h_addrtype ;  /* host address type: AF_INET or AF_INET6 */
                   int      h_length ;    /* length of address : 4  or 16*/
                   char ** h_addr_list    /* ptr to array of ptrs with IPv4 or IPv6 addrs*/
                   }
          #define h_addr  h_addr_list[0]         /* first address in list*/
```

In terms of DNS, gethostbyname() performs a querry for an A record or for a AAAA record. This function can return either IPv4 or IPv6 address.

Arrangement of hostent structure is shown below:



Similar structure is ther6 for IPv6 wherein h_addrtype is AF_INET6 and h_length is equal 6.

Returned h_name is called canonical name of the host. When error occurs, the functions sets the global integer h_errono to one of the following constants defined by including <netdb.h>.

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA (same as N0_ADDRESS)

NO_DATA error is valid and it indicates that the hostent has only MX record

Example:

```
#include        "unp.h"
int  main(int argc, char **argv)
{
        char                    *ptr, **pptr;
        char                    str[INET6_ADDRSTRLEN]; /* handles longest IPv6 address */
        struct hostent  *hptr;

        while (--argc > 0) {
                ptr = *++argv;
                if ( (hptr = gethostbyname(ptr)) == NULL) {
                        err_msg("gethostbyname error for host: %s: %s",
                                        ptr, hstrerror(h_errno));
                        continue;
                }
                printf("official hostname: %s\n", hptr->h_name);

                for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
                        printf("\talias: %s\n", *pptr);

                switch (hptr->h_addrtype) {
                case AF_INET:
#ifdef  AF_INET6
                case AF_INET6:
#endif
                        pptr = hptr->h_addr_list;
                        for ( ; *pptr != NULL; pptr++)
                                printf("\taddress: %s\n",
                                        Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
                        break;
                default:
                        err_ret("unknown address type");
                        break;
                }
        }
        exit(0);
}
```

IN this gethostname() is called for each command line argument. The official hostname output followed by the list of alias names. This program supports both IPv4 and IPv6 address type.

RES_USE_INET6 Resolver Option:

This option is used to tell the resolver that we want IPv6 addresses returned by gethostname() instead of IPv4 addresses.

- A application can set this option itself by first calling the resolver's re_init function and than enabling the option as shown below:
    - #include <resolv.h>
      res_init();
      res.options |=*RES*USE_INET6;

  This must be done before the first call to gethostbyname() or gethostbyaddr(). The effect of this option is only on the application that sets the option,.

- IF the environment variable RES_OPTION contains the string inet6, the option is enabled. It is set in the file **.profile** file with the **export attribute as in**

  export RES_OPTIONS = inet6;

This setting affects every program that we run from out login shell. But if it set in the command line, then if affects only that command.

- The resolver configuration file – normally **/etc/resolv.conf** – can contain the line

  options inet6

Setting this option ion the resolver configuration file affects all applications on the host that call the resolver functions. Hence this should not be used until all applications in the host are capable of handling IPv6 addresses returned in a hostent structure.

The first method sets the option on a per application basis, the second method on a per user basis and the third method on a per system basis.

gethostbyname2() function and IPv6 support:

gethostbyname2() allows us to specify the address family.

```
#include  <netdb.h>
struct hostent    *gethostbyname2(cost char *hostname, int family);
  returns : non null pointer if OK,  NULL pointer on error with h_errono set
```

The return value is the same as with gethostbyname, a pointer to a hostent structure and this structure remains the same. The logic function depends on the family argument and on the RES_USE_INET6 option.

Following table summarizes the operation of the gethostbyname6 with regard to the new RES_USEINET6 option.

getbostbyname and gethostbyname2 with resolver RES_USE_INET6 options

The logic works on

- whether the RES_USE_INET6 options is **on or off**
- whether the second argument to gethostbyname2() is **AF_INET** OR **AF_INET6**

- o whether the resolver searches for A records or for AAAA records and
- . whether the returned addresses are of length 4 or 16.

	RES_USE_INET6 option	
	off	On
Gethostbyname (host)	Search for A records. IF found, return IPv4 address (h_length=4) Else error This provides backward compatibility for all existing IPv4 applications	Search for AAAA records, If found, return IPv6 addresses (h_length = 16). Else search for A records. If found, return IPv4 mapped IPv6 addresses (h_length=16). Else error.
Gethostbyname2 (host, AF_INET)	Search for A record. IF found return IPv4 addresses (h_length = 4). Else error	Search for A record. IF found return IPv4 mapped IPv6 addresses (h_length = 16). Else error
Gethostbyname2 (host, AF_INET6)	Search for AAAA record. IF found return IPv6 addresses (h_length = 16). Else error	Search for AAAA record. IF found return IPv6 addresses (h_length = 16). Else error

The operation of gethostbyname 2 is as follows:

- If the family argument is AF_INET, a query is made for the A records. IF unsuccessful, the function returns a null pointer. IF successful, the type and assize of the returned addresses depends on the new RES_USE_INET6 resolver option : if the option is not set, IPv4 address are returned and the h_length members of the hostent structure will be 4. if the option is set, IPv4 mapped IPv6 address are returned and the h_length member of the hostent structure will be 16.
- If the family argument is AF_INET6, a query is made for AAAA records. IF successful, IPv6 addresses are returned and the h_length member of the hostent structure will be 16. otherwise the function returns a null pointer.

The source code that is given below describe the action of gethostbyname and RES_USE_INET^ options.

```
        Strcut  hostent * gethostbyname (const char *name)  {
                Struct hostent

}
```

Small Questions

1. What are various ways to get and set the options that affect a socket?
2. Explain Elementary UDP sockets.
3. Explain UDP server and UDP client.
4. What are the two functions used in Elementary UDP?
5. Difference between main function and dg_echo function.
6. What are the four steps used in client processing loop?
7. Difference between server function dg_echo and client function dg_cli.
8. Define DNS.
9. Define Resource Records.
10. What are the types which affect the RRS?
11. Define Resolvers and Name servers.
12. Explain Gethostbyname function
13. State the role of pointer queries in DNS.
14. What are the three ways to set RES_USE_INET6?
15. Explain gethostbyaddr function.
16. What are unamed functions?
17. Explain gethostname function.
18. Explain getservbyname and getservbyport functions.
19. Explain IPv4 socket option.
20. Explain ICMPv4 socket option.
21. Explain IPv6 socket option.

Big Questions

1. a) Assume both a client and server set the SO_KEEPALIVE socket option and the connectivity is maintained between the peers but them is no exchange of data. When the keepalive timer expires every 2 hours, how many TCP segments are exchanged across the connection? justify your answer with an illustration. (06)

b) Write a program that checks all the socket option of a socket and sets the value for receiver buffer size to 520 bytes. (10)

2. a) Write notes on RES_USE_INET6 resolver option in gethostbyname and gethostbyname2 functions. (08)

b) Discuss any four TCP socket option in detail. (08)

3. a) Discuss about IP socket option and ICMP socket options in detail with Suitable example. (08)

b) Write the similarities between UDP socket, TCP socket and raw socket. (08)

4. a) Explain the purpose and usage of UDP sockets and their different functions. (10)

b) Brief the way in which a TCP client server different from UDP client server. (06)

5. Briefly discuss about DNS with an example. (16)
6. Briefly discuss about TCP Echo server and client. (16)
7. Briefly discuss about UDP Echo server and client. (16)

UNIT IV

ADVANCED SOCKETS

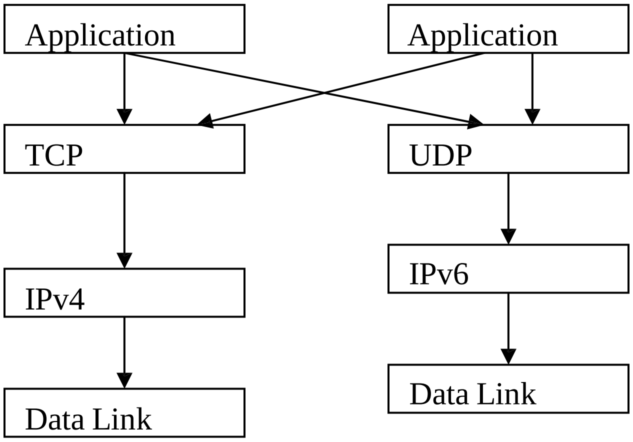
IPV4 and IPV6 interoperability – Threaded servers – Thread creation and termination– TCP echo server using threads – Mutexes – Condition variables – Raw sockets – Raw socket creation – Raw socket output – Raw socket input – Ping program – Trace route program.

ADVANCED SOCKETS

IPv4 and ipV6 INTEROPERABILITY

Till the time, IPv6 is established all over the world, there is a need for one to host dual stacks – that is both IPv4 and IPv6 are running concurrently as shown below:

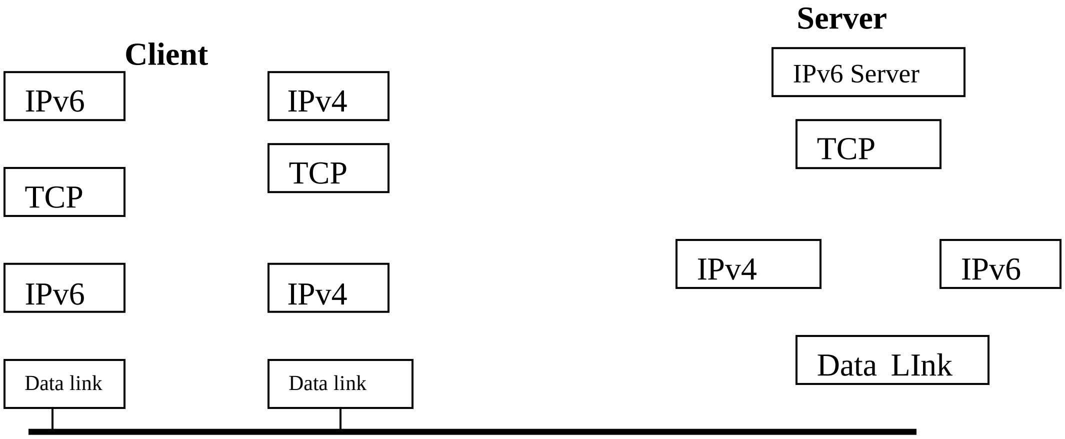
IPv4 , AF_INET **IPv6, AF_INET6**
sockaddr_in () **sockaddr_in6()**



While being so, the client can be on IPv4 or IPv6 format and the server must be able to accept both application.. Here the peculiarities that are involved when the client is on IPv4 and the server is on IPv6 format and when the client is IPv6 and the server is in IPv4 format are discussed.

IPv4 client and IPv6 server:

Following diagram shows the system when the client is either IPv4 or IPv6



Enet hdr	IPv4 Hdr	TCP Hdr	TCP Data

Enet Hdr	IPv6 hdr	TCP Hdr	TCP Datqa

The system considered is on Ethernet for the sake of simplicity. The server is running on dual host and the client can be of either IPv4 or IPv6. The server has created an IPv6 listening TCP socket. It is bound to the IPv6 wild card address and TCP port 8888.

To make connection, the client sends the SYN segment to the server. In this case it appears as Ethernet header (contains a type field 0x0800which identifies the frame as IPv4 frame), followed by IPv4 header (destination IP address in IPv4 format) and the TCP header that contains destination port.

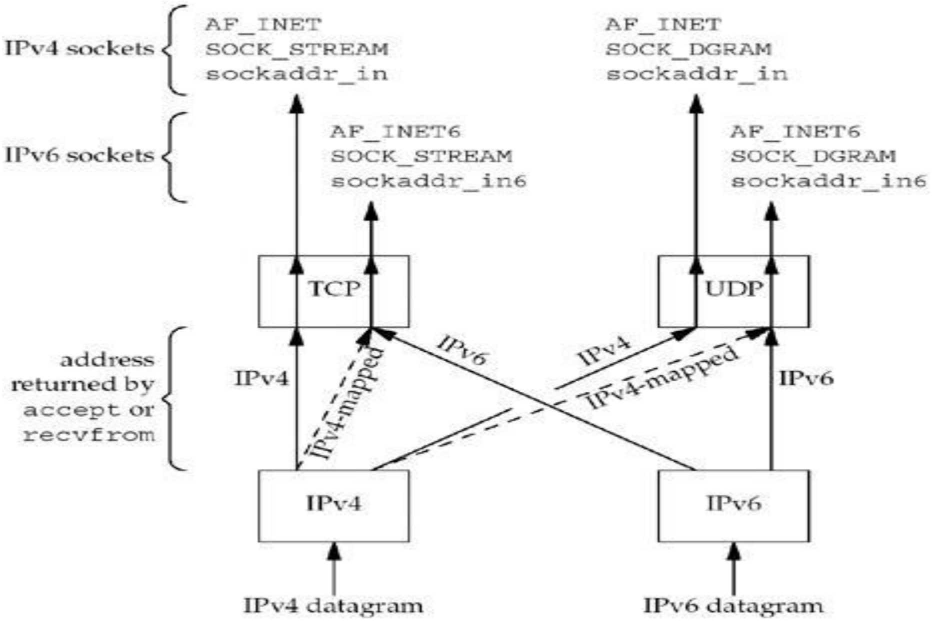
IN case of IPv6 client connection, the clients sends SYN segment to the server. . In this case it appears as Ethernet header (contains a type field 0x86dd which identifies the frame as IPv6 frame), followed by IPv6 header (destination IP address in IPv6 format) and the TCP header that contains destination port.

The receiving datalink looks at the Ethernet type field and passes each frame to the appropriate IP module. The IPv4 module in conjunction with TCP module detects that the destination socket is an IPv6 socket and the source IPv4 address in the IPv4 header is converted into equivalent IPv4 mapped IPv6 address. That mapped address is returned to the IPv6 socket as the client's IPv6 address when accept returns to the server with IPv4 client connection. All remaining datagram for this connections are IPv4 datagram. When accept returns to the server with the IPv6 client connection, the client's IPv6 address does not change from whatever source address appears in the IPv6 header. All remaining datagram for this connections are IPv6 datagram

Steps that allow an IPv4 TCP client to communicate with an IPv6 server:

- The IPv6 server starts, creates an IPv6 listening socket, and we assume it binds the wildcards address to the socket.
- The IPv4 client calls **gethostbyname** and finds an A record for the server. The server host will have both A record and AAAA record, since it supports both protocols but the IPv4 client asks for only an A record.
- The client calls connect and the client host sends an IPv4 SYN to the server.
- The server host receives the IPv4 SYN directed to the IPv6 to the listening socket, sets a flag indicating that this connection is using IPv4 mapped IPv6 addresses and responds with an IPv4 SYN/ACK. When the connection is established, the address returned to the server by accept is the IPv4 mapped IPv6 address.
- All communication between this client and the server tak place using IPv4 datagram
- Both client and server explicitly do not know that they are communicating with different address schemes.

The scenario is similar for an IPv6 UDP server, but the address format can change for each datagram. If anIPv6 server receives a datagram from an IPv4 client, the address returned by **recvfrom** will be the client's IPv4 mapped IPv6 address. The server responds to this clients requests by calling **sendto** with the IPv4 mapped IPv6 address as the destination. This address format tells the kernel to send IPv4 datagram to the client. But the next datagram received to the server could be an IPv6 datagram, and **recvfrom** will return the IPv6 address. IF the server responds, the kernel will generate an IPv6 datagram.



Most dual stack hosts should use the following rules in dealing with listening sockets:

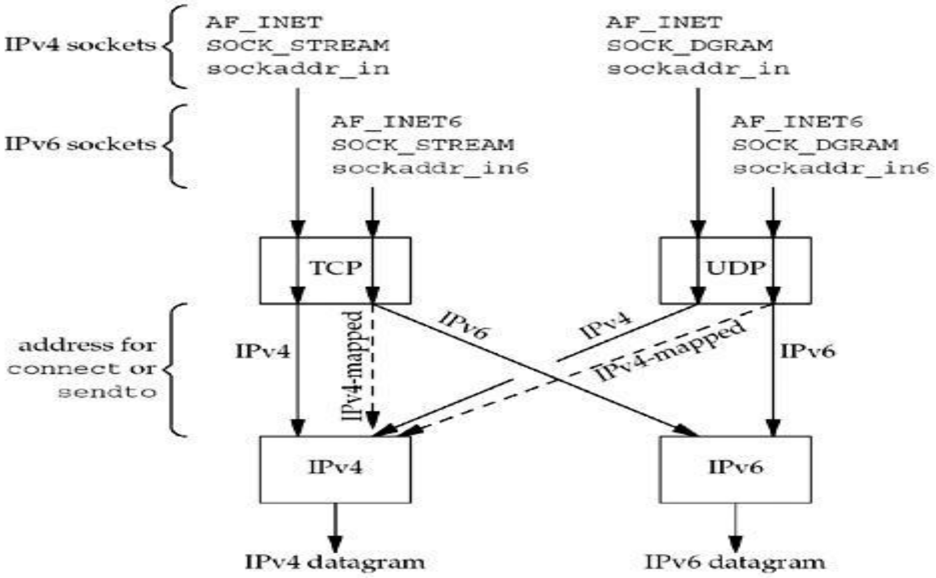
- A listening IPv4 socket can accept incoming connection from only IPv4 client.
- If server has a listening IPv6 socket that has bound the wildcard address, that socket can accept incoming connections from either IPv4 client or IPv6 client. For connection from IPv4 client the server's local address for the connection will be the corresponding IPv4 mapped IPv6 address.
- If a server has a listening IPv6 socket that has bound an IPv6 address other than an IPv4 mapped IPv6 address, then the socket can accept incoming connections from IPv6 clients only.

IPv6 client and IPV4 Server:

This scene is the swapping of the client and server protocols used in the previous case. Consider the IPv6 TCP client running on dual stack host.

- An IPv4 server starts on an IPv4 only host and creates an IPv4 listening socket.
- The IPv6 client starts, calls **gethostbyname()** asking for only IPv6 addresses. (It enables the RES_USE_INET6 option).Since the IPv4 only server host has only A record, an IPv4 mapped IPv6 address is returned to the client.
- The IPv6 client calls **connect** with IPv4 mapped IPv6 address in the IPv6 socket address structure. The kernel detects the mapped address and automatically sends an IPv4 SYN to the server

- The server responds with IPv4 SYN/ACK and the connection is established using IPv4 datagrams.



The scenario can be summarized as given below:

- If an IPv4 TCP client calls connect specifying an IPv4 address, or If an IPv4 UDP client calls sendto specifying an IPV4 address, nothing special is done. Shown as IPv4 arrows in the figure.
- IF an IPv6 TCP client calls connect specifying an IPv6 address, or if an IPv6 UDP client calls sendto specifying an IPv6 address, nothing special is done. These are two arrows labeled IPv6 in the figure.
- IF an IPv6 TCP client specifies an IPv4 mapped IPv6 address to connect or if an IPv6 UDP client specifies an IPv4 mapped.IPv6 address to **sendto**, the kernel detects the mapped address and causes an IPv4 client cannot specified causes an IPv4 datagram to sent, instead of IPv6 datagram. These are the two dashed arrows in the figure.
- An IPv4 client cannot specify an IPv6 address to either connect or sendto because a 16 byte IPv6 address does not fit in the 4 byte **in_addr** structure within the IPv4 client to the IPv6 protocol box in the figure.

IN the previous section, (an IPv4 datagram arriving for an IPv6 server socket), the conversion of the received address to the IPv4 mapped IPv6 address is done by the mernel and returned transparently to the application by accept or recvfrom. IN this secion, (an IPv4 datagram needing to be sent on an IPv6 socket) the conversion of the IPv4 address to the IPv4 mapped IPv6 address is done by the resolver according to the rules. And the mapped address is then passed transparently by the application to the connect or sendto.

THREADS

When a process needs something to be performed by another entity, it forks a child process and lets the child perform processing. (similar to concurrent server program.) The problem associated with this are:

a. All descriptors are copied from parent to child thereby occupying more memory.
b. Inter process communication requires to pass information between the parent and child after each fork. Returning the information form the child to parent takes more work.

Threads being a light weight process help to overcome these drawbacks. However, as they share the same variables – known as global variables - they need to be synchronized to avoid errors. IN addition to these variables, threads also share

- Process instructions
- Data,
- Open files
- Signal handlers and signal dispositions
- Current working directory
- User Groups Ids.

Each thread has its own thread ID, set of registers, program counter and stack pointer, stack, errno, signal mask and priority.

Basic thread functions:

pthread_create function:

int ptheread_create (pthread_t) *tid, const pthread_attr_t *attr, void * (*func) (*void), void *arg);

tid : is the thread ID whose data type is **pthread_t -** unsigned integer. ON successful creation of thread, its ID is returned through the pointer tid.

pthread_atttr_t : Each thread has a number of attributes – priority, initi8al stack size, whether is demon thread or not. If this variable is specified, it overrides the default. To accept the default, attr argument is set to null pointer.

***func :** When the thread is created, a function is specified for it to execute. The thread starts by calling this function and then terminates either explicitly (by calling pthread_exit) or implicitly by letting this function to return. The address of the function is specified as the **func** argument. And this function is called with a single pointer argument, **arg.** If multiple arguments are to be passed, the address of the structure can be passed

The function takes one argument – a generic pointer (void *) and returns a generic pointer (void *). This lets us to pass one pointer to the thread and return one pointer.

The return is normally 0 if OK or nonzero on an error

pthread_join function :
int pthread_join (pthread_t tid, void ** status)

We can wait for a given thread to terminate by calling pthread_join . We must specify the tid of the thread that we wish to wait for. If the status pointer is non null, the return value from the thread is stored in the location pointed to by status.

pthread_self function : Each thread has an ID that identifies it within a given process. The thread ID is returned by pthread_create. This function fetches this value for itself by using this function:
pthread_t pthread_self(void);

pthread_detach function :

A thread is joinable (the default) or detached. When a joinable thread terminates, its thread ID and exit status are retained until thread calls **pthread_join().** But a detached thread for example daemon thread- when it terminates all its resources are released and we cannot wait for it terminate.

When one thread needs to know when another thread terminates, it is best to leave the thread joinable.

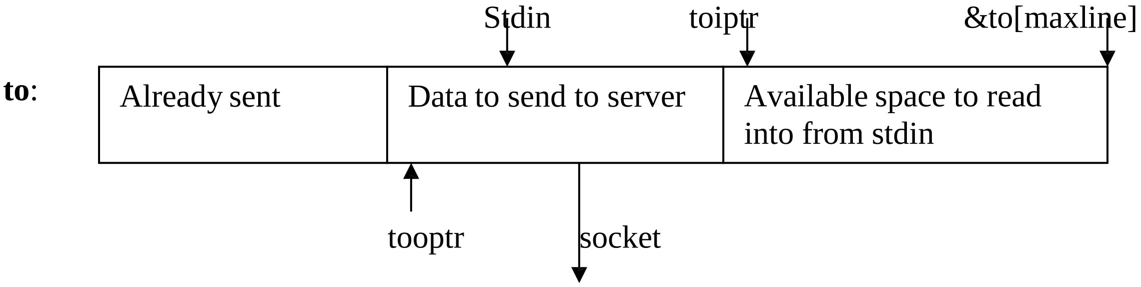
Int pthread_detach (pthread_t tid);

pthread_exit function:

One way for the thread to terminate is to call pthread_exit().

void pthread_exit (void *status);

The str_cli function uses fputs and the fgets to write to and read from the server. While doing so, fgets my be putting the data from **stdin** into the buffer wherein already there is data waiting to be **writen.** This will block the other processes as the function is waiting to write the pending data. If there is data to be read form the server by the function **readen** at the client, it will be blocked. Similarly, if a line of input is available from the socket we can block in the subsequent call to fputs, if the standard output is slower thant the network. IN such situation non blocking methods are used. This requires creating elaborate arrangement of buffer management: where pointer are used to find the data that is already sent, data that are yet to be sent and available space in the read buffer. As shown below:



The program comes to be about 100 lines . This can be reduced by using threads. In this, call to pthread is given in the main function where in the fgets function are invoked. When the thread returns, fputs is invoked.

Mutexes: Mutual Exclusion

Notice the following Figure1.a that when a thread terminates, the main loop decrements both nconn and nlefttoread. We could have placed these two decrements in the function do_get_read, letting each thread decrement these two counters immediately before the thread terminates. But this would be a subtle, yet significant, concurrent programming error.

threads/web01.c

```
    while (nlefttoread > 0) {

        while (nconn < maxnconn && nlefttoconn > 0) {
                /* find a file to read */
            for (i = 0; i < nfiles; i++)
                if (file[i].f_flags == 0)
                    break;
            if (i == nfiles)
                err_quit("nlefttoconn = %d but nothing found", nlefttoconn);

            file[i].f_flags = F_CONNECTING;
            Pthread_create(&tid, NULL, &do_get_read, &file[i]);
            file[i].f_tid = tid;
            nconn++;
            nlefttoconn--;
        }


        if ( (n = thr_join(0, &tid, (void **) &fptr)) != 0)
            errno = n, err_sys("thr_join error");

        nconn--;
        nlefttoread--;
        printf("thread id %d for %s done\n", tid, fptr->f_name);
    }

    exit(0);
}
```

The problem with placing the code in the function that each thread executes is that these two variables are global, not thread-specific. If one thread is in the middle of decrementing a variable, that thread is suspended, and if another thread executes and decrements the same variable, an error can result. For example, assume that the C compiler turns the decrement operator into three instructions: load from memory into a register, decrement the register, and store from the register into memory. Consider the following possible scenario:

- Thread A is running and it loads the value of nconn (3) into a register.

- The system switches threads from A to B. A's registers are saved, and B's registers are restored.

- Thread B executes the three instructions corresponding to the C expression nconn--,

- Sometime later, the system switches threads from B to A. A's registers are restored and A continues where it left off, at the second machine instruction in the three-instructionsequence. The value of the register is decremented from 3 to 2, and the value of 2 is stored in nconn.

The end result is that nconn is 2 when it should be 1. This is wrong.

These types of concurrent programming errors are hard to find for numerous reasons. First, they occur rarely. Nevertheless, it is an error and it will fail (Murphy's Law). Second, the error is hard to duplicate since it depends on the nondeterministic timing of many events. Lastly, on some systems,the hardware instructions might be atomic; that is, there might be a hardware instruction to decrement an integer in memory (instead of the three-instruction sequence we assumed above) and the hardware cannot be interrupted during this instruction. But, this is not guaranteed by all systems, so the code works on one system but not on another.

We call threads programming *concurrent programming*, or *parallel programming*, since multiple threads can be running concurrently (in parallel), accessing the same variables. While the error scenario we just discussed assumes a single-CPU system, the potential for error also exists if threads A and B are running at the same time on different CPUs on a multiprocessor system. With normal Unix programming, we do not encounter these concurrent programming problems because with fork, nothing besides descriptors is shared between the parent and child. We will, however, encounter this same type of problem when we discuss shared memory between processes.

We can easily demonstrate this problem with threads. Figure 2 is a simple program that creates two threads and then has each thread increment a global variable 5,000 times.

We exacerbate the potential for a problem by fetching the current value of counter, printing the new value, and then storing the new value. If we run this program, we have the output shown in Figure 1.b.

Figure 2 Two threads that increment a global variable incorrectly.

threads/example01.c

```
#include    "unpthread.h"

#define NLOOP 5000

int     counter;               /* incremented by threads */

void   *doit(void *);

int
main(int argc, char **argv)
{
    pthread_t tidA, tidB;

    Pthread_create(&tidA, NULL, &doit, NULL);
    Pthread_create(&tidB, NULL, &doit, NULL);

        /* wait for both threads to terminate */
    Pthread_join(tidA, NULL);
    Pthread_join(tidB, NULL);

    exit(0);
}

void *

doit(void *vptr)
{
    int    i, val;

    /*
     * Each thread fetches, prints, and increments the counter NLOOP times.
     * The value of the counter should increase monotonically.
     */

    for (i = 0; i < NLOOP; i++) {
        val = counter;
        printf("%d: %d\n", pthread_self(), val + 1);
        counter = val + 1;
    }
    return (NULL);
}
```

Notice the error the first time the system switches from thread 4 to thread 5: The value 518 i sstored by each thread. This happens numerous times through the 10,000 lines of output.

The nondeterministic nature of this type of problem is also evident if we run the program a few times: Each time, the end result is different from the previous run of the program. Also, if we redirect the output to a disk file, sometimes the error does not occur since the program runs faster, providing fewer opportunities to switch between the threads. The greatest number of errors occurs when we run the program interactively, writing the output to the (slow) terminal, but saving the output in a file using the Unix script program (discussed in detail in Chapter 19 of APUE).

The problem we just discussed, multiple threads updating a shared variable, is the simplest problem. The solution is to protect the shared variable with a *mutex* (which stands for "mutual exclusion") and access the variable only when we hold the mutex. In terms of Pthreads, a mutex is a variable of type pthread_mutex_t. We lock and unlock a mutex using the following two functions:

```
#include   <pthread.h>

int   pthread_mutex_lock(pthread_mutex_t      * mptr);

int   pthread_mutex_unlock(pthread_mutex_t      * mptr);
```

Both return: 0 if OK, positive E*xxx* value on error

If we try to lock a mutex that is already locked by some other thread, we are blocked until the mutex is unlocked.

If a mutex variable is statically allocated, we must initialize it to the constant PTHREAD_MUTEX_INITIALIZER. We will see in next Section that if we allocate a mutex in shared memory, we must initialize it at runtime by calling the pthread_mutex_init function.

> Some systems (e.g., Solaris) define PTHREAD_MUTEX_INITIALIZER to be 0, so omitting this initialization is acceptable, since statically allocated variables are automatically initialized to 0. But there is no guarantee that this is acceptable and other systems (e.g., Digital Unix) define the initializer to be nonzero.

Figure3 is a corrected version of Figure 2 that uses a single mutex to lock the counter between the two threads.

Figure 3 Corrected version of Figure 2 using a mutex to protect the shared variable.

threads/example02.c

```
#include   "unpthread.h"

#define NLOOP 5000
```

```
int     counter;                /* incremented by threads */
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void   *doit(void *);
int
main(int argc, char **argv)
{
    pthread_t tidA, tidB;

    Pthread_create(&tidA, NULL, &doit, NULL);
    Pthread_create(&tidB, NULL, &doit, NULL);

        /* wait for both threads to terminate */
    Pthread_join(tidA, NULL);
    Pthread_join(tidB, NULL);

    exit(0);
}
void *
doit(void *vptr)
{
    int     i, val;

    /*
     * Each thread fetches, prints, and increments the counter NLOOP times.
     * The value of the counter should increase monotonically.
     */

    for (i = 0; i < NLOOP; i++) {
        Pthread_mutex_lock(&counter_mutex);

        val = counter;
        printf("%d: %d\n", pthread_self(), val + 1);
        counter = val + 1;


        Pthread_mutex_unlock(&counter_mutex);
    }

    return (NULL);
}
```

We declare a mutex named counter_mutex and this mutex must be locked by the thread before the thread manipulates the counter variable. When we run this program, the output is always correct: The value is incremented monotonically and the final value printed is always 10,000.

How much overhead is involved with mutex locking? The programs in Figures 2 and 3 were changed to loop 50,000 times and were timed while the output was directed to /dev/null.The difference in CPU time from the incorrect version with no mutex to the correct version that used a mutex was 10%. This tells us that mutex locking is not a large overhead

Condition Variables

A mutex is fine to prevent simultaneous access to a shared variable, but we need something else to let us go to sleep waiting for some condition to occur. Let's demonstrate this with an exampleBut, we cannot call the Pthread function until we know that a thread has terminated. We first declare a global variable that counts the number of terminated threads and protect it with a mutex.

```
     int          ndone;      /* number of terminated threads */
     pthread_mutex_t ndone_mutex = PTHREAD_MUTEX_INITIALIZER;
```

We then require that each thread increment this counter when it terminates, being careful to use the associated mutex.

```
  void *
  do_get_read (void *vptr)
  {
     ...

Pthread_mutex_lock(&ndone_mutex);
     ndone++;
    Pthread_mutex_unlock(&ndone_mutex);

     return(fptr);     /* terminate thread */
  }
```

A *condition variable*, in conjunction with a mutex, provides this facility. The mutex provides mutual exclusion and the condition variable provides a signaling mechanism.

In terms of Pthreads, a condition variable is a variable of type pthread_cond_t. They are used with the following two functions:

```
 #include    <pthread.h>

 int    pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);

 int     pthread_cond_signal(pthread_cond_t *cptr);
```

Both return: 0 if OK, positive E*xxx* value on error

The term "signal" in the second function's name does not refer to a Unix SIG*xxx* signal.

Why is a mutex always associated with a condition variable? The "condition" is normally the value of some variable that is shared between the threads. The mutex is required to allow this variable tobe set and tested by the different threads. For example, if we did not have the mutex in the example code just shown, the main loop would test it as follows:

```
     /* Wait for thread to terminate */
  while (ndone == 0)
     Pthread_cond_wait(&ndone_cond, &ndone_mutex);
```

The thread would have to unlock and lock the mutex and the code would look like the following:

```
     /* Wait for thread to terminate */
  Pthread_mutex_lock(&ndone_mutex);
  while (ndone == 0) {
     Pthread_mutex_unlock(&ndone_mutex);
     Pthread_cond_wait(&ndone_cond, &ndone_mutex);
     Pthread_mutex_lock(&ndone_mutex);
  }
```

```
#include    <pthread.h>

int   pthread_cond_broadcast    (pthread_cond_t     * cptr);

int   pthread_cond_timedwait    (pthread_cond_t     * cptr, pthread_mutex_t *mptr,
const
struct timespec *abstime);
```

Both return: 0 if OK, positive E*xxx* value on error

pthread_cond_timedwait lets a thread place a limit on how long it will block. *abstime* is a timespec structure (as we defined with the pselect function, that specifies the system time when the function must return, even if the condition variable has not been signalled yet. If this timeout occurs, ETIME is returned.

This time value is an *absolute time*; it is not a *time delta*. That is, *abstime* is the system time—the number of seconds and nanoseconds past January 1, 1970, UTC—when the function should return. This differs from both select and pselect, which specify the number of seconds and microseconds (nanoseconds for pselect) until some time in the future when the function should return. The normal procedure is to call gettimeofday to obtain the current time (as a timeval structure!), and copy this into a timespec structure, adding in the desired time limit. For example,

```
struct timeval tv;
    struct timespec ts;
    if (gettimeofday(&tv, NULL) < 0)
        err_sys("gettimeofday error");
    ts.tv_sec = tv.tv_sec + 5;     /* 5 seconds in future */
    ts.tv_nsec = tv.tv_usec * 1000; /* microsec to nanosec */


pthread_cond_timedwait( ..., &ts);
```

The advantage in using an absolute time instead of a delta time is if the function prematurely returns (perhaps because of a caught signal), the function can be called again, without having to change the contents of the timespec structure. The disadvantage, however, is having to call gettimeofday before the function can be called the first time.

The POSIX specification defines a clock_gettime function that returns the current time as a timespec structure.

RAW SOCKETS

INTRODUCTION:

Raw sockets, are those that bypass the TCP and IP layers and pass the ICMPv4, (Internet Control Message Protocol), IGMPv4 ()Internet Group Management Protocol – used with multicasting) and ICMPv6 packets directly to the link layers.

This allows the application to build ICMP and IGMP entirely as user processes instead of putting more code into the kernel. Examples are route discovery daemon which processes router advertisement and router solicitation are built this way.

With raw sockets a process can read and write IPv4 datagram with IPv4 protocol filed (an 8 bit filed in IPv4 packet) that is not processed by the kernel. Most kernels process datagrams containing values of 1 (ICMP), 2 (IGMP), 6 (TCP), and 17 (UDP). But values like 89 (OSPF) routing protocol does not use TCP or UDP but uses IP directly by setting the protocol field to 89.

With raw sockets, a process can build its own IPv4 header using the IP_HDRINCL socket option
We learn the raw socket creation, input and the output and developing few programs that work with IPv4 and IPv6.

RAW SOCKET CREATION

1. To create raw sockets, the second argument in socket function SOCK_RAW. And the third argument is nonzero (normally) as shown below:

```
Int sockfd;
Sockfd = socket (AF_INET, SOCK_RAW, protocol);
```

In this the protocol is th one of the constants defined by IPPROTO_XXX which is done by including <netinet/in.h> header. For example IPPROO_ICMP. Only super user can create raw socket.

2. The IP-HDRINCL socket option can be set to:

```
const int ON =1;
if (setsocketopt(sockfd, IPPROTO_IP, IP_HDRINCL, &ON, soze0f(ON)) <0)
        error
```

3. Bind may not be called on raw sockets. If called, it sets the local IP address and not the port number as there is no concept of port number with raw sockets. With regard to output, calling bind sets the IP address that will be used for datagrams sent on the raw socket (only if IP_HDRINCL socket option is not set). If bind is not called, the kernel sets the source IP address of the outgoing interface.

4. connect can be call on the raw socket but this is also rare. This function sets only the foreign address and again there is no concept of port number. With regard to output, calling connect lets us call write or send instead of sendto, since the destination IP address is already specified.

Raw Socket Output:

The output of raw socket is governed by the following rules:

- Normal output is performed by calling **sendto** or **sendmsg** and specifying the destination IP address. IN case the socket has been connected, **write** and **send** functions can be used.
- If the **IP_HDRINCL** option is **not set,** the IP header will be built by the kernal and it will be prepend it to the data.
- If **IP_HDRINCL** is **set,** the header format will remain the same and the process builds the entire IP header except the IPv4 identification field which is set to 0 by the kernel
- The kernel fragments the raw packets that exceed the outgoing interface.

IPv6 Differences:

- All fields in the protocol headers sent or received on a raw IPv6 sockets are in network byte order.
- There ae no option fields in IPv6 format. Almost all fields in an IPv6 header and all extension headers (Optional header that follow have their own length field. There is a separate fragmentation header.) are available to the application through socket options.
- Checksum are handled differently.

IPv6 CHECKSUM Socket option

- In case of ICMPv4, the checksum is calculated by the application. Whereas in the application it is done by the kernel.

Raw Socket Input:

The question to be answered in this is which received IP datagrams does the kernel pass to raw sockets.

- Received TCP and UDP packets are never passed to a raw socket.
- Most ICMP packets are passed to a raw socket after the kernel has finished processing the ICMP message. BSD derived implementations pass all received ICMP raw sockets other than echo requests, timestamp request and address mask request. These three ICMP messages are processed entirely by the kernel.
- All IGMP packets are passed to a raw sockets, after the kernel has finished processing the IGMP message.
- All IP datagram with a protocol field that kernel does not understand are passed to a raw socket. The only kernel processing done on these packets is the minimal verification of some IP header field: IP version, IPv4 Header checksum, header length and the destination IP address.
- If the datagram arrives in fragments, nothing is passed to a raw sockets until all fragments have arrived and have been reassembled.

When kernel has to pass IP datagram, it should satisfy all the three tests given below:

- If a nonzero protocol is specified when the raw socket is created (third argument to socket), then the received datagram's protocol field must match this value or the datagram is not delivered.
- IF a local IP address is bound to the raw socket by bind, then the destination IP address of the received datagram must match this bound address or the datagram is not delivered.
- IF foreign IP address was specified for the raw socket by connect, then the source IP address of the received datagram must match this connected address or datagram is not delivered.

If a raw socket is created with protocol of 0, and neither bind or connect is called, then that socket receives a copy of every raw datagram that kernel passes to raw sockets.

When a received datagram is passed to a raw IPv4 socket, the entire datagram, including the IP header, is passed to the process.

ICMPv6 Type Filtering:

A raw ICMPv6 is a superset of ICMPv4, ARP and IGMP and hence the socket can receive many more packets compared to ICMPv4 socket. To reduce the number of packets passed form kernet ot the application , an application specific filter is provided A filter is declared with a data type of **struct icmp_filter** which is defined by including <netinet/icmp6.h> header. The current filter for a raw socket is set and fetched using **setsockopt** and **getsockopt** with a level of **IPPROTO_ICMPv6** and optname **ICMP_FILTER**.

Ping Program:

In this ICMP echo request is sent to some IP address and that the node responds with an ICMP echo reply. These two ICMP messages are supported under IPv4 and IPv6. Following figure shows the format of the ICMP messages.

<table>
<tr><td>Type</td><td>Code</td><td>Checksum</td></tr>
<tr><td colspan="2">Identifier</td><td>Sequence number</td></tr>
<tr><td colspan="3">Optional Data</td></tr>
</table>

Type	Code	Description	Handled by error no
0	0	Echo reply	User process (ping)
3		Destination unreachable	
	0	Network unreachable	EHOSTUNREACH
	1	Host unreachable	EHOSTUNREACH
`	2	Protocol un reachable	ECONNREFUSED.
	4	port unreachable	ECONNREFUSED

5		REDIRECT	
	0	Redirect for network	Kernel updates routing table
	1		

Checksum is the standard Internet Checksum,
Identifier is set to the process ID of the ping process and the sequence number is incremented by one for each packet that we send. An 8 buit timestamp is stored when a packet is sent as optional data. The rules of ICMP requires that the identifier, sequence number and any optional data be returned in the echo reply Storing the timestamp in the packet lets us calculate the RTT when the reply is received.

Trace route Program:

Traceroute lets us determine the path that IP datagrams follow from our host to some other destination. Its operation is simple and Chapter 8 of TCPv1 covers it in detail with numerous examples of its usage. traceroute uses the IPv4 TTL field or the IPv6 hop limit field and two ICMP messages. It starts by sending a UDP datagram to the destination with a TTL (or hop limit) of 1.This datagram causes the first-hop router to return an ICMP "time exceeded in transit" error. The TTL is then increased by one and another UDP datagram is sent, which locates the next router in the path. When the UDP datagram reaches the final destination, the goal is to have that host return an ICMP "port unreachable" error. This is done by sending the UDP datagram to a random port that is (hopefully) not in use on that host.

The figure shows our trace.h header, which all our program files include.

1–11 We include the standard IPv4 headers that define the IPv4, ICMPv4, and UDP structures and constants. The rec structure defines the data portion of the UDP datagram that we send, but we will see that we never need to examine this data. It is sent mainly for debugging purposes.

Define proto structure

32–43 As with our ping program in the previous section, we handle the protocol differences between IPv4 and IPv6 by defining a proto structure that contains function pointers, pointers to socket address structures, and other constants that differ between the two IP versions. The global pr will be set to point to one of these structures that is initialized for either IPv4 or IPv6, after the destination address is processed by the main function (since the destination address is what specifies whether we use IPv4 or IPv6).

Include IPv6 headers

44–47 We include the headers that define the IPv6 and ICMPv6 structures and constants.

Figure trace.h header.

traceroute/trace.h

```
#include     "unp.h"
#include    <netinet/in_systm.h>
#include    <netinet/ip.h>
#include    <netinet/ip_icmp.h>
#include    <netinet/udp.h>

#define BUFSIZE      1500

struct rec {                       /* of outgoing UDP data */
    u_short rec_seq;               /* sequence number */
    u_short rec_ttl;               /* TTL packet left with */
    struct timeval rec_tv;         /* time packet left */
};

            /* globals */
char     recvbuf [BUFSIZE];
char     sendbuf [BUFSIZE];

int     datalen;                   /* # bytes of data following ICMP header */
char   *host;
u_short sport, dport;
int     nsent;                     /* add 1 for each sendto () */
pid_t   pid;                       /* our PID */
int     probe, nprobes;
int     sendfd, recvfd;            /* send on UDP sock, read on raw ICMP sock */
int     ttl, max_ttl;
int     verbose;

            /* function prototypes */
const char *icmpcode_v4 (int);
const char *icmpcode_v6 (int);
int     recv_v4 (int, struct timeval *);
int     recv_v6 (int, struct timeval *);
void    sig_alrm (int);
void    traceloop (void);
void    tv_sub (struct timeval *, struct timeval *);

struct proto {
    const char *(*icmpcode) (int);
    int     (*recv) (int, struct timeval *);
    struct sockaddr *sasend;    /* sockaddr{} for send, from getaddrinfo */
    struct sockaddr *sarecv;    /* sockaddr{} for receiving */
    struct sockaddr *salast;    /* last sockaddr{} for receiving */
    struct sockaddr *sabind;    /* sockaddr{} for binding source port */
    socklen_t salen;            /* length of sockaddr{}s */
    int     icmpproto;          /* IPPROTO_xxx value for ICMP */
41    int    ttllevel;          /* setsockopt () level to set TTL */
42    int    ttloptname;          /* setsockopt () name to set TTL */
43 } *pr;

44 #ifdef IPV6

45 #include     <netinet/ip6.h>
46 #include     <netinet/icmp6.h>

47 #endif
```

The main function is shown in Figure 28.18 (p. 759). It processes the command-line arguments, initializes the pr pointer for either IPv4 or IPv6, and calls our traceloop function.

Define proto structures

2–9 We define the two proto structures, one for IPv4 and one for IPv6, although the pointers to the socket address structures are not allocated until the end of this function.

Set defaults

10–13 The maximum TTL or hop limit that the program uses defaults to 30, although we provide the -m command-line option to let the user change this. For each TTL, we send three probe packets,but this could be changed with another command-line option. The initial destination port is 32768+666, which will be incremented by one each time we send a UDP datagram. We hope that these ports are not in use on the destination host when the datagrams finally reach the destination,but there is no guarantee.

Process command-line arguments

19–37 The -v command-line option causes most received ICMP messages to be printed.

Process hostname or IP address argument and finish initialization

38–58 The destination hostname or IP address is processed by our host_serv function, returning a pointer to an addrinfo structure. Depending on the type of returned address, IPv4 or IPv6, we finish initializing the proto structure, store the pointer in the pr global, and allocate additional socket address structures of the correct size.

59 The function traceloop , shown in Figure 28.19 , sends the datagrams and reads the returned ICMP messages. This is the main loop of the program.

Short questions

1. Explain IPv4 and IPv6 server.
2. What are Address Testing macros?
3. Explain the implementations of threads.
4. What are the advantages and disadvantages of threads?
5. What are the basic function of thread creation and termination?
6. Define thread.
7. List out the unique values maintained by a thread.
8. What are the common thread interfaces?
9. Explain thread function.
10. Define multithreading.
11. Mention the purpose of ping program.
12. Explain trace route program.
13. Define mutexes.
14. Explain basic thread functions.
15. Explain raw sockets.
16. Define proto structure.
17. Differentiate ping and trace route program.

Big Questions

1. a) Compare Fork and Thread. (04)

b) Compare Wait and Waitpid. (04)

c) Write a ‘C’ program that can generate an ICMPv4 echo request packet and Process the received ICMPv4 echo reply. (08)

2. a) Write notes on raw socket creation (04)

b) Write notes onraw socket output (06)

c) Write notes on raw socket input (06)

3. a) Explain how a TCP echo server using thread created and also give their advantages. (10)

b) Write short notes on mutexes and condition variables. (06)

4. a) Compare IPv4 and IPv6. (08)

b) Explain about thread creation and thread termination with suitable example. (08)

5. Explain the trace route program with sample code and example . (16)

6. Explain in detail IPv4 and IPv6 interoperability. (16)

UNIT V

SIMPLE NETWORK MANAGEMENT

SNMP network management concepts – SNMP management information – Standard MIB’s – SNMP V1 protocol and practical issues – Introduction to RMON, SNMP V2 and SNMP V3.

S N M P – INTRODUCTION

Network Management Requirements:

- Fault Management:
    - o Determine the location of fault
    - o Isolate the fault and continue working
    - o Minimize the impact of the fault
    - o Repair or replace the failed component
- Account Management :
    - o Established charges for use of services and charge accordingly.
- Configuration and Name Management :
    - o Initializing a network and shutting down gracefully
    - o Maintaining, adding and updating the relationship among the components and status of the components.
- Performance Management;
    - o Facility needed to evaluate the behaviour of managed objects and the effectiveness of the communication activities.
    - o Control effectiveness of communication activities at various levels.
- Security Management:
    - o Address the security aspects essential for network management and to protect managed objects
    - o Protection of target network security, access control of facilities, generating, storing, distributing encryption keys, passwords, authorization control information etc.

Network Management System:

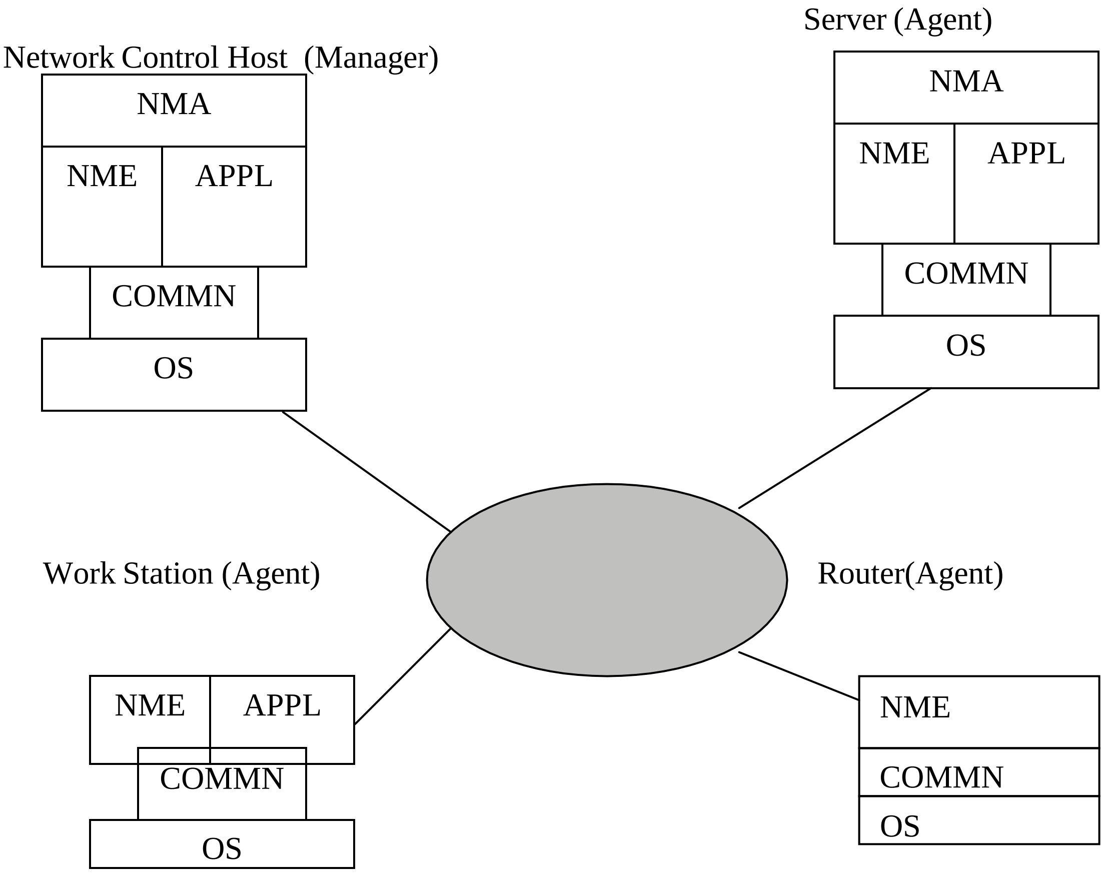
A Network Management System is a collection of tools (hardware and software) for network monitoring and control. It is the incremental hardware and software additions implemented among the existing network components. The software is used in accomplishing the network management tasks residing the host computers and communication processors (bridges, routers, front end processors, cluster controller terminals etc)

Network Management Configuration:

The architecture of the network management system is shown in the next page.
Each node contains a collection of software devoted to the network management tasks referred as NME – Network Management Entity. Each NME performs the following tasks:

- Collect statistics on communication and network related activities.
- Store statistics locally
- Respond to commands from the **network control center**, including commands to :
    - o Transmit collected statistics to the network control center
    - o Change parameter
    - o Provide status information (parameter values. Active links etc)
    - o Generate artificial traffic to perform a test.
- Send message to the network control center when local conditions undergo significant change.

At least one node in the network is designated as the network control host, or manager. IN addition to NME, the network control host includes a collection of software called network management application (NMA). NMA includes operators interface to allow an authorized user to manage the network. NMA responds to user commands by displaying information and / or by issuing commands to NME through the network. This communication is carried out through an application level network management protocol that employs the communication architecture in the same fashion as any other distributed application.



NMA : Network Management Application; Appl: Application;
NME Network Management Entity ; Comm : Communication software ; OS Operating System.

Important points to remember are:

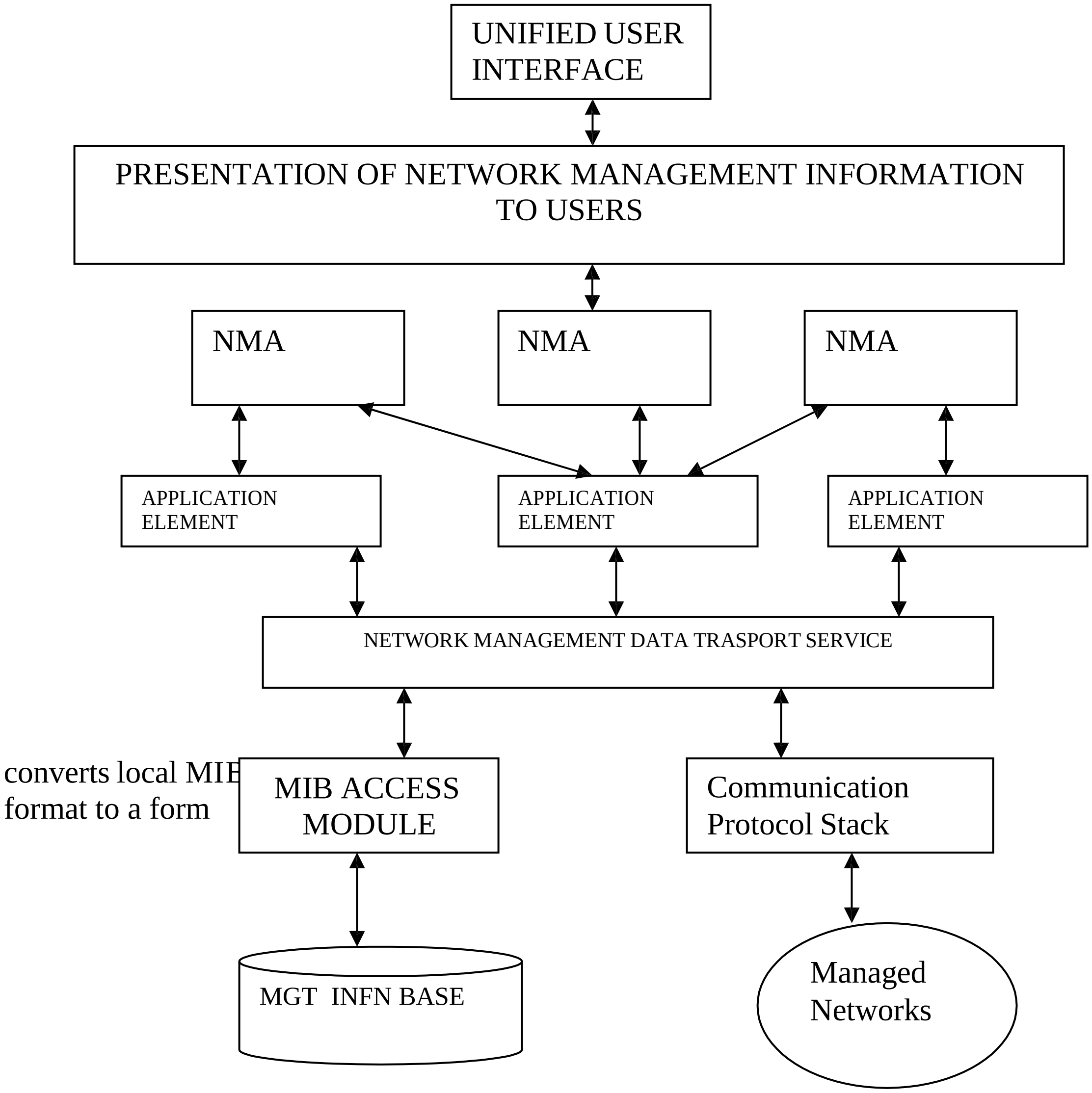
- Network Management software relies on the host operating system and on the communication architecture.
- Network Control Host communicates with and controls the NMEs in other systems.
- For maintaining high availability on the network management functions, two or more network control hosts are used. One may be collecting statistics or even idle where as other may be used for control. IF the primary network fails, the back ups system can be used.

Network Management Software Architecture can be divided into three broad categories;

- User Presentation software:

  Interaction between user and NM Software takes place across user interface. Such interface allows the manager to monitor and control the network
  Such interface has to unified user interface at any node regardless of vendor.

- Network Management Software:
  This normally consists of three layers. Top layer consists of collection of network management application that provide the services of interest to users – like fault management, configuration management, performance management and security
- The small number of network management applications is supported by a larger number of application elements. There are the modules that implement more [primitive and more general purpose network management functions such a as generating alarms summarizing data etc. The application elements implements basic tools that are of use tone or more of the network management applications.
- Lowest level of management specific software is a network management data transport service. This module consists of a network management protocol used exchange management information among managers and agents and service interface to the application elements


UNIFIED USER INTERFACE
PRESENTATION OF NETWORK MANAGEMENT INFORMATION TO USERS
NMA
NMA
NMA
APPLICATION ELEMENT
APPLICATION ELEMENT
APPLICATION ELEMENT
NETWORK MANAGEMENT DATA TRASPORT SERVICE
converts local MIE format to a form
MIB ACCESS MODULE
Communication Protocol Stack
MGT INFN BASE
Managed Networks

ARCHITECTURAL MODEL OF NETWORK MANAGEMENT SYSTEM

S N M P - NETWORK MANAGEMENT CONCEPTS

Network Management Architecture:

The model of network management that is used for TCP / IP network management includes the following key elements.

- Management Station
- Management Agent
- Management Information Base
- Network Management protocol

Management Station: It is a stand alone device but it may be a capability implemented on a shared platform. IN either case Management Station serves as the interface for the hu8man network manager into the network management system. At a minimum the station will consists of the following:

- A set of management application for data analysis, fault recovery and so on.
- An interface by which the network manager my monitor and control the network.
- The capability of translating the network manager's requirenment into the actual monitoring and control of remote elements in the network
- A database of information extracted from the MIB of all the managed entities in the network.

Management Agent: This is other active element. Platforms like bridges, hosts, routers and hubs may be equipped with SNMP agents so that they may be managed from the management stations. The management agent responds to request for information and actions from the management stations and may asynchronously provide the management stations with important but unsolicited information.

Management Information Base :

Resources in the network may be managed by representing those resources as objects. Each objects is essentially a data variable that represents one aspects of management agent. The collection of object is referred to as a **Management Information Base (MIB)** The MIB functions as a collection of access points at the agent for the management station. These objects are standardized across system of a particular class (common set of objects is used for the management of various bridges). A management station performs the monitoring functions by retrieving the value of MIB objects. An Management station can cause an action to take place at an agent or can change the configuration setting at an agent by modifying the value of specific variables.

Network Management Protocol :

The management stations and agents are linked by Network Management Protocol. The protocol used for the management of TCP / IP networks is SNMP, which includes the following key capabilities:

get : enables the management station to retrieve the value of the objects at the agent.

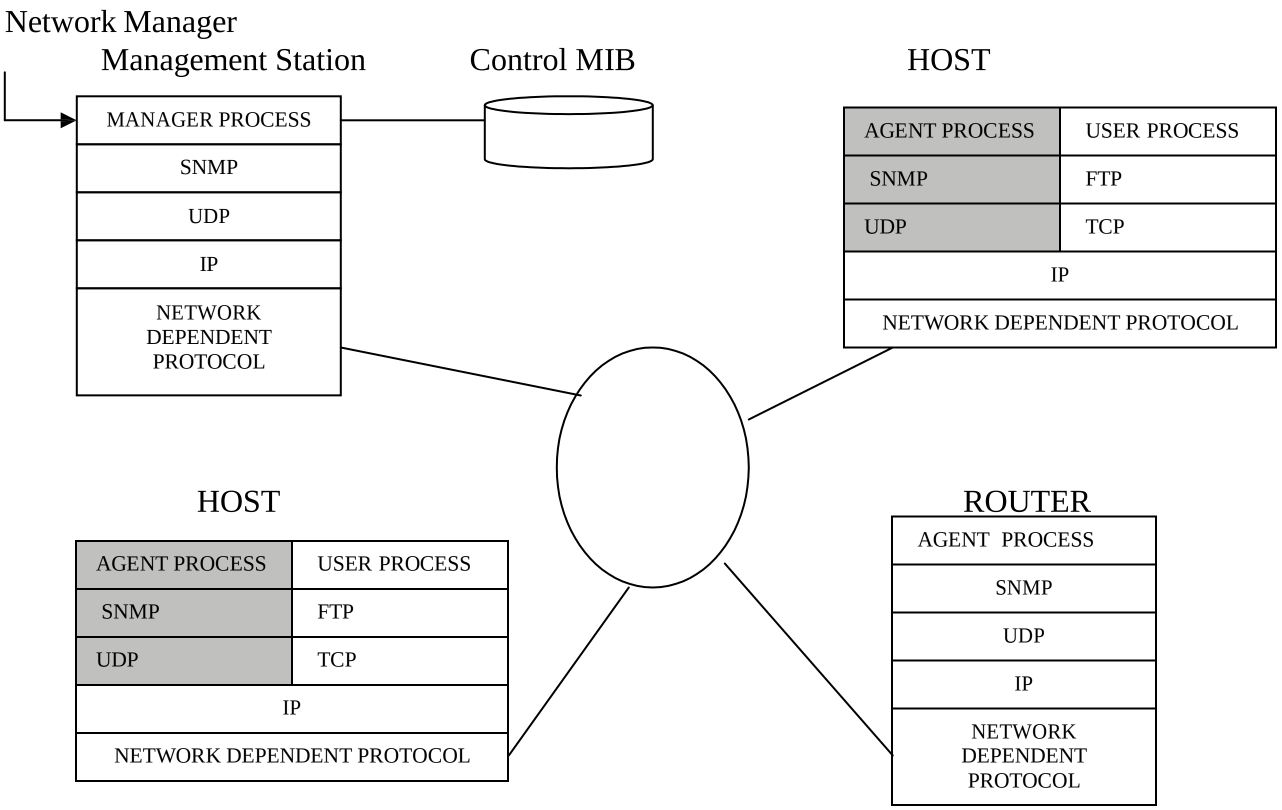
set : Enables the management station to set the4 value of objects at the agent.

trap: enables an agent to notify the management station of significant events.

It is prudent to have two system to perform Management station functions to provide redundancy.

Network Management Protocol Architecture:

SNMP was designed to be an application level protocol that is part of TCP / IP protocol suite. It operates over UDP. Following figure shows the typical configuration of SNMP Protocol:



CONFIGURATION OF SNMP

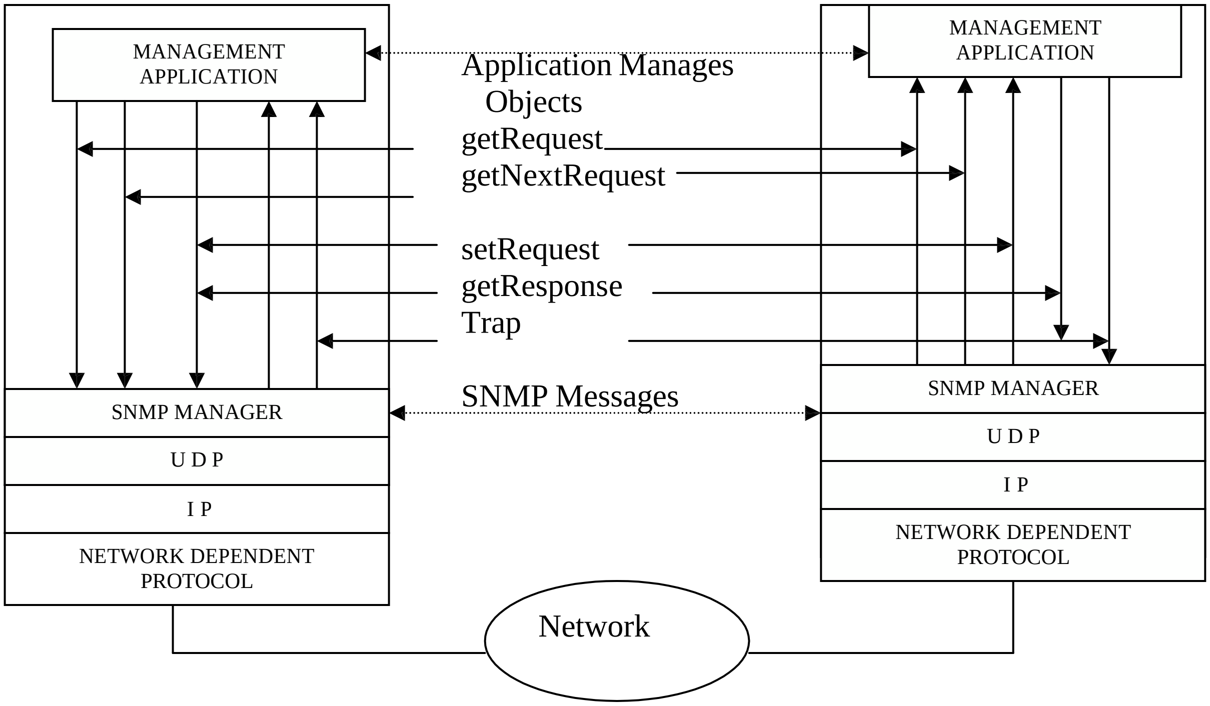
For a stand alone management station, a manager process controls access to a central MIB and provides interface to the network manager. The manager process achieves network management by using SNMP which is implemented on top of UDP, IP and the relevant network dependent protocol (Ethernet, FDDI, and X.25).

Each agent implements SNMP, UDP and IP. For an agent device that supports other application such as FTP, both TCP and UDP are required. Shaded area represents the support provided to network management functions.

Following figure provides further elaborate look at the protocol context on SNMP. From management station three types of SNMP messages are issued on behalf of a management application : getRequest, getNextRequest, setRequest. The first two are variation of get function. All the three messages are acknowledged by the agent in the form of getResponse message. In addition a agent may issue a tap message in response to an event that affects the MIB and the underlying managed resources.

As SNMP relies on UDP, SNMP by itself is connectionless protocol. No ongoing connections are maintained between a management station and its agents. Instead, each exchange is a separate transaction between a management station and an agent.

Role of SNMP



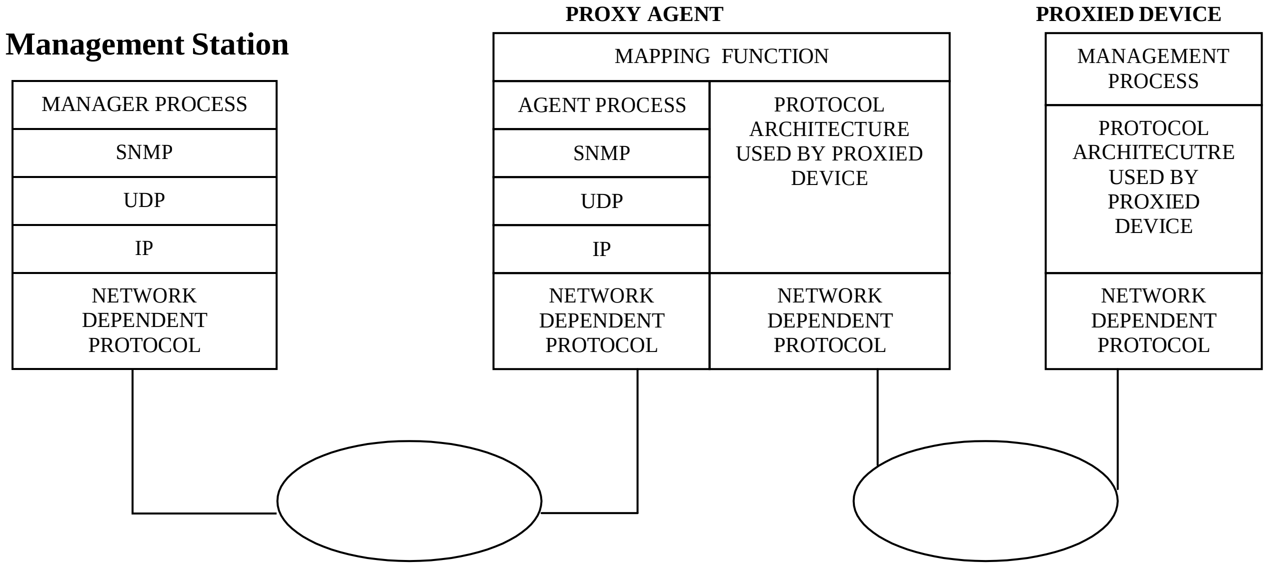
Trap Directed Polling :

If a management station is responsible for a large number of agents and if each agent maintains a large number of objects, then it becomes difficult for the management station to regularly poll agents for all of their readable data. Instead SNMP and associated MIB are designed to encourage the manager to use a technique referred to as ***trap directed polling.***

In this at initialization time, and at frequent intervals (Once a day) a management station can poll all of the agents it knows for some key information such as interface characteristics and for some baseline performance statistics (average number of packets sent) and received over a given period of time. Once this baseline is established, the management station refrains from polling. However any unusual incident is notified by the each agent for example agent crashing and rebooting, link failure, overload condition etc. These events are communicated in SNMP messages known as ***traps***.

Proxies: The use of SNMP requires that all agents, as well as management stations must support a common protocol suite, such as UDP and IP. This limits direct management to such devices such as some bridges , modems that are not part of TCP / IP protocol suit. Similarly a number of PCs, Programmable controllers that do not implement TCP/ IP for which it is not desirable to add SNMP, agent logic and MIB maintenance etc.

To accommodate devices that do not implement SNMP, the concept of proxy was developed. In this scheme an SNMP agent acts as a proxy for one or more other devices. This is shown in the following figure.


PROXY AGENT
PROXIED DEVICE
Management Station
MANAGER PROCESS
SNMP
UDP
IP
NETWORK DEPENDENT PROTOCOL
MAPPING FUNCTION
AGENT PROCESS
SNMP
UDP
IP
NETWORK DEPENDENT PROTOCOL
PROTOCOL ARCHITECTURE USED BY PROXIED DEVICE
NETWORK DEPENDENT PROTOCOL
MANAGEMENT PROCESS
PROTOCOL ARCHITECUTRE USED BY PROXIED DEVICE
NETWORK DEPENDENT PROTOCOL

Above figure shows that management stations sends queries concerning a device to its proxy agent. The proxy agent converts each query into the management protocol that the device is using. When the agent receives a reply to a query, it passes that reply back to the management station. Similarly, if an event notification of some sort from which the device is transmitted to the proxy, the proxy sends that on to the management station in the form of trap message.

<u>S N M P MANAGEMENT INFORMATION</u>

- The foundation of Network Management System is creation of database that contains information about the elements to be managed.
- An MIB is an structured collection of information about objects that are part of the network(servers, workstations, routers, bridges etc.)
- Each system in a network maintains a MIB that reflects the status of the managed resources at that system. NME (Network Management Entity) can monitor the resources at that system by reading the values of objects in the MIB and may control the resources at that system by modifying those values.

To serve these needs, the MIB must meet certain objectives:

- Data base where manageable objects are defined.
- The objects or objects used to represent a particular resources must be the same at each system. (Keeping data regarding active, passive or total open connections with any two of these data which must be uniform)
- A common scheme for representation must be used to support interoperability.

The first points details the objects types and the second point details the type of structure for uniformity.

Structure Management Information

Structured Management Information explains "How to write and define MIB". The SMI defines the general framework within which a MIB can be defined and constructed . The SMI identifies the data type that can be used in the MIB and specifies how within MIB are represented and named.

For the sake of simplicity, SMI must do the following:

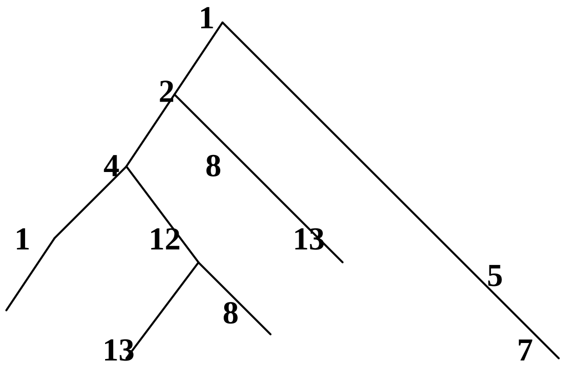
- Provide a standardized technique for defining the structure of a particular MIB
- Provide standardized technique for defining individual objects, including the systems and the value of each object.
- Provide a standardized technique for encoding object values.

MIB Structure

- The Internet Naming Hierarchy
- Objects Types
- Simple/Tabular Objects
- Instances Identification

The Internet Naming Hierarchy

All managed objects in the SNMP environment are arranged in a hierarchical or tree structure. The leaf objects of the tree are the actual managed objects, each of which represents some resources, activity or related information that is to be managed. The tree structure itself defines a grouping of objects into logically related sets. Each object is named by the sequence of the identifiers from the root to the object



The object identifier is : 1.2.4.12.3

Object Types:

A restricted subset of ASN.1 is used to describe objects types

Two ASN.1 classes are used :

Universal Types Application Independent

Application-Wide Types :

- Defined in the context of a particular application
- Each application, including SNMP, is responsible for defining its own application-wide data types

Following data types are permitted :

Integer (ex. : 5, - 10)
Octet string (ex. : protocol)
Null (object with no value associated)
Object identifier (ex. : 1.3.6.1.2)
And the constructor type (used to build tables) : Sequence, Sequence- of

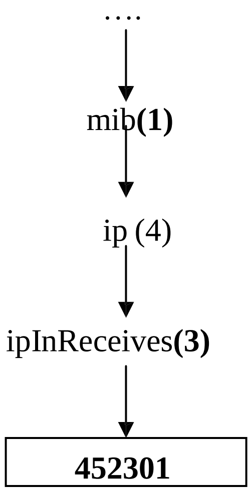
RFC 1155 defines the following application-wide data types :

Network address , IP address : Internet 32- bit address
Counter : Non- negative integer (can be incremented but not decremented)
Gauge : Non- negative integer that may increase or decrease
Timeticks : Non- negative integer counting the time in hundredths of second
Opaque : Arbitrary data transmitted in the form of an octet string

Simpler And Tabular Objects

Simple Objects : Object with a unique instance within the agent.
Its type is one of the following : integer, octet string, null, object identifier, network address, IP address, counter, gauge, time ticks or opaque.

Example: The ipInreceives object has one instance

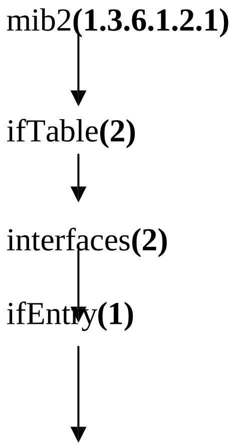


ipInReceives object has one instance.

Tabular Objects :

Two-dimensional table containing zero or more rows .
Each row is made of one or more simple objects (components).
One or more components are used as indexes to unambiguously identifying the rows
The definition of tables is based on ASN.1 types “Sequence" and "Sequence- of "ASN.1 type.

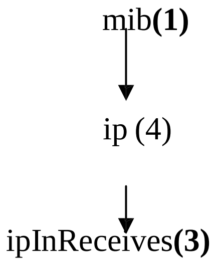
- The table is indexed by ifIndex.
- Each row is an instance of the ifIndex, ifPhysAddress and ifAdminStatus objects



ifIndex**(1)**	ifPhysAddress**(6)**	ifAdminStatus**(7)**	
1	**00:00:39:20:04**	**1 (up)**	row **1**
2	**08:00:56:16:11**	**3 (testing)**	**r**ow **2**
3	**00:00:b4:02:33**	**2 (down)**	row **3**

Instance Identifier:

Instance identifier = Object identifier + 0



...

Object	**Instance identifier**
ipInReceives	mib.4.3.0

The internet node has the object identifier value of 1.3.6.1. This value serves as the prefix for the nodes at the next lower level of the tree.

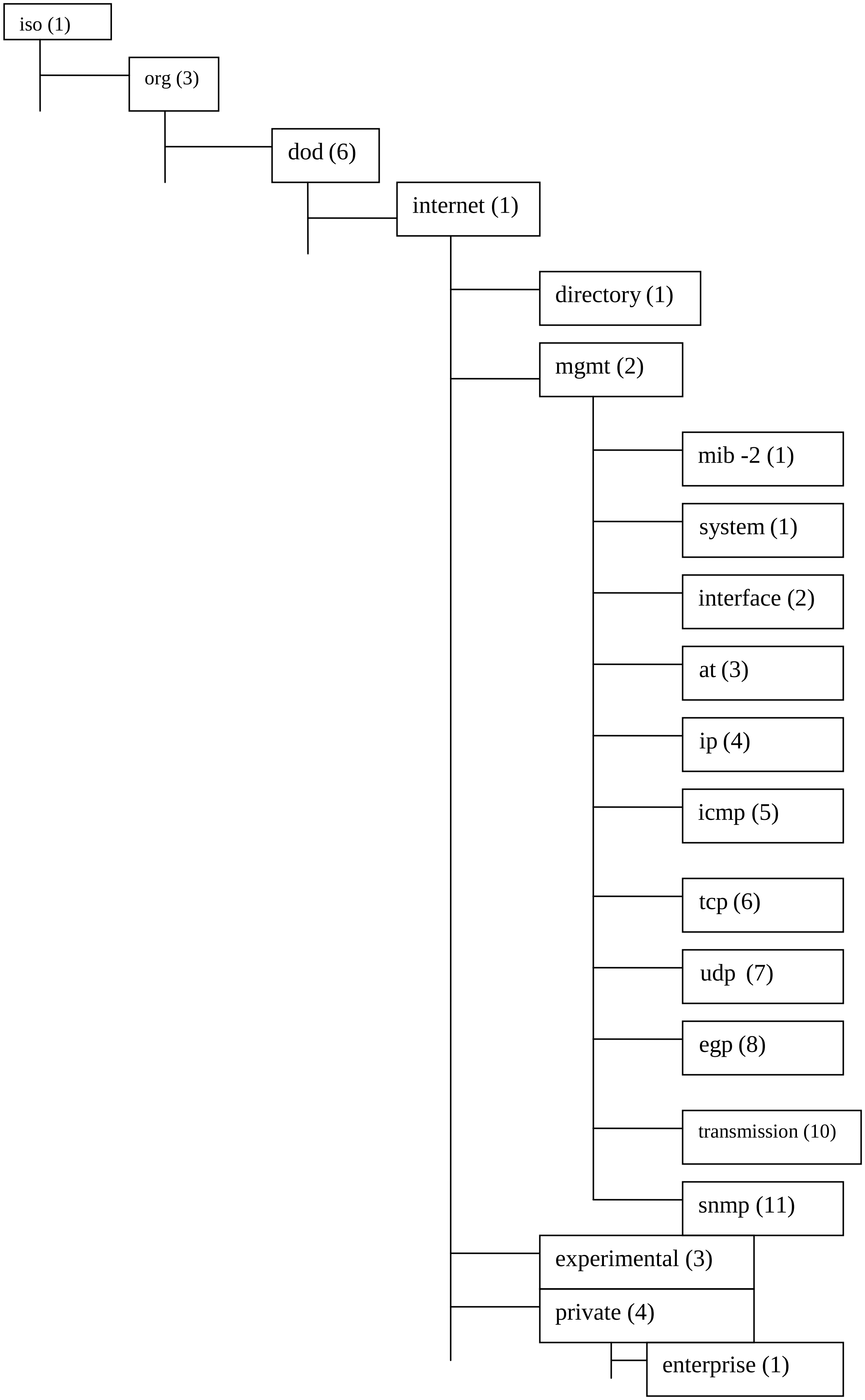
The SMI document defines four nodes under the internet node:

directory: reserved for future use with OSI directory

mgmt : used for objects defined in IAB approved document. (Internet Activity Board)

experimental : Used to identify objects used in internet experiments.

private : used to identify objects defined unilaterally.


iso (1)
org (3)
dod (6)
internet (1)
directory (1)
mgmt (2)
mib -2 (1)
system (1)
interface (2)
at (3)
ip (4)
icmp (5)
tcp (6)
udp (7)
egp (8)
transmission (10)
snmp (11)
experimental (3)
private (4)
enterprise (1)

The **mgmt** subtree consists of the definition of management information Bases that have been approved by the IAB (Internet Activity Board). At present two version of the MIB have been developed, **mib -1** and **mib-2.** The second MIB is an extension of the first. Both are provided with the same object identifier in the subtree . Additional objects can be defined by

- By expanding mib-2
- By creating experimental mib
- By creating private extensions under private tree structure.

MIB I defined 114 objects in 8 groups where as MIB II defined 173 objects with 10 groups.

Standard MIB'S:

MIB II object grouping is given above in the tree format: The only addition with respect to MIB I were addition **transmission** and snmp **node** objects as shown. MIB I was issued as RFC 1156 and the MIB II was defined in the RFC 1213. In this some additional object group are added. The mib II group is sub divided into the following groups.

- **system**: overall information about the system.
- **interfaces**: information about each of the interfaces from the system to a sub network
- **at** : address translation: description of address translation table for the internet to subnet address mapping
- **ip** : information related to the implementation and execution experience of the IP on this system.
- **icmp**: information related to the implementation and execution experience of ICMP on this system.
- **tcp**: information related to the implementation and execution experience of UDP on this sytem.
- **egp**: information related to the implementation and execution experience of EGP (External Gateway Protocol) on this system.
- **dot3 (transmission)**: information about the transmission schemes and access protocols at each system interface.
- **snmp**: information related to the implementation and execution experience of SNMP on this system.

SIMPLE NETWORK MANAGEMENT PROTOCOL

The operation that are supported in SNMP are the alteration and inspection of variables. The three general purpose operations may be performed on scalar objects.

- Get : A management station retrieves a scalar object value from a amanged station.
- Set: A management station updates a a scalar object value in a managed station.
- Tap: A managed station sends an unsolicited scalar object value to a management station.

Few points to understand in this respect are :

- It is not possible to change the structure of a MIB by adding or deleting object instances – that is addition and deletion of a row of a table is not possible.
- It is not possible to issue commands for an action to be performed.
- Access is provided only to an leaf object in the object identifiertree.
- It is not possible to access an entire table or row of a table with one atomic action.

Communities and Community Names:

In SNMP network management, there are a number of managed station that control its won MIB and there are a number of management stations that access some of these agents' MIB as per its requirement. Each MIB managed station controls its won local MIB and must be able to control the use of that MIB by a number of management stations. There are three aspects to this control.

- Authentication service : The managed station may wish to limit the access to the MIB to authorized management stations only.
- Access Policy : The managed stations may wish to give different access privileges to different management stations.
- Proxy Service : A managed station may act as a proxy to other managed stations. This may involve implementing the authentication service and / or access policy for the other managed systems on the proxy system.

An SNMP community is a relationship between an SNMP agent and a set of SNMP managers that defines authentication, access control and proxy characteristics. The managed system establishes one community for each desired combination of authentication, access contro9l and proxy characteristics. Each community is given a unique community name and the management station within that community are provided with and must employ the community name in all get and set opetations. The agent may establish a number of communities with overlapping management station membership.

Authentication Service: In the case of SNMP message, the function of an authentication service would be to assure the recipient that the message is from the source from which it claims to be. The scheme of authentication is that the management station includes the community name which functions like a password. This is although very trivial. But for sensitive application like **set ,** this may trigger an authentication procedure which may involve encryption and decryption procedure.

Access Policy : BY defining a community, the agent limits access to its MIB to a selected set of management stations. BY the use of more than one community, the agent can provide different categories of MIB access to different management stations. The two aspects of this control are :

- SNMP MIB view: a subset of objects with an MIB. Different MIB views may be defined for each community.
- SNMP access control: an element of the set {READ-ONLY, READ-WRITE}. An access mode is defined for each community.

The combination of a **MIB** view and an access mode is referred to as an SNMP community profile. Thus a community profile consists of a defined subset of the MIB at the agent, plus an access mode for those objects.

MIB ACCESS category	SNMP access Mode	
	Read Only	READ-WRITE
Read only	Available for get and trap operation	
Read write	Available for get and trap operations	Available for get, set, and trap Operations
Write only	Available for get and trap operations, but the value is implementation specific	Available for get, set and trap operations, but the value is implementation specific for get and trap operations.
Not accessible	Unavailable	

Proxy Service : Proxy is an SNMP agent that acts on behalf of other devices. Typically other devices do not support SNMP. For each devices that the proxy system represents, it maintains an S?NMP access policy.

Protocol Specification:

With SNMP, information is exchanged between a management station and agent in the form of an SNMP message. Each message includes a version number indicating the version of SNM, a community name to be used for this exchange and one of five types of protocol data units as shown below:

Version	Community	SNMP PDU

SNMP Message Format

PDU Type	**request ID**	**0**	**0**	**Variablebindings**

GetRequest PDU, GetNextRequest PDU and setRequest PDU

PDU type	**Request Id**	**Error status**	**Error index**	**Variable bindings**

GetResponse PDU

PDU type	**Enterprise**	**Agent addr**	**Generic trap**	**Specific trap**	**Timestamp**	**Variablebindings**

Trap PDU

Name1	**Value 1**	**Name2**	**Value2**	**……….**	**Name n**	**Value n**

Variable bindings

GetRequest, getNextRequest and setRequest PDU 's have the same format as the getResponse PDU with error-status and error-index fields set to zero. This convention reduces by one the number of different PDU format with which the SNMP entity must deal.

Details of fields are given below:

Version : SNMP version (RFC 1157 is version 1)
Community : A pairing of an SNMP agent with some arbitrary set of SN?MP application entities
Request-id : Unique ID is provided for each request
Error-status: Indicates occurrence of exception while processing a request.
noError (0), tooBIg(1), noSuchNAme(2), badValues(3), readOnly(4), genErr(5)
error-index: When error status is nonzero, may provide additional information by indicating which variable in a list caused the exception.
Variable bindings: A list of variable names and correcsponding values
Enterprise : Type of object generating tap: based on sysObjectID
Agent-addr: Address of object generating trap.
Generic trap: generic trap type: values are coldStart(0), warmStart(1), linkDown(2), linkUp(3), authentication-Failure(4), egpNeighborLoss (5), enterprise-Specific(6)
SpecificTrap: Specific trap code;
Time-stamp: Time elapsed between the last initialization of the network entity and the generation of the trap.

Transmission of an SNMP message: Steps involved:

1. The PDU is constructed, using the ASN.1 structure defined in RFC 1157
2. This PDU is then passed to an authentication service, together with the source and destination transport address and community name. The authentication performs any required transformations for this exchange, such as encryption or the inclusion of authentication code, and returns the result.
3. The protocol entity then constructs a message, consisting of a version field, the community name and the result from step 2
4. The new ASN.1 object is then encoded using the basic encoding rules and passed to the transport service.

REMOTE NETWORK MONITORING(R M O N)

With MIB II, the network manager can obtain information that is purely local to individual devices. That is, in a LAN with a number of devices on it each with an SNMP agent, the manager can learn of the amount of traffic into and out of each device, but cannot easily learn about the traffic on the LAN as a whole. Network monitors are the devices that are traditionally employed to study the network traffic as a whole. They are also called network analyzers, probes etc. They may be standalone devices or may be workstations a server, or a router with additional monitoring functionality. The monitor can produce summary information, including error statistics, such as count of undersized packets and the number of collision performance statistics (packet delivered per second) etc. The monitor may also store packet for later analysis.

For effective network management, they need to communicate with a central network management station. In this context, they are referred to as remote monitors.

RMON Goals:

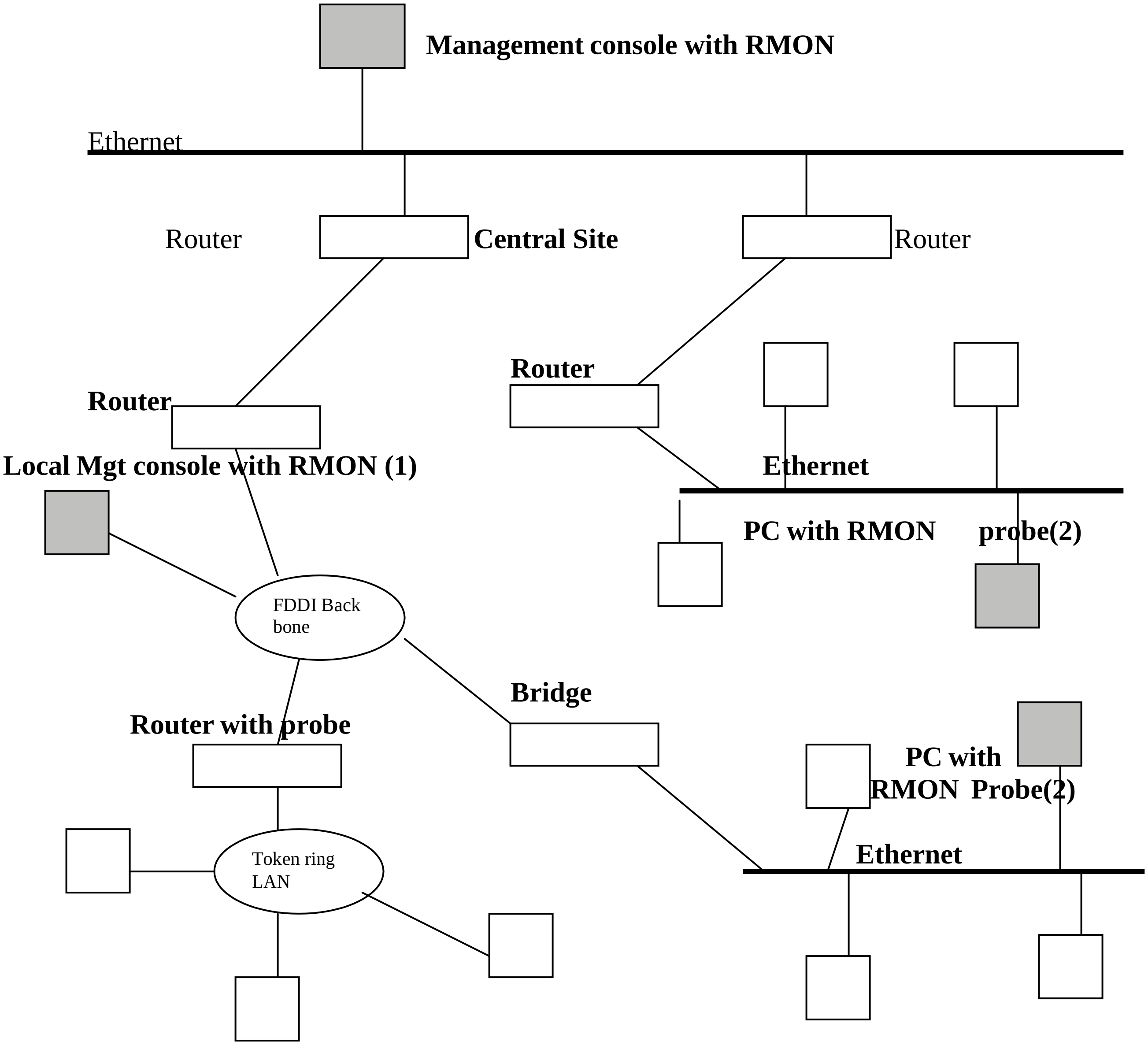
RMON specification is a definition of a MIB. The effect is to define standards network monitoring functions and interfaces for communicating between SNMP based management consoles and remote monitors.

RMON goals as listed in RFC 1757 are given below:

- **Off line operation**: The monitor should collect fault, performance and configuration information continuously. It accumulates statistics that may be retrieved by the manager at a later time. It may also intimate the manager if an exception occurs.
- **Proactive monitoring:** If it is not disruptive, the monitor can run diagnostic and log network performance. In case of failure in the net, the monitor may be able to notify the management station of the failure and provide the management station with information useful in diagnosing the failure.
- **Problem detection and reporting:** Instead of proactive monitoring by running diagnostic tools, it may check for congestion and error condition passively**.** When one of these conditions occur, the monitor can log this condition and attempt to intimate the management station.

- **Value-added data:** The monitor may reduce the work load of management station by performing analysis specific to the data collected on its subnet work.
- **Multiple managers:** An internetworking configuration may have more than one management station to take care of improved reliability, to perform different functions etc. The monitor can be configured to deal with more than one management station concurrently.

Configuration using RMON : Following figure depicts the configuration using RMON.



The top of the figure contains a management station with dedicated RMON capability which is attached to the central LAN. ON the two sub network, RMON MIB is implemented in personal computers.(1 & 2). It may be used exclusively for network performance monitoring or may share other functions. There are other RMON elements that are installed on router as well.

A system that implements the RMON MIB is referred to as RMON probe. The probe has an agent similar to SNMP agent. The probe is capable of reading writing the local RMON MIB in response to management action .

RMON MIB:

The RMON MIB is divided into ten groups

1. **statistics :** Maintains low level utilization and error statistics for each sub net work monitored.
2. **history:** records periodical statistical samples form information available in the statistics group.
3. **alarm**: Allows the management consol user to set a sampling interval and alarm threshold for any counter or integer recorded by the RMON.
4. **host:** contains counters for various types of traffic to and from hosts attached to the subnetwork.
5. **hostTopN** : Contains sorted host statistics that report on the hosts that top a list based on some parameter in the host table.
6. **matrix:** shows error and utilization information in matrix form, so the operator can retrieve information for any pair of network addresses.
7. **filter**: allows the monitor to observe packets that match a filter
8. **packet captures**: governs how data is sent to a management console.
9. **event** : gives a table of all events generated by the RMON probe.
10. **tokenRing :** maintains statistics and configuration information for token ring subnetwork.

SNMP v2 MANAGEMENT INFORMATION

SNMPv2 is a major upgrade over SNMPv1 that expands functionality of SNMP and broadens its applicability to include OSI based as well as TCP/IP based networks. SNMPv1 based on SMI and MIB was quite simple and easy to implement. However, it was not able to take care of the more complex environment containing arbitrary resources and had very low security features. Grouping based on the community name alone for security was inadequate as an attacker can easily observe the message content and find the community name. Because of this SNMPv1 was vulnerable to attacks that can modify or disable a network configuration.

Two different working groups were formed - one to deal with security aspects and other to deal with other aspects including protocol management information. However, over a period of 4 eight years – 1992 - 1996 the issue of security implementation could not be worked out satisfactorily. Hence the SNMPv2 was released without security enhancement.

SNMPv2 can support either a highly centralized network management strategy or a distributed one. IN the latter case, some systems operate in the role of both manager and agent. In its agent role, such a system will accept commands from a superior management system; these commands may deal with access to information stored locally at the intermediate manager or may require the intermediate manager to provide summary information about agents subordinate to itself.

The key enhancements to SNMPv2 are in the following category:

- **Structure of Management Information (SMI)** : This deals with Object definition, Conceptual Tables, Notification definition and information modules.
- **Manager to Manager to capability**.
- **Protocol Operation.**

IN this the macro used to define objects types has been expanded to include several new data types and to enhance the documentation associated with an object. A new convention has been provided for creating and deleting conceptual rows in a table. SNMPv2 MIB contains basic traffic information about operation of the SNMPv2 protocol. It includes two new PDUs.

SNMPv3 : The final form of SNMPv2 contains no provision for security. This deficiency is removed in SNMPv3. The documents (RFC 2271,72,73,74, &75) define a set of security capability and a framework that enables that set to be used with the SNMPv2 or SNMPv1.

ABSTRACT SYNTAX NOTATION ONE (ASN.1)

ABSTRACT SYNTAX NOTATION ONE (ASN.1) IS A FORMAL LANGUAGE FOR ABSTRACTLY DESCRIBING MESSAGES TO BE EXCHANGED AMONG AN EXTENSIVE RANGE OF APPLICATIONS INVOLVING THE INTERNET, INTELLIGENT NETWORK, CELLULAR PHONES, GROUND-TO-AIR COMMUNICATIONS, ELECTRONIC COMMERCE, SECURE ELECTRONIC SERVICES, INTERACTIVE TELEVISION, INTELLIGENT TRANSPORTATION SYSTEMS, VOICE OVER IP AND OTHERS. DUE TO ITS STREAMLINED ENCODING RULES, ASN.1 IS ALSO RELIABLE AND IDEAL FOR WIRELESS BROADBAND AND OTHER RESOURCE-CONSTRAINED ENVIRONMENTS.

EXAMPLE OF A MESSAGE DEFINITION SPECIFIED WITH ASN.1 NOTATION:

```
Report ::= SEQUENCE {
    author          OCTET
    STRING, title  OCTET
    STRING, body    OCTET
    STRING,
   }
```

REPORT" IS THE NAME OF THIS TYPE OF MESSAGE. SEQUENCE INDICATES THAT THE MESSAGE IS A SEQUENCE OF DATA ITEMS.

THE FUNDAMENTAL UNIT OF ASN.1 IS THE MODULE. THE SOLE PURPOSE OF A MODULE IS TO NAME A COLLECTION OF TYPE DEFINITIONS AND/OR VALUE DEFINITIONS (ASSIGNMENTS) THAT CONSTITUTE A DATA SPECIFICATION. A TYPE DEFINITION IS USED TO DEFINE AND NAME A NEW TYPE BY MEANS OF A TYPE ASSIGNMENT

AND A VALUE DEFINITION IS USED TO DEFINE AND NAME A SPECIFIC VALUE, WHEN IT IS NECESSARY, BY MEANS OF A VALUE ASSIGNMENT.

THE FIGURE BELOW CONTAINS AN EXAMPLE MODULE. IT IS DEFINED AS A *MODULE REFERENCE* INVENTORYLIST, FOLLOWED BY AN OPTIONAL *OBJECT IDENTIFIER* VALUE 1 2 0 0 6 1 (SEE THE SIMPLE TYPES SECTION.), FOLLOWED BY THE KEYWORD DEFINITIONS, FOLLOWED BY THE OPTIONAL *TAG DEFAULT* (NOT INCLUDED IN THE EXAMPLE), FOLLOWED BY THE ASSIGNMENT CHARACTER SEQUENCE ::= , FOLLOWED BY THE KEYWORDS BEGIN AND END BRACKETING THE *MODULE BODY*.

```
InventoryList {1 2 0 0 6 1} DEFINITIONS ::=
  BEGIN
    {
     ItemId ::= SEQUENCE
       {
        partnumber IA5String,
        quantity INTEGER,
        wholesaleprice REAL,
        saleprice REAL
       }
     StoreLocation ::= ENUMERATED
       {
        Baltimore (0),
        Philadelphia (1),
        Washington (2)
       }
    }
  EN
  D
```

Figure: Example of an ASN.1 module.

TYPE ASSIGNMENT:

A TYPE ASSIGNMENT CONSISTS OF A TYPE REFERENCE (THE NAME OF THE TYPE), THE CHARACTER SEQUENCE ::= (‘‘IS DEFINED AS’’), AND THE APPROPRIATE TYPE.

VALUE ASSIGNEMENT

```
gadget   ItemId   ::=
  {
   partnumber       "7685B2",
   quantity         73,
   wholesaleprice   13.50,
```

saleprice	**24.95**

)

BUILT IN TYPE:

SN.1'S BUILT-IN SIMPLE TYPES ARE SHOWN IN THE FOLLOWING TABLE . THE UNIVERSAL CLASS NUMBER (TAG) AND A TYPICAL USE OF EACH TYPE ARE ALSO INCLUDED.

Simple Types	Tag	Typical Use
BOOLEAN	1	Model logical, two-state variable values
INTEGER	2	Model integer variable values
BIT STRING	3	Model binary data of arbitrary length
OCTET STRING	4	Model binary data whose length is a multiple of eight
NULL	5	Indicate effective absence of a sequence element
OBJECT	6	Name information objects
REAL	9	Model real variable values
ENUMERATED	10	Model values of variables with at least three states
CHARACTER STRING	*	Models values that are strings of characters from a specified characterset

Type BOOLEAN takes values TRUE and FALSE. Usually, the type reference for BOOLEAN describes the true state.

TYPE INTEGER TAKES ANY OF THE INFINITE SET OF INTEGER VALUES. ITS SYNTAX IS SIMILAR TO PROGRAMMING LANGUAGES SUCH AS C OR PASCAL. IT HAS AN ADDITIONAL NOTATION THAT NAMES SOME OF THE POSSIBLE VALUES OF THE INTEGER. FOR EXAMPLE,

```
ColorType ::= INTEGER
      {
            red          (0)
            white        (1)
            blue         (2)
      }
```

TYPE BIT STRING TAKES VALUES THAT ARE AN ORDERED SEQUENCE OF ZERO OR MORE BITS.

```
OCCUPATION     ::=    BIT STRING
               {
                     clerk          (0)
```

```
                editor        (1)
                artist        (2)
                publisher     (3)
            }
```

Names the first bit ``clerk", the second bit ``editor", and so on. Strings of bits can then be written by listing the named bits that are set to 1. For example, (editor, artist) and '0110'b are two representations for the same value of ``occupation".

TYPE OCTET STRING TAKES VALUES THAT ARE AN ORDERED SEQUENCE OF ZERO OR MORE EIGHT-BIT OCTETS.

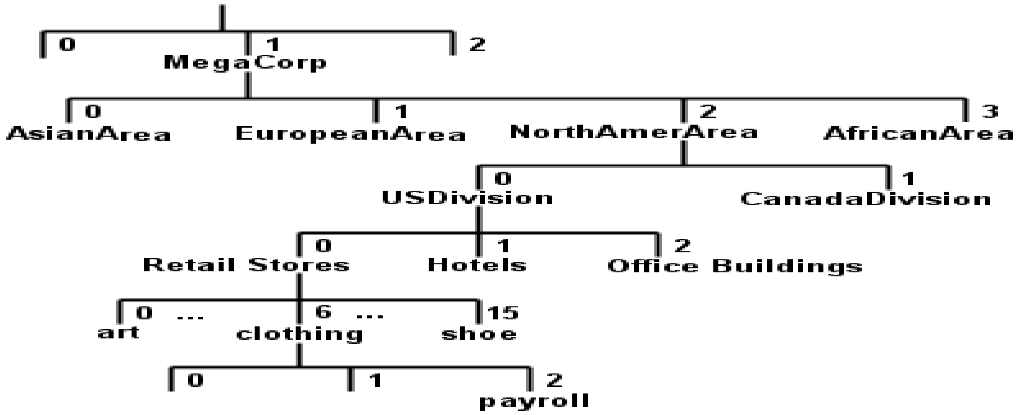
TYPE NULL TAKES ONLY ONE VALUE, NULL. IT CAN BE USED AS A PLACE MARKER, BUT OTHER ALTERNATIVES ARE MORE COMMON.

TYPE OBJECT IDENTIFIER NAMES INFORMATION OBJECTS (FOR EXAMPLE, ABSTRACT SYNTAXES OR ASN.1 MODULES). THE TYPE NOTATION REQUIRES THE KEYWORDS OBJECT IDENTIFIER. THE NAMED INFORMATION OBJECT IS A NODE ON AN OBJECT IDENTIFIER TREE
THAT IS MANAGED AT THE INTERNATIONAL LEVEL. ISO, CCITT, OR ANY OTHER ORGANIZATION IS ALLOWED A SUBTREE WHICH THE ORGANIZATION DEFINES. ON EACH LEVEL J OF THE OBJECT IDENTIFIER TREE, NODES ARE NUMBERED 0,1,2,.... A LIST OF POSITIVE NUMBERS, ENCLOSED IN BRACES AND ORDERED BY LEVEL STARTING FROM THE ROOT, UNIQUELY IDENTIFIES AN INFORMATION OBJECT AT A NODE OF THE TREE. THIS ORDERED LIST OF POSITIVE NUMBERS DELIMITED BY BRACES IS THE VALUE NOTATION FOR TYPE OBJECT IDENTIFIER.

THE FOLLOWINGFIGURE ILLUSTRATES THE CONCEPT OF AN OBJECT IDENTIFIER TREE. FOR EXAMPLE, IN THE SUBTREE WITH ROOT ``RETAILSTORES'' THE INFORMATION OBJECT ``PAYROLL'' HAS LOCAL VALUE 0 6 2. MORE FORMALLY, IF

```
    CLOTHINGTYPE     ::=   OBJECT IDENTIFIER
              THEN PAYROLL      CLOTHINGTYPE
              ::=     {0 6 2}.
```

IF THE RETAIL STORES ARE CONSIDERED AS PART OF AN INTERNATIONAL ``MEGACORP'' THEN 1 2 0 0 6 2 UNIQUELY IDENTIFIES ``PAYROLL''.



TYPE ENUMERATED IS SIMILAR TO THE INTEGER TYPE, BUT NAMES SPECIFIC VALUES ONLY. FOR EXAMPLE,

```
ColorType ::= ENUMERATED
    {
        red       (0)
        white     (1)
        blue      (2)
    }
```

HAS THE SAME INTERPRETATION AS IN THE TYPE INTEGER EXAMPLE NEAR THE BEGINNING OF THIS SECTION, EXCEPT THAT COLORTYPE CAN TAKE ONLY THE VALUES SPECIFICALLY IN THE LIST; THAT IS, NO OTHER VALUES THAN 0 FOR ``RED", 1 FOR ``WHITE", OR 2 FOR ``BLUE".

TYPE CHARACTER STRING TAKES VALUES THAT ARE STRINGS OF CHARACTERS FROM SOME DEFINED (ISO- OR CCITT-REGISTERED) CHARACTER SET.

Character String Type	Tag	Character Set
`NumericString`	18	0,1,2,3,4,5,6,7,8,9, and space
`PrintableString`	19	Upper and lower case letters, digits, space, apostrophe, left/right parenthesis, plus sign, comma, hyphen, full stop, solidus, colon, equal sign, question mark

`TeletexString`
`(T61String)`
	20	The Teletex character set in CCITT's T61, space, and delete

`VideotexString`	21	The Videotex character set in CCITT's T.100 and T.101, space, and delete

VisibleString (ISO646String)	**26**	**Printing character sets of international ASCII, and space**
IA5String	**22**	**International Alphabet 5 (International ASCII)**
GraphicString	**25**	**All registered G sets, and space**
GraphicString	**27**	**All registered C and G sets, space and delete**

STRUCTURED TYPES

ASN.1'S BUILT-IN STRUCTURED TYPES ARE SHOWN IN THE FOLLOWING TABLE. THE UNIVERSAL CLASS NUMBER (TAG) AND A TYPICAL USE OF EACH TYPE ARE ALSO INCLUDED.

Structured Types	Tag	Typical Use
SEQUENCE	**16**	**Models an ordered collection of variables of different type**
SEQUENCE OF	**16**	**Models an ordered collection of variables of the same type**
SET	**17**	**Model an unordered collection of variables of different types**
SET OF	**17**	**Model an unordered collection of variables of the same type**
CHOICE	*****	**Specify a collection of distinct types from which to choose one type**
SELECTION	*****	**Select a component type from a specified CHOICE Type**
ANY	*****	**Enable an application to specify the type Note: ANYis a deprecated ASN.1 Structured Type. It has been replaced with X.680 Open Type.**

Type SEQUENCE is an ordered list of zero or more component types. The type notation requires braces around the list and permits a local identifier preceding the list to act as the name of the sequence type.

```
     AirlineFlight      ::=
                            SEQUENCE
  {
   airline       IA5String,
   flight        NumericString,
seats       SEQUENCE
                 {
                   maximum
                                INTEGE
                   R,             occupied
                                INTEGE
                   R,                vacant
                                INTEGE
                   R
                 },
airport
            SEQUEN
CE
                 {
                   origin                    IA5String,
                   stop1            [0]      IA5String
                                    OPTIONAL, stop2   [1]
                                    IA5String
                                    OPTIONAL, destination
                                    IA5String
                 },
 crewsize
 ENUMERATED
                 {
                     six      (6),
                     eight    (8),
                     ten      (10)
                 },
 cancel     BOOLEAN  DEFAULT FALSE
}.
```

THIS INSTANCE OF AIRLINEFLIGHT INDICATES THAT AMERICAN AIRLINES FLIGHT 1106 FLIES NON-STOP FROM BALTIMORE-WASHINGTON AIRPORT TO LOS ANGELES. THE AIRPLANE REQUIRES A CREW OF 10 PEOPLE, HAS 320 SEATS, OF WHICH 107 ARE FILLED AND 213 ARE EMPTY. THE FLIGHT IS NOT CANCELED. TWO COMPONENTS, STOP1 AND STOP2 OF THE SEQUENCE TYPE AIRPORT ARE TAGGED WITH THE CONTEXT-SPECIFIC TAGS [0] AND [1] TO AVOID AMBIGUITY DUE TO CONSECUTIVE OPTIONAL COMPONENTS NOT HAVING DISTINCT TYPES. WITHOUT THE TAGS, THE DEFINITION OF AIRPORT WOULD BE INVALID IN ASN.1.

TYPE SEQUENCE OF IS SIMILAR TO SEQUENCE, EXCEPT THAT ALL VALUES IN THE ORDERED LIST MUST BE OF THE SAME TYPE. FOR EXAMPLE, THE SEATS TYPE IN THE ABOVE EXAMPLE COULD BE SEQUENCE OF INTEGER INSTEAD OF SEQUENCE.

TYPE SET TAKES VALUES THAT ARE UNORDERED LISTS OF COMPONENT TYPES. THE

TYPE AND VALUE NOTATIONS FOR SET ARE SIMILAR TO SEQUENCE, EXCEPT THAT THE TYPE OF EACH COMPONENT MUST BE DISTINCT FROM ALL OTHERS AND THE VALUES CAN BE IN ANY ORDER.

Person ::= SET

```
    {
     name        IA5String,
     age         INTEGER,
     female      BOOLEAN
    }.
```

TYPE SET OF TAKES VALUES THAT ARE UNORDERED LISTS OF A SINGLE TYPE. THE SEQUENCE TYPE EXAMPLE ABOVE WOULD BE VALID IF THE SEATS TYPE WERE SET OF INTEGER INSTEAD OF SEQUENCE.

Type CHOICE takes one value from a specified list of distinct types.

```
  Prize    ::=    CHOICE
     {
      car           IA5String,
      cash          INTEGER,
      nothing       BOOLEAN
     }.
```

Type SELECTION enables the user to choose a component type from a specified CHOICE type.

```
  Winner  ::=    SEQUENCE
     {
      lastName      VisibleString,
      ssn           VisibleString,
      cash     <    Prize
     }
```

TAGGED

TYPE TAGGED IS USED TO ENABLE THE RECEIVING SYSTEM TO CORRECTLY DECODE VALUES FROM SEVERAL DATATYPES THAT A PROTOCOL DETERMINES MAY BE TRANSMITTED AT ANY GIVEN TIME.

ITS TYPE NOTATION CONSISTS OF THREE ELEMENTS: A USER-DEFINED TAG, FOLLOWED BY THE VALUE NOTATION OF THE TYPE BEING TAGGED.

THE USER-DEFINED TAG CONSISTS OF A CLASS AND CLASS NUMBER CONTAINED IN BRACES. CLASS IS UNIVERSAL, APPLICATION, PRIVATE, OR CONTEXT-SPECIFIC. THE UNIVERSAL CLASS IS RESTRICTED TO THE ASN.1 BUILT-IN TYPES. IT DEFINES AN APPLICATION-INDEPENDENT DATA TYPE THAT MUST BE DISTINGUISHABLE FROM ALL OTHER DATA TYPES. THE OTHER THREE CLASSES ARE USER DEFINED. THE APPLICATION CLASS DISTINGUISHES DATA TYPES THAT HAVE A WIDE, SCATTERED USE WITHIN A PARTICULAR PRESENTATION CONTEXT. PRIVATE DISTINGUISHES DATA TYPES WITHIN A

PARTICULAR ORGANIZATION OR COUNTRY. CONTEXT-SPECIFIC DISTINGUISHES MEMBERS OF A SEQUENCE OR SET, THE ALTERNATIVES OF A CHOICE, OR UNIVERSALLY TAGGED SET MEMBERS. ONLY THE CLASS NUMBER APPEARS IN BRACES FOR THIS DATA TYPE; THE TERM COONTEXT-SPECIFIC DOES NOT APPEAR.

Character String Types

The most common Character String Types are listed as follows:

Character Type	Tag	Description
BMPString	**30**	**Basic Multilingual Plane of ISO/IEC/ITU 10646-1**
IA5String	**22**	**International ASCII characters (International Alphabet 5)**
GeneralString	**27**	**all registered graphic and character sets plus SPACE and DELETE**
GraphicString	**25**	**all registered G sets and SPACE**
NumericString	**18**	**1, 2, 3, 4, 5, 6, 7, 8, 9, 0, and SPACE**
PrintableString	**19**	**a-z, A-Z, ' () +,-.?:/= and SPACE**
TeletexString	**20**	**CCITT and T.101 character sets**
UniversalString	**28**	**ISO10646 character set**
UTF8String	**12**	**any character from a recognized alphabet (including ASCII control characters)**
VideotexString	**21**	**CCITT's T.100 and T.101 character sets**
VisibleString	**26**	**International ASCII printing character sets**

Macros in ASN.1 are similar to macros in application software, they provide the capability of defining types and values that are not included in the standard repertoire One significant use of ASN.1 macros is in OSI application protocol standards, specifically for defining remote operations and object classes. In this section, we include two macros, ERROR and OPERATOR, that appear in the common service elements

```
<macro name> MACRO
        ::= BEGIN
```

```
    TYPE NOTATION::=      <user-defined type
    notation> VALUE NOTATION  ::=      <user-
    defined value notation>
  <supportin
  g syntax>
  END
```

THE FOLLOWING ERROR MACRO DEFINED IN X.219 PROVIDES A SPECIFIC INSTANCE OF THE GENERAL TEMPLATE.

```
  ERROR          MACRO    ::= BEGIN
    TYPE NOTATION::=       Parameter
    VALUE NOTATION          ::=   value (VALUE CHOICE
                                         {
                                           localValue
                                                          INTEGE
                                           R, globalValue OBJECT
                                           IDENTIFIER
                                         }
                                         )
    Parameter             ::=       ``PARAMETER'' NamedType | empty
    NamedType             ::=       identifier type         |    type
  END
```

IN THIS DEFINITION, DETAILS OF PARAMETER AND NAMEDTYPE ARE IN THE SUPPORTING SYNTAX. PARAMETER CONSISTS OF THE KEYWORD ``PARAMETER'' FOLLOWED BY A NAMED TYPE; IT MAY NOT HAVE AN ENTRY. THE VALUE NOTATION IS A CHOICE OF INTEGER OR OBJECT IDENTIFIER. THE DEFINITION ALLOWS USERS TO DEFINE OPERATION ERRORS. FOR EXAMPLE, THE ERROR MACRO IS USED IN THE REMOTE OPERATIONS SERVICE ELEMENT (ROSE) OF A FOLLOWING CHAPTER TO DEFINE BADQUEUENAME AS FOLLOWS:

```
  BadQueueName  ERROR
                 PARAMETE
                     R

                 QueueName
                  ::= 0
```

BadQueueName HAS TYPE ERROR, ONE PARAMETER ``queuename'' (IDENTIFIED ELSEWHERE AS TYPE IA5STRING), AND VALUE 0. IN THE REMOTE OPERATION, ONLY THE VALUE 0 IS TRANSMITTED, THE OTHER TERMS IN

THE DEFINITION ARE FOR THE USER'S BENEFIT.

Short Questions

1. Define SNMP.
2. What are the key elements of TCP/IP network management?
3. Define MIB.
4. Define message agent.
5. Define SMI.
6. What are the four nodes under the internet node in SMI document?
7. Explain private MIBs.
8. What are the limitations of MIBs?
9. What are the limitations of SNMP?
10. Define RMON.
11. What are the design goals of RMON?
12. Explain RMON MIB.
13. What are the advantages of SNMPv2?
14. Explain SNMPv3.
15. What are the disadvantages of SNMPv1/v2?
16. Where does RMON used in network?

Big Questions

1. a) Explain the data types in UNIVERSAL class of ASN.1 for SNMP MIB. (08)
   b) Write notes on Network configuration control. (08)
2. Explain the syntax of the various SNMPv1 message formats. (16)
3. Explain the architecture of SNMP entity and traditional SNMP manager, as specified in RFC2271. (16)
4. Explain the architecture of SNMPV3 with neat diagram. (16)
5. a) Compare SNMPV2 and SNMPV3. (06)
   b) Discuss about MIB. (06)
   c) Write note on RMON. (04)

NPM/QB/1

QUESTION BANK

Part – A

Unit I

1. What is socket address structure? Write down posix definition for the same.
2. Explain briefly the byte order functions
3. Write the syntax for socket function
4. Write the syntax for connect function
5. Write the syntax for bind function
6. Write the syntax for listen function
7. Write the syntax for accept function
8. Distinguish concurrent server from iterative server
9. What function has to be called to create n no. of child ?
10. What is descriptor reference count?

Unit II

1. Draw the diagram that depicts simple client server along with functions used for input and output
2. Expansion for posix
3. What is signal ? how it can be sent?
4. What is disposition and how it can be set?
5. What are the choices for disposition ?
6. Distinguish termination of server process from crashing of server host
7. Explain the scenario crashing and re-booting of server host
8. How does shout down of server host take place ?
9. What is mean by i/o multiplexing?
10. Specify the scenario where networking applications can be done using i/o multiplexing.
11. What are the i/o models available?
12. Distinguish synchronous i/o and asynchronous i/o
13. Write down the syntax for the function select
14. What are the limitation with close that can be avoided with shutdown

Unit III

1. Explain the function getsockopt() and setsockopt().
2. Specify some of the generic socket option
3. Mention IPV4 socket option
4. Explain the ICMPV6 socket option
5. Mention the IPV6 socket option
6. Mention the TCP socket option
7. Write the syntax for
    - i . gethostbyname
    - ii. gethostbyaddr
    - iii. getservbyname
    - iv. getservbyport

8. What is DNS?
9. What is resolver?
10. Mention the resolving functions

Unit IV

1. Summarize the interoperability for IPV4/IPV6 clients and servers
2. What are threads?
3. Address the problems with fork that can be shared by threads of a process and that can be solved by threads
4. Write the syntax for
    pthread – create
    pthread – join
    pthread – self
    pthread – detach
    ptheard – exit
5. What are different ways to terminate thread
6. What are the functions that can be used with mutex?
7.What are the functions that can be used with conditional variables?

Unit V

1. What is network management station? Write down the functionalities
2. What is trap directed polling
3. What is object identifier?
4. How additional objects can be defined for a MIB?
5. What are the operations supported by SNMP?
6. What are the limitations of SNMP?
7. Expand RMON
8. What are the design goals of RMON
9. Is that possible to have shared RMON. If so, what are difficulties that may arrive.
10. What are the four nodes defined under the internet node by SMI documents?
11. What are the data types that are available under Universal class of ANS.1?

Part – B

Unit I

1. explain in detail the socket functions for elementary TCP functions with a neat diagram.

2. Explain in detail the parameters of the functions :Socket(),Connect(),Bind() and also explain the Error possibilities.

3. Explain the function listen() and how 2 queues are maintained in TCP along with 3 way handshaking
4. Write day time server that prints client IP address and port
5. Explain in detail about the concurrent server. Distinguish it from Iterative Server

Unit II

1. With program explain TCP echo client server application
2. Explain in detail POSIX Signal handling
3. Explain the various Boundary conditions
4. Explain in detail the various IO models
5. Explain in detail the Select(),Poll() with the necessary code

Unit III

1. Explain in detail the socket options

   a)Generic Socket option
   b)IPV4
   c)IPV6
   d)TCP
2. With a example program explain UDP client server application
3. Explain in detail DNS

Unit IV

1. Explain in detail the interoperability between IPV4 and IPV6
2. Explain the various thread functions
3. Write TCP Echo Client server program using threads
4. explain in detail the raw socket functions
5. Write down the main definition for PING and Traceout with necessary code

Unit V

1. Explain the SNMP management information
2. Explain SNMP V2 Concepts in detail
3. Explain SNMP V3 Concepts in detail
4. Explain RMON Concepts in detail

UNIVERSITY QUESTION PAPER

IT2351 –Network Programming and Management

NOV/DEC-2008

PART – A

1.What are the steps involved in creating a socket on the client side?
2.Define iterative server?
3.What is the difference between the close() system call and shutdown system call?
4.Write about poll function?
5.What is the important feature of ICMP?
6. What is the function of gethostbyname()?
7.What is the significance of ping?
8.Define threaded server?
9.What are the objectives of MIB?
10.What are the goals of RMON?

PART-B

11. (a) Explain in detail about the TCP/IP layers for internet working and management (16)

Or

11.(b) Write short notes on (i) Iterative server
(ii) Concurrent server (16)

12.(a) Explain the following concept
(i) Server with multiple clients
(ii) Server process crashes
(iii) Server host crashes
(iv) Server crashes and boots (16)

Or

12.(b) Write notes on select function and poll function (16)

13.(a) Discuss about IP socket options and ICMP socket options in detail with suitable example (16)

Or

13.(b) Write a UDP Socket program to implement a echo server and echo client (16)

14.(a) Write short notes on

(i) IPV4 & IPV6 interoperability

(ii) Mutexes (16)

Or

14.(b) Explain the trace route program with sample code and example (16)

15.(a) Explain the architecture of SNMP in detail (16)

Or

15.(b) Explain the following :

(i) RMON

(ii) Compare SNMPV2 & SNMPV3 (16)

UNIVERSITY QUESTION PAPER

IT2351 –Network Programming and Management

April/May-2008

PART – A

1.Why must value result argument such as the length of a socket address structure be passed by reference ?

2.Write a program to return the address family of a socket ?

3.Explain the syntax of the Signal Function ?

4.Assume that in concurrent server the child runs first after the call to fork .The child then complete the service of the client before the call to fork returns to the parent .What happens in the two calls to close ?

5.State the role of pointer queries in DNS?

6. Show the members of the structure , which is referred by a function that looks up for a hostname?

7.How the server knows about the version of the communicating client?

8.List out the unique values maintained by a thread?

9.Specify the MIP-II specification of TCP connection entries ?

10.List out few performance indicators in a network management system?

PART-B

11. (a) (i) Compare the implementation details of concurrent and iterative server (8)

11.(a) (ii) Discuss the syntactical issues of various address conversion functions (8)

Or

11.(b) (i) Compare the IPV4,IPV6,UNIX domain and data link socket address structure . State your assumptions (8)

11.(b)(ii) Write notes on byte ordering functions (8)

12.(a) Discuss about following scenarios of server operations

(i) Crashing of Server host (6)

(ii) Crashing and rebooting of Server host (6)

(iii) Shutdown of Server host (4)

Or

12.(b)Explain in detail about the various I/O models in UNIX operating system (16)

13.(a) (i) Assume both a client and server set the SO_KEEPALIVE socket options and the connectivity is maintained between the peers but them is no exchange of data .When the keep alive timer expires every 2 hours , how many TCP segments are exchanged across the connection ? Justify your answer with an illustration ?

(6)

13.(a)(ii) Write program that checks all the socket options of a sockets and sets the value for receiver Buffer size to 520 bytes. (10)

Or

13.(b)(i) Write notes on RES_USE_INET6 resolver options in gethostbyname and gethostbyname2 functions
(8)

13.(b)(ii) Discuss any four TCP socket options in details (8)

14.(a) (i) Compare

(a) Fork and Thread (4)

(bi) Wait and Waitpid with code (4)

(ii) Write a ‘C’ program that can generate an ICMP V4 echo request packet and process the received ICMPV4 echo reply (8)

Or

14.(b) Write notes on

(i) Raw socket creation (4)

(ii) Raw socket output (6)

(iii) Raw socket input (6)

15.(a) (i) Explain the data types in UNIVERSAL class of ASN.1 for SNMP MIB (8)

15.(a)(ii) Write notes on Network configuration control (8)

Or

15.(b) Explain the Syntax of the various SNMPV1 message formats (16)