SATHYABAMA UNIVERSITY

(Established under Section 3, UGC Act 1956)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SCSX1024 NETWORK PROGRAMMING AND MANAGEMENT

UNIT – I

ELEMENTARY TCP SOCKETS **10 hrs.**

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.

1. Introduction to Socket programming

- A socket is an endpoint used by a process for bi-directional communication with a socket associated with another process.
- Sockets, introduced in Berkeley Unix, are a basic mechanism for IPC on a computer system, or on different computer systems connected by local or wide area networks.

2. Overview of TCP/IP Protocols

The TCP/IP protocol suite maps to a four-layer conceptual model known as the DARPA model, which was named after the U.S. government agency that initially developed TCP/IP. The four layers of the DARPA model include the following.

- Application layer
- Transport layer
- Internet layer
- Network Interface layer

Each layer in the DARPA model corresponds to one or more layers of the seven-layer OSI model. Fig 1.1 shows the TCP/IP protocol suite.

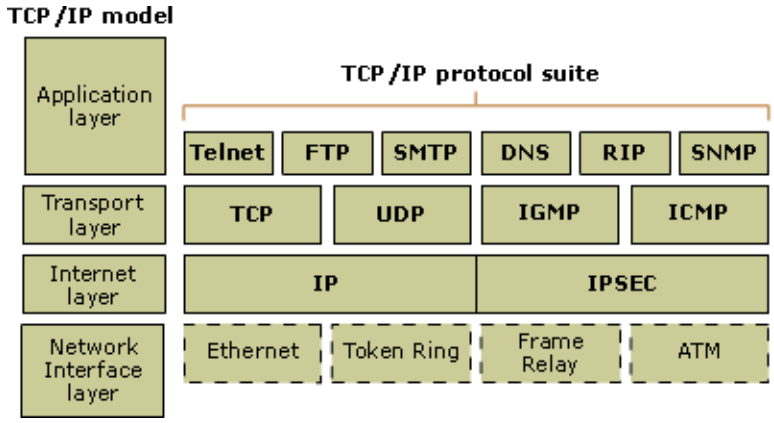


Fig 1.1 TCP/IP protocol suite

2.1 Layers

2.1.1 Application layer

The Application layer allows applications to access the services of the other layers, and it defines the protocols that applications use to exchange data. The Application layer contains many protocols, and more are always being developed.

The most widely known Application layer protocols help users exchange information:

- The Hypertext Transfer Protocol (HTTP) transfers files that make up pages on the World Wide Web.
- The File Transfer Protocol (FTP) transfers individual files, typically for an interactive user session.
- The Simple Mail Transfer Protocol (SMTP) transfers mail messages and attachments.
- The Domain Name System (DNS) protocol resolves a host name, such as www.microsoft.com, to an IP address and copies name information between DNS servers.
- The Routing Information Protocol (RIP) is a protocol that routers use to exchange routing information on an IP network.
- The Simple Network Management Protocol (SNMP) collects and exchanges network management information between a network management console and network devices such as routers, bridges, and servers.

2.1.2 Transport layer

- The Transport layer (also known as the Host-to-Host Transport layer) provides the Application layer with session and datagram communication services.
- The Transport layer encompasses the responsibilities of the OSI Transport layer.
- The core protocols of the Transport layer are TCP and UDP.

TCP

- TCP provides a one-to-one, connection-oriented, reliable communications service.
- TCP establishes connections, sequences and acknowledges packets sent, and recovers packets lost during transmission.

UDP

- UDP provides a one-to-one or one-to-many, connectionless, unreliable communications service.

- UDP is used when the amount of data to be transferred is small (such as the data that would fit into a single packet), when an application developer does not want the overhead associated with TCP connections, or when the applications or upper-layer protocols provide reliable delivery.TCP and UDP operate over both IPv4 and IPv6 Internet layers.

2.1.3 Network Interface Layer

- The Network Interface layer (also called the Network Access layer) sends TCP/IP packets on the network medium and receives TCP/IP packets off the network medium.
- TCP/IP was designed to be independent of the network access method, frame format, and medium.

2.1.4 Internet Layer

The Internet layer responsibilities include addressing, packaging, and routing functions. The Internet layer is analogous to the Network layer of the OSI model. There are two versions of IP. They include IPv4 and IPv6.

The core protocols for the IPv4 Internet layer consist of the following:

- The Address Resolution Protocol (ARP) resolves the Internet layer address to a Network Interface layer address such as a hardware address.
- The Internet Protocol (IP) is a routable protocol that addresses, routes, fragments, and reassembles packets.
- The Internet Control Message Protocol (ICMP) reports errors and other information to help you diagnose unsuccessful packet delivery.
- The Internet Group Management Protocol (IGMP) manages IP multicast groups.

3. Introduction to sockets

Sockets are communication points on the same or different computers to exchange data. Sockets are supported by Unix, Windows, Mac, and many other operating systems.

3.1 Socket Types

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

3.1.1 Stream Sockets

- Delivery in a networked environment is guaranteed.
- These sockets use TCP (Transmission Control Protocol) for data transmission.
- If delivery is impossible, the sender receives an error indicator.
- Data records do not have any boundaries.

3.1.2 Datagram Sockets

- Delivery in a networked environment is not guaranteed.
- They're connectionless but don't need to have an open connection as in Stream Sockets

3.1.3 Raw Sockets

- These provide users access to the underlying communication protocols, which support socket abstractions.
- These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol.
- Raw sockets are not intended for the general user.
- They have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

3.1.4 Sequenced Packet Sockets

- They are similar to a stream socket, with the exception that record boundaries are preserved.
- This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications.
- Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

4. Socket address structure

Most of the socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of this structure begin with sockaddr with a unique suffix for each protocol suite. Table 1.1 shows the different data types and their descriptions.

Table 1.1 Data type and description

Data Type	Description	Header
Int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address stucture	<sys/types.h>
socketlen_t	Length of socket address structure	<sys/types.h>
In_addr_t	Ipv4 address	<netinet/in.h>
In_port_t	TCP or UDP port	<netinet/in.h>

4.1 IPv4 Socket Address Structure

The 1PV4 address structure, commonly called an " Internet socket address structure ," is named sockaddr _in and defined by including the <netinet/in.h> header.

```
Struct in_addr        {  /*32-bit 1PV4 address*/

In_addr_t  s_addr; /* network byte ordered*/

};

Struct sockaddr_in {

   Uin8_t      sin_len     /* length of structure (16)*/

   Sa-family_t sin_family; /* AF_INST*/

   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*/ };
```

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in network byte order. We must be cognizant of this when using these members. (We say more about the difference between host byte order and network byte order in .
- The 32-bit IPV4 address can be accessed in two different ways. For example, if serv.is defined as an Internet socket address structure, then serv. Sin_addr references the 32 IPV4 address as an in_addr structure while serv. Sin_addr.s.addr references the same 32 bit IPV4 address as an in_addr_t (typically an unsigned 32-bit integer). We must be certain that we are referencing the IPV$ address correctly, especially when it is as an argument to a function , because compilers often pass structures differently from integers.
- The sin_zero member is unused, but we always set it to 0 when filling in one of these structures. By convention, we always set the entire structure to 0 before filling it in, not just the sin_zero member.
- Socket address structures are used only on a given host, the structure itself is not communicated between different hosts although certain fields (e.g., the IP address and port) are used for communication.

4.2 **Generic Socket Address Structure**

Socket address structures are always passed by reference when passed as an argument to any o the socket functions. But the socket functions that like one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.

```
Struct sockaddr    {
Uint8_t          sa_len;
Sa_family_t  sa_family  /* address family:AF_xxx value*/
Char           sa_data[14]; /*protocol –specific address
};
```

- The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the bind function.

```
int  bind(int, struct sockadd*/ soclen_t};
```

- This requires that any calls to these functions must cast the point to the protocol specific socket address structure to be a pointer to a generic socket address structure.

- For example,

Struct sockaddr_in ser; /* IPV4 socket address structure */

- /* fill in serv{}*/
- bind(sockfd, (struct sockaddr *) & serv, size of (serv)

4.3 IPV6 Socket Address Structure

The IPv6 socket is defined by including the <netinet/in. h>header

```
Struct in6_addr{
Uint8_t  s6_addr[16];   /* 128-bit IPV6 address*/
                              /* network byteordered*/
};
#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_adddr;     /*IPV6 address*/

                                   /*network byte ordered*/

};
```

- The SIN6_LEN constant must be deined if the system supports the length member for socket address structures.
- The IPv6 family is AF_INET6, whereas the IPv4 family is AF_INET
- 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. On some 64-bit processor, data access of 64-bit values is optimized if stored on a 64-bit boundary.

- The sin6_flowinfo member is divided into three fields. The low-order 24 bits are the flow label. The next 4 bit are the priority. The next 4 bits are reserved.

4.4 Comparison of Socket Address Structure

- The socket address structures all contain a 1-byt length field, that the family field also occupies 1 byte and that any field that must be at least some number of bit is exactly that number of bits.
- Two of the socket address structures are fixed length, while the Unix domain structure and the data link structure are variable length.
- To handle variable-length structures whenever we pass a pointer to a socket address structure as an argument to one of the socket functions, we pass its length as another argument.
- Fig. 1.2 shows the comparison of socket address structures.

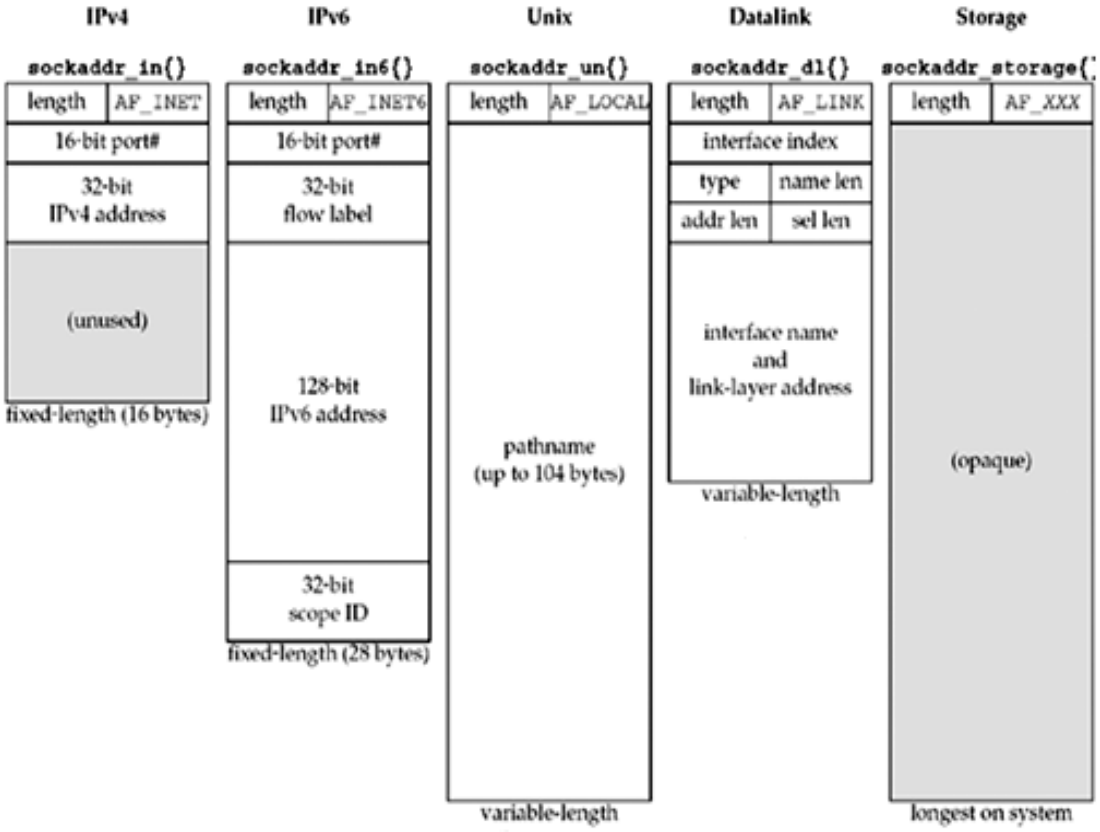


Fig. 1.2 Comparison of socket address structure

5 Byte ordering functions

A 16-bit integer that is made up of 2 bytes. There are two ways to store the two bytes in memory:

- with the low-order byte at the starting address, known as *little-endian* byte order
- with the high-order byte at the starting address, known as *big-endian* byte order.

Fig. 1.3 shows the increasing memory addresses going from right to left in the top, and from left to right in the bottom. The terms "little-endian" and "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.

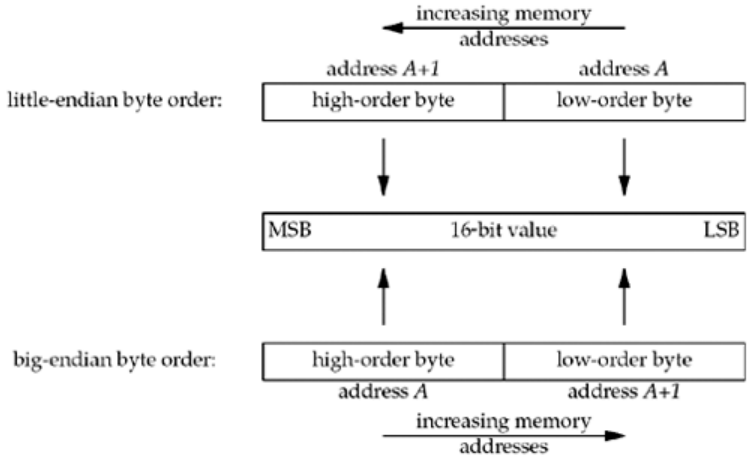


Fig. 1.3 Byte Ordering

- The implementation could store the fields in a socket address structure in host byte order and then convert to and from the network byte order when moving the fields to and from the protocol headers, saving us from having to worry about this detail.
- The following four functions used to convert between these two byte orders.

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

uint16_t ntohs(uint16_t *net16bitvalue*) ;
uint32_t ntohl(uint32_t *net32bitvalue*) ;
Both return: value in host byte order

- In the names of these functions, h stands for *host*, n stands for *network*, s stands for *short*, and lstands for *long*.
- Most Internet standards use the term *octet* instead of byte to mean an 8-bit quantity.

6. Address conversion functions

These functions convert IP address in ASCII dotted decimal format to binary format in network byte order and vice versa. The functions include,

- inet_aton
- inet_ntoa

Syntax

```
int inet_aton(const char *strptr, struct in_addr *addrptr);

Returns: 1 if string is valid (Successful), 0 on error
```

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

Returns: pointer to dotted-decimal string
```

7. Elementary TCP sockets

This section describes the necessary functions required to write a client server program. The Fig. 1.4 below shows the timeline of TCP client server communication.

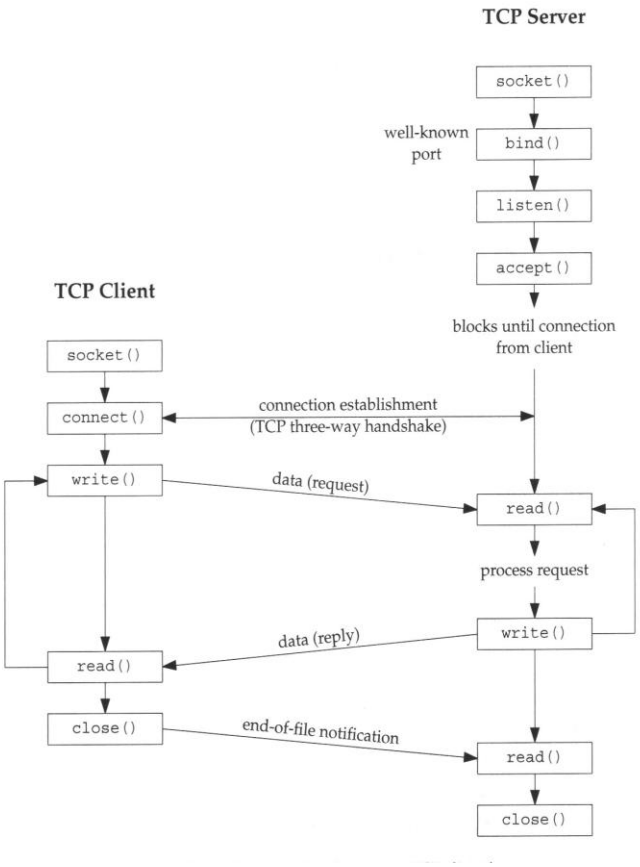


Fig. 1.4 Elementary TCP socket functions

7.1 Functions

7.1.1 Socket

To perform the input output operation in the network, first the process should call the socket function and specify the communication protocol as required.

Syntax

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

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

                    Returns : Non-negative descriptor if OK, -1 on error
```

The *family* field represents the protocol family and takes any one of the constant as shown in Table below.

family	Description
AF_INET	IPv4 protocol
AF_INET6	IPv6 protocol
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing socket
AF_KEY	Key socket

The socket type field represents the type of the socket used for communication and it takes any one of the constants as shown in Table below.

type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

The protocol field represents the type of the protocol used and it takes any one of the constants as shown in Table below. This field can be set a value 0 for default selection of the protocol for the given combination of *family* and *type.*

protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

7.1.2 Connect

This function is used by a TCP client to establish a connection with the server.

Syntax

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

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

                                        Returns : 0 if OK, -1 on error .
```

- The second argument is a pointer to the socket address structure. The socket address structure contains the IP address and the port number for the required connection.
- The third argument represents the size of the socket address structure.

The connect function initiates TCP's 3 way handshake. This function returns only when the connection is established or an error occurs. The possible error returns include the following.

- ❖ If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned.
- ❖ If the response from the server is reset (RST) for client's SYN, it indicates that no process is waiting for connections on server host at the specified port. This is a hard error and the error ECONNREFUSED is returned to the client.
- ❖ If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered as soft error. The kernel of the client saves the message but keeps sending SYN. If there is no response after some fixed amount of time, the saved ICMP error is returned to the process as EHOSTUNREACH or ENETUNREACH.

7.1.3 Bind

This function assigns a local protocol address to a socket.

Syntax

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

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

                                        Returns : 0 if OK, -1 on error
```

- The first argument *sockfd* is a socket descriptor returned by the socket function.
- The second argument is a pointer to the socket address structure. The socket address structure contains the IP address and port number of the local host.
- The third argument represents the size of the socket address structure.

Server binds the well known port when they start. If a client or server does not call the bind function, the kernel chooses an ephemeral port. Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP address when the socket is connected

based on the outgoing interface. If a TCP server does not bind an IP address, the kernel uses the destination IP address of the client's SYN as server source IP address.

7.1.4 Listen

The listen function is used only by the TCP server and it performs the following actions.

- When the socket is created is will be active. The listen function converts the unconnected socket into a passive socket, indicating that the kernel should accept the incoming connection requests to this socket.
- The second argument represents the maximum number of connections the kernel can queue for this socket.

Syntax

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

int listen (int sockfd, int backlog);

                                    Returns : 0 if OK, -1 on error
```

This function should be called after the socket and bind functions and before the accept function. The kernel maintains two queues for a listening socket.

- **Incomplete connection queue**

  It maintains an entry for each SYN that has arrived from a client for which the server is waiting for the completion of Three-way handshake.

- **Completed connection queue**

  It maintains an entry for each client which has completed the three-way handshake.The Fig. 1.5 depicts the two queues for a listening socket.

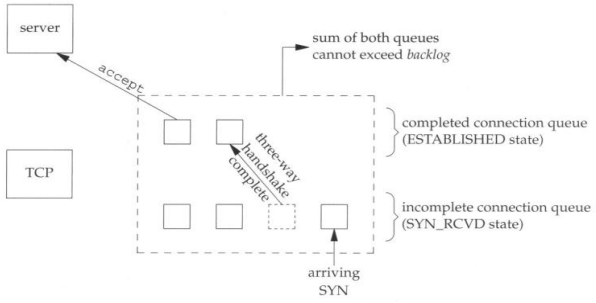


Fig. 1.5 Queues for a listening socket

Fig. 1.6 below shows the packet exchange during connection establishment with the two queues.

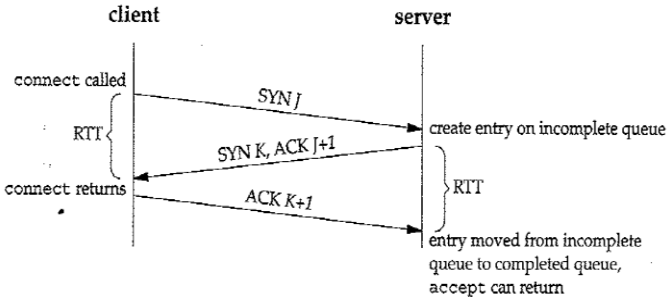


Fig. 1.6 Packet exchange in TCP

7.1.5 Accept

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

Syntax

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

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);

                                    Returns : non-negative descriptor if OK, -1 on error
```

- The first argument *sockfd* is a socket descriptor returned by the socket function.
- The second argument is a pointer to the socket address structure. The socket address structure contains the IP address and port number of the client.
- The third argument *addrlen* represents the size of the socket address structure pointed to by *cliaddr*.

This function returns upto three values. It returns an integer value which is either a new socket descriptor or an indication of an error, the address of the client and the size of this address. If the protocol address of client is not required, both *cliaddr* and *addrlen* are set to null pointers.

7.1.6 Fork

This function is used to create a new child process to handle each incoming client request. This is the only function in unix to create a child process.

Syntax

```
# include <unistd.h>

        pid_t fork(void);

                                Returns : 0 in child, process ID of child in parent, -1 on error
```

This function is called once but it returns twice once in the parent process and once in the child process. All the descriptors open in parent before calling fork() are shared with the child after fork returns.

The **uses** of fork function include the following.

- A process makes a copy of itself so that one copy can handle one operation while the other copy can do other tasks.
- After the creation of new process by using fork function, one of the process calls exec to replace itself with the new program.

7.1.7 Exec

In Unix, the only way to execute a program on disk is to call an exec function by any existing process. There are six exec functions that can be used.

Syntax

```
#include 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[]);
```

The Fig. 1.7 below shows the types of exec functions.

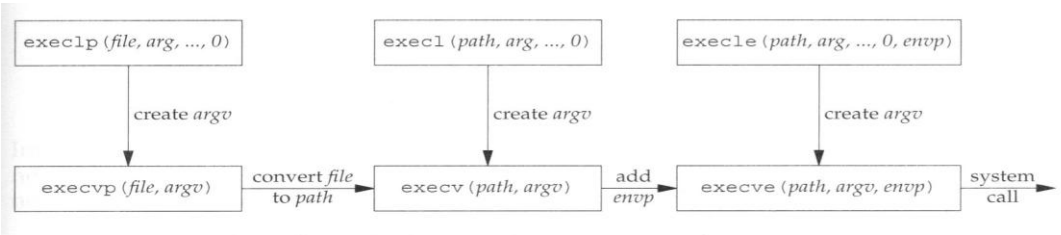


Fig. 1.7 Exec functions

7.1.8 Write

This function is used to send the data over stream sockets.

Syntax

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

int write (int sockfd, const void *msg, int len, int flags);

                                    Returns : No. of bytes send if OK, -1 on error
```

- The first argument *sockfd* is a socket descriptor returned by the socket function.
- The second argument is a pointer to the data to be send.
- The third argument is the length of the data to be send in terms of bytes.
- The fourth argument is flags which is set to 0.

7.1.9 Read

This function is used to receive the data from a stream socket.

Syntax

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

int read (int sockfd, void *buf, int len, int flags);

                                    Returns : No. of bytes send if OK, -1 on error
```

- The first argument *sockfd* is a socket descriptor returned by the socket function.
- The second argument is a pointer to the buffer to read the data.
- The third argument is the maximum length of the buffer.
- The fourth argument is flags which is set to 0.

7.1.10 Close

This function is used to terminate a TCP connection.

Syntax

```
# include <unistd.h>

        int close(int sockfd);

                                    Returns : 0 if ok, -1 on error
```

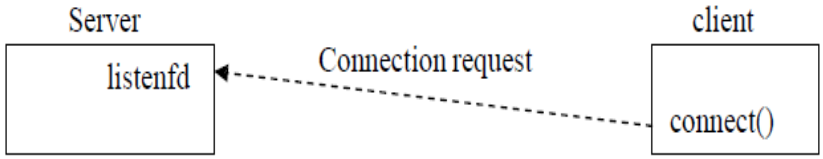
- The default operation of close function is to mark the socket as closed and return to the process immediately.

8. Concurrent Server

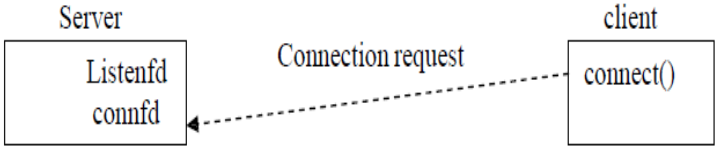
A concurrent server handles multiple requests from the clients at the same time. This is done by creating a child process for each client request using a function called fork().This The following program is a typical concurrent server.

```
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*/ }
```

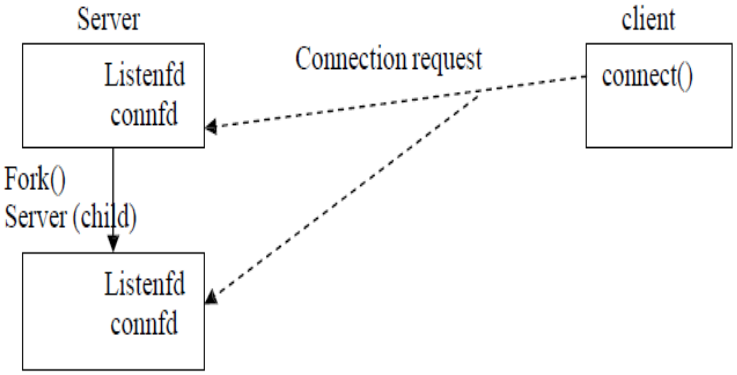
When a connection is established, accept returns, the server calls the fork() function and child process services the client and the parent process waits for further connection. The parent closes the connected socket since the child handles this new client. Fig. 1.8 shows the communication between client and the concurrent server.



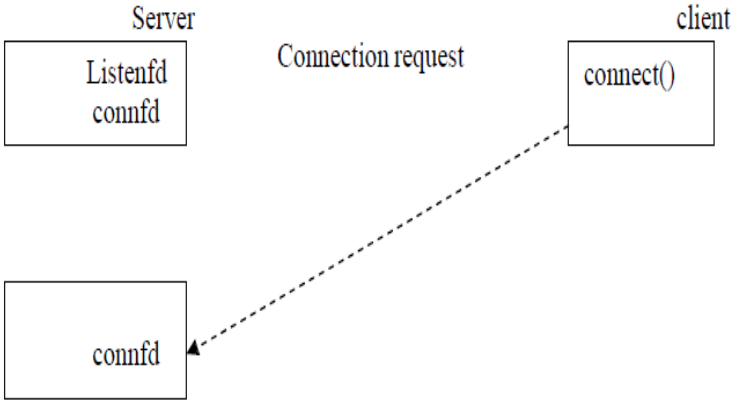
Status of client –server before call to accept().



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.

Fig. 1.8 Concurrent server communication

9. Iterative server

An iterative server handles a single request from the client at a time. This service can be used when the requests are guaranteed to be serviced within a small amount of time.

Problems:

- Server is locked while dealing with the request.
- If the request takes longer time, no other clients are serviced.

The program below shows example of an iterative day time server.

```
#include    "unp.h".
#include    <time.h>

int
main(int argc, char **argv)
{
    int     listenfd, connfd;
    struct sockaddr_in servaddr;
    char    buff[MAXLINE];
    time_t ticks;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzeros(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(13); /* daytime server */

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

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
       connfd = Accept(listenfd, (SA *) NULL, NULL);

       ticks = time(NULL);
       snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
       Write(connfd, buff, strlen(buff));

       Close(connfd);
    }
}
```

Questions

PART – A

1. Define sockets and list out its types.
2. What is the use of bind() system call?
3. Write the Internet Socket address structure.
4. List out the address transformation functions.
5. What is the use of connect() system call?
6. Name the functions that is used alone by the TCP server and give the syntax for the same.
7. List out the functions used by UDP to send and receive messages.
8. Define the byte order used by the TCP/IP protocol suite.
9. Differentiate iterative and concurrent server.
10. What is physical address and Internet address.

PART - B

1. Explain the TCP/IP layering in detail with a neat sketch
2. Explain the following system calls
   (a) Socket
   (b) Bind
   (c) Read
   (d) Write
   (e) Close
3. With a neat diagram explain the connectionless iterative server.
4. Briefly explain the connection oriented concurrent server with a neat diagram
5. Compare and contrast the concurrent server and iterative server
6. Give a brief note on the following:
   (a) Byte manipulation functions (6)
   (b) Byte order transformation functions (6)

APPLICATION DEVELOPMENT **10 hrs.**

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)

1. TCP Echo client server

A TCP client server communication involves the following steps.

1. The client reads the input data from the standard input and writes the line to the server.
2. The server reads the data from the network input and echoes the line back to client.
3. The client reads the echoed line and prints it on the standard output.

Fig 2.1 shows the typical echo client server.

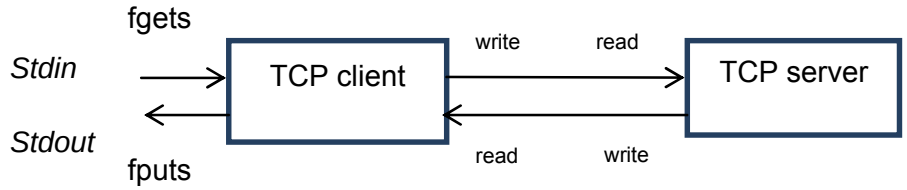


Fig. 2.1 TCP echo client server

1.1 TCP echo server

The role of the TCP echo server is written in two functions. One is the main() function and the other is str_echo() function.

TCP echo server main() function includes the following steps namely,

- Create socket
- Bind server's well known port
- Wait for client connection to complete
- Concurrent server

TCP echo server str_echo() function includes the following steps namely,

- Read a buffer

- Echo the buffer

Fig. 2.2 shows the TCP echo server main() function and Fig. 2.3 shows the TCP echo server str_echo() function.

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

Fig 2.2 TCP echo server main() function

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.

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

Fig 2.3 TCP echo server str_echo function

1.2 TCP echo client

The role of the TCP echo client is written in two functions. One is the main() function and the other is str_cli() function.

The TCP echo client main() function includes the following steps namely,

- Create socket
- Fill in the socket address structure

- Connect to server

The TCP echo client str_cli() function includes the following steps namely,

- Read a line, write to server
- Read echoed line from server, write to standard output
- Return to main

Fig. 2.4 shows the TCP echo client main() function and Fig. 2.5 shows the TCP echo client str_cli() function.

```
1  #include    "unp.h"
2  int
3  main(int argc, char **argv)
4  {
5      int     sockfd;
6      struct sockaddr_in servaddr;
7      if (argc != 2)
8          err_quit("usage: tcpcli <IPaddress>");
9      sockfd = Socket(AF_INET, SOCK_STREAM, 0);
10     bzero(&servaddr, sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_port = htons(SERV_PORT);
13     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
14     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
15     str_cli(stdin, sockfd);      /* do it all */
16     exit(0);
17 }
```

tcpcliserv/tcpcli01.

Fig. 2.4 TCP echo client main() function

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.

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.

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

Fig. 2.5 TCP echo client str_cli() function

*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.

2. POSIX signal handling

2.1 Signal – Definition

A signal is a notification to a process or within a process about the occurrence of an event.The receiving process can ignore a signal or can call a routine (handled by signal handler). After returning from signal handler, the receiving process will resume its execution at the point where it is interrupted.

Conditions for occurrence of a signal

1. Hardware exceptions
2. Process can send signals to themselves.

3. Kernel can generate and send signal to process when something happens. (eg) SIGPIPE will be generated when a process writes to a pipe which has been closed by the reader.

2.2 POSIX signal handling – Portable Operating System Interface for UNIX

• Every signal has a ***disposition***, which is called the ***action*** associated with the signal. The disposition is set by calling the ***sigaction*function**.

2.2.1 Choices for disposition

There are three choices for the disposition. These include the following.

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:

conststructsigaction 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.

- 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 behavior can be set.

• 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.

Fig. 2.6 shows the signal function that calls the POSIX sigaction function. Follwing is the description of each line.

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.

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

Fig. 2.6 signal function that calls the POSIX sigaction function

3. Server with multiple clients

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet may want to connect to it. You can use threads to handle the server's multiple clients simultaneously. Simply create a thread for each connection. Fig 2.7 shows a server that serves multiple clients.

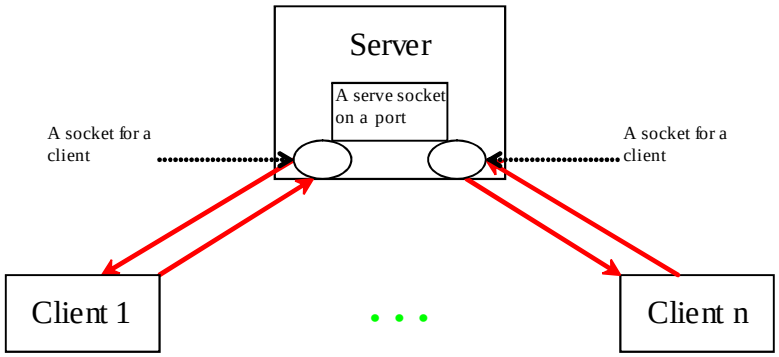


Fig. 2.7 A server that serves multiple clients

Zombies

- Zombie = a process that has terminated, but whose parent has not yet waited for it
- One way to see zombies is to press Ctrl-Z (suspend) in the midst of execution and then enter a ps command. Zombies appear as <defunct> processes.

Wait / Waitpid

Either of wait or waitpid can be used to remove zombies.

wait (and waitpid in it's blocking form) temporarily suspends the execution of a parent process while a child process is running. Once the child has finished, the waiting parent is restarted.

Declarations:

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

```
pid_t wait(int *statloc);
/* returns process ID if OK, or -1 on error */

pid_t waitpid(pid_t pid, int *statloc, int options);
/* returns process ID : if OK,
 *        0       : if non-blocking option && no zombies around
 *        -1      : on error
 */
```

The statloc argument can be one of two values:

- NULL pointer: the argument is simply ignored
- pointer to an integer: when wait returns, the integer this describes will contain status information of the terminated process (see Stevens p.198 for macros that examine the termination status)

wait()	**waitpid()**
wait blocks the caller until a child process terminates	waitpid can be either blocking or non-blocking: • If options is 0, then it is blocking • If options is WNOHANG, then is it non-blocking
if more than one child is running then wait() returns the first time one of the parent's offspring exits	waitpid is more flexible: • If pid == -1, it waits for any child process. In this respect, waitpid is equivalent to wait

	• If pid > 0, it waits for the child whose process ID equals pid • If pid == 0, it waits for any child whose process group ID equals that of the calling process • If pid < -1, it waits for any child whose process group ID equals that absolute value of pid

4. Boundary conditions

4.1 Connection abort before accept returns

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 acceptwhen the RST arrives. Sometime later the server process calls accept.Depending on the type of implementation the aborted connection differs.

- The Berkley derivedimplementation handles the aborted connection completely within the kernel and the server process neversees it.
- Most of the SVR4 (System V release 4) implementation returns an error to the process as the returnfrom **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. The reason for this is that EPROTO is also returned when some fatalprotocol related event occurs on the bytestream. Receiving the same error EPROTO by the server makes itdifficult to decide whether to call **accept** again or not. IN case of ECONNABORTED error, theserverignores the error and just calls accept again.

4.2 Termination of Server Process

After starting client and server, the child process is killed at the server (kill the child process basedon its ID). This simulates the crashing of the server process, then what happens to client: Thefollowing steps take place.

1. We start the server and client on different hosts and type one line to the client to verify that all isOK. 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 opendescriptors in the child are closed. This causes the FIN to bysent to the client and the client TCPresponds 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 withan ACK. But the problem is that the client process is blocked in the call to the **fgets**waiting for aline 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 theserver process has closed its end of the connection and will not send any more data. The receipt ofFIN 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 processthat had that socket open has terminated. We can verify that the RST is sent by watching thepackets with **tcpdump.**

7. But the client process will not see the RST because it calls **readline**immediately after the call to**write** and **readline**returns 0. Our client is not expecting to receive an end of line at thispoint 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 thesocket. The client is really working with the two descriptors - the socket and the user input – andinstead of blocking on input from any one of the two sources, it should block on input from eithersource. 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 ofthis signal is to terminate the process so the process must catch the signal to avoid involuntarilyterminated.If the process catches the signal and returns from the signal handler, or ignores the signal, the writeoperation returns EPIPE (error pipe)

```
#include <unp.h>
voidstr_cli (FILE *fp, intsockfd)
{
charsendline[MAXLINE], recvline[MAXLINE];
while (fgets(sendline, MAXLINE, fp)!=null)
{
writen(sockfd, sendline, 1);
sleep(1);
writeln (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 writeln is called two times: the first time the first byte of data is writeln tothe socket, followed by a pause of 1 sec, followed by the remainder of the line. The intention is for the firstwriteln to elicit the RST and then for the second writeln to generate SIGPIPE.We start with the client, type in one line, see that line is echoed correctly, and then terminates theserver child on the server host, we then type another line, but nothing is echoed and we just get a shellprompt. 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 thisoccurs. IF there is nothing special to do, then setting the signal disposition to SIG_IGN is easy, assumingthat 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 anydesired 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 theerror. IF we need to know which write caused the error, then we must either ignore the signal or return fromthe signal handler and handle **EPIPE** from **write.**

4.3 Crashing of Server Host

This scenario lets us know what happens when the server host crashes. To simulate this we must run theclient and server on different hosts. We then start server, start the client, type in a line to the client toverify that the connection is up, disconnect the server host from the network, andtype in another line at the client. This also covers the scenario of the server host being unreachablewhen 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 iswe 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 thedata segment, trying to receive ACK from the server. Berkley derived implementations transmitthe date segments 12 times, waiting around 9 minutes before giving up. When the client finallygives 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 responsesat all to the client‘s data segments, the error is ETIMEDOUT. But if some intermediate routerdetermine that the server was unreachable and responded with ICMP destination unreachablemessage, 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 usedwhich used SO_KEEPALIVE socket option

4.4 Crashing and Rebooting of Server Host

In the following example, we will establish a connection between the client and server and then assume the server host crashes and reboots. The easiest way to simulate this is to establish the connection, disconnect the server from the network, shut down the server host and then reboot it, and then reconnect the server host to the network. We do not want the client to see the server host shut down.

As stated in the previous section, if the client is not actively sending data to the server when the server host crashes, the client is not aware that the server host has crashed. The following steps take place:

1. We start the server and then the client. We type a line to verify that the connection is established.
2. The server host crashes and reboots.
3. We type a line of input to the client, which is sent as a TCP data segment to the server host.
4. When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore, the server TCP responds to the received data segment from the client with an RST.
5. Our client is blocked in the call to readline when the RST is received, causing readline to return the error ECONNRESET.

If it is important for our client to detect the crashing of the server host, even if the client is not actively sending data, then some other technique, such as the SO_KEEPALIVE socket option or some client/server heartbeat function, is required.

4.5 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 20seconds), 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 bySIGKILL signal.When the process terminates, all the open descriptors are closed, and we then follow thesame sequence

of steps discussed under termination of server processWe need to select the select or poll function in the client to have the client detect thetermination of the server process as soon it occurs.

5. I/O Multiplexing

- TCP echo client is handling two inputs at the same time: standard input and a TCP socket
  - when the client was blocked in a call to read, the server process was killed
  - server TCP sends FIN to the client TCP, but the client never sees FIN since the client is blocked reading from standard input
    - ✓ We need the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready.
    - ✓ I/O multiplexing (select, poll, or newer pselect functions)
- Scenarios for I/O Multiplexing
  - client is handling multiple descriptors (interactive input and a network socket).
  - Client to handle multiple sockets (rare)
  - TCP server handles both a listening socket and its connected socket.
  - Server handle both TCP and UDP.
  - Server handles multiple services and multiple protocols

6. I/O Models

The five I/O models available under UNIX:

- blocking I/O
- nonblocking I/O
- I/O multiplexing (select and poll)
- signal driven I/O (SIGIO)
- asynchronous I/O (the POSIX aio_functions)

Two distinct phases for an input operation

- Waiting for the data to be ready (for a socket, wait for the data to arrive on the network, then copy into a buffer within the kernel)
- Copying the data from the kernel to the process (from kernel buffer into application buffer)

Categories

- Synchronous I/O
  - causes the requesting process to be blocked until that I/O operation (recvfrom) completes. (blocking, nonblocking, I/O multiplexing, signal-driven I/O)
- Asynchronous I/O
  - does not cause the requesting process to be blocked

5.1 Blocking I/O model

The most prevalent model for I/O is the blocking I/O model. By default, all sockets are blocking. Fig. 2.8 shows the blocking I/O model.

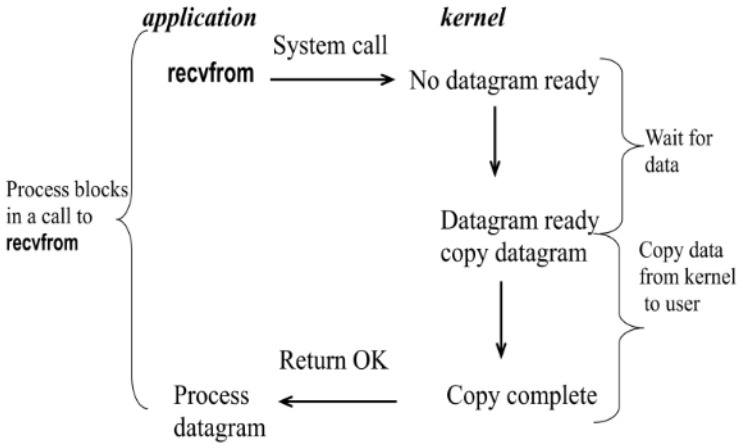


Fig. 2.8 Blocking I/O Model

UDP is used in this example instead of TCP because with UDP, the concept of data being "ready" to read is simple: either an entire datagram has been received or it has not. With TCP it gets more complicated, as additional variables such as the socket's low-water mark come into play.

In fig. 2.8, 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, The process is blocked the entire time from when it calls recvfrom until it returns. When recvfrom returns successfully, the application processes the datagram.

5.2 Nonblocking I/O model

When a socket is set to be nonblocking, when an I/O operation requested cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead. Fig. 2.9 shows the non-blocking I/O model.

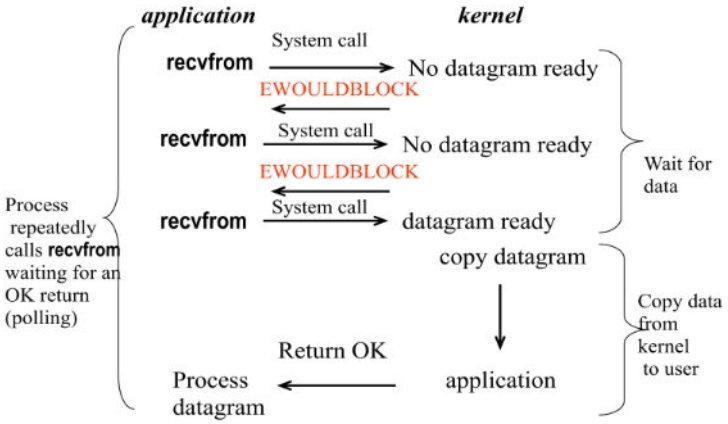


Fig. 2.9 Non-blocking I/O model

- For the first three recvfrom, there is no data to return and the kernel immediately returns an error of EWOULDBLOCK.
- For the fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. The data is further processed.

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

5.3 I/O Multiplexing

With **I/O multiplexing**, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call. Fig. 2.10 is a summary of the I/O multiplexing model.

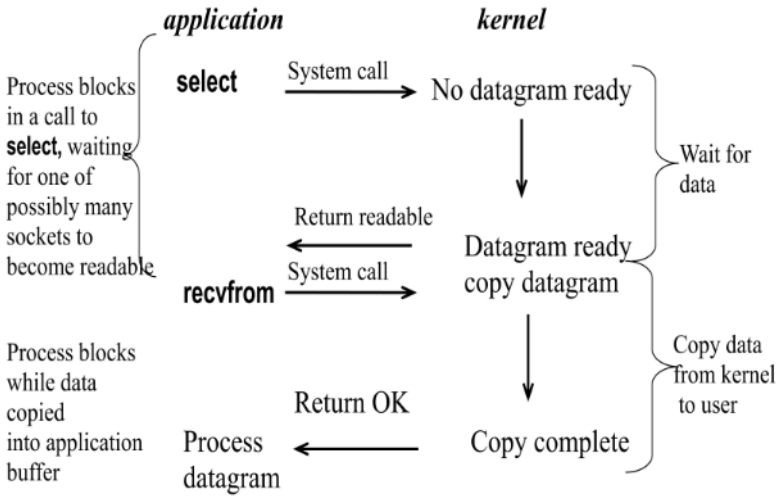


Fig. 2.10 I/O Multiplexing

We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

Comparing to the blocking I/O model

- Disadvantage: using select requires two system calls (select and recvfrom) instead of one
- Advantage: we can wait for more than one descriptor to be ready.

Multithreading with blocking I/O

Another closely related I/O model is to use multithreading with blocking I/O. That model very closely resembles the model described above, except that instead of using select to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like recvfrom.

5.4 Signal-driven I/O

The **signal-driven I/O model** uses signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready. Fig. 2.11 shows the signal driven I/O model.

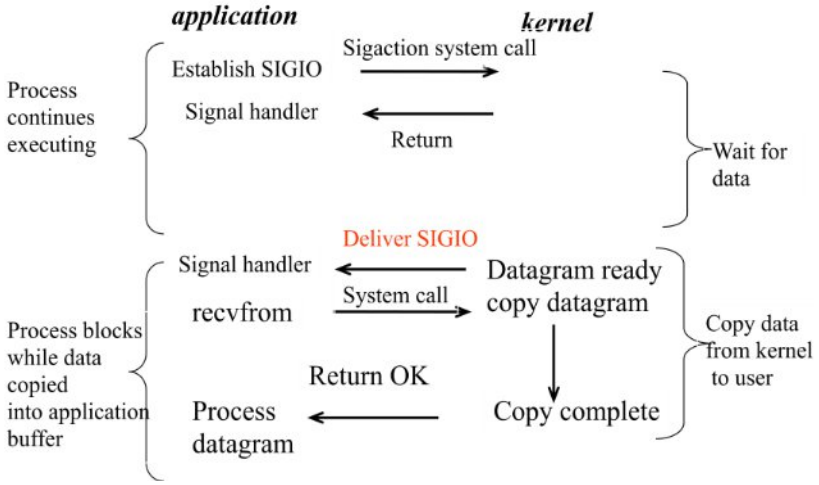


Fig. 2.11 Signal driven I/O model

- We first enable the socket for 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 continues; it is not blocked.
- When the datagram is ready to be read, 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 processed.
  - notify the main loop and let it read the datagram.

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

5.5 Asynchronous I/O model

Asynchronous I/O is defined by the POSIX specification, and various differences in the real-time functions that appeared in the various standards which came together to form the current POSIX specification have been reconciled.

These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete. The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete. Fig. 2.12 shows the asynchronous I/O model.

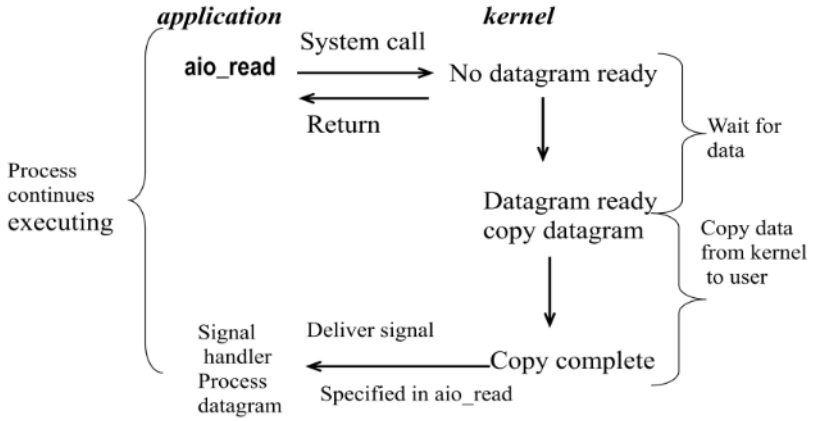


Fig. 2.12 Asynchronous I/O model

- We call aio_read (the POSIX asynchronous I/O functions begin with aio_ or lio_) and pass the kernel the following:
  - descriptor, buffer pointer, buffer size (the same three arguments for read),
  - file offset (similar to lseek),
  - and how to notify us when the entire operation is complete.

  This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

Comparison of I/O Models

Fig. 2.13 shows the comparison of I/O models.

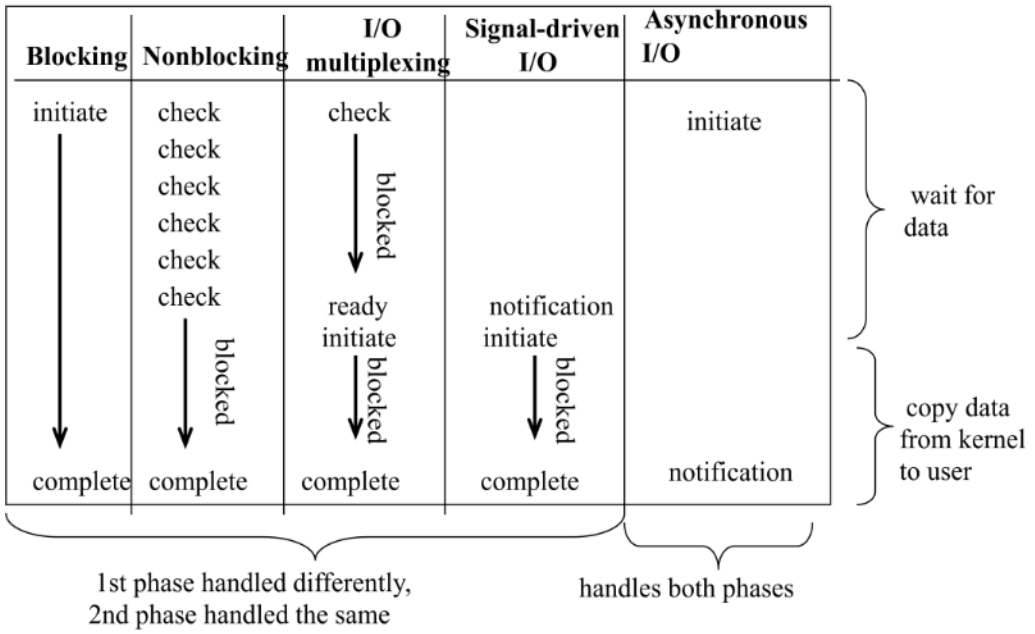


Fig. 2.13 Comparison of I/O models

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

6. SELECT FUNCTION

- Allows the process to instruct the kernel to *wait for any one of multiple events to occur* and to wake up the process only when one or more of these events occurs or *when a specified amount of time has passed*.
- What descriptors we are interested in (readable ,writable , or exception condition) and how long to wait?

6.1 Possibilities for select function

- Wait forever : return only when descriptor (s) is ready (specify **timeout** argument as NULL)
- wait up to a fixed amount of time
- Do not wait at all : return immediately after checking the descriptors. Polling (specify **timeout** argument as pointing to a **timeval** structure where the timer value is 0)
- The wait is normally interrupted if the process catches a signal and returns from the signal handler
    - **select** might return an error of **EINTR**
    - Actual return value from function = -1

6.2 Syntax

```
#include <sys/select.h>
#include <sys/time.h>

int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *);
//Returns: +ve count of ready descriptors, 0 on timeout, -1 on error

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

select function Descriptor Arguments

- **readset** → descriptors for checking readable
- **writeset** → descriptors for checking writable
- **exceptset** → descriptors for checking exception conditions (2 exception conditions)
    - ✓ arrival of out of band data for a socket

- ✓ the presence of control status information to be read from the master side of a pseudo terminal (Ignore)

- If you pass the 3 arguments as NULL, you have a high precision timer than the sleep function

Descriptor Sets

- Array of integers : each bit in each integer correspond to a descriptor (**fd_set**)
- 4 macros
  - void FD_ZERO(fd_set *fdset); /* clear all bits in fdset */
  - void FD_SET(int fd, fd_set *fdset); /* turn on the bit for fd in fdset */
  - void FD_CLR(int fd, fd_set *fdset); /* turn off the bit for fd in fdset*/
  - int FD_ISSET(int fd, fd_set *fdset);/* is the bit for fd on in fdset ? */

Example of Descriptor sets Macros

```
fd_set  rset;

FD_ZERO(&rset);       /*all bits off : initiate*/

FD_SET(1, &rset);     /*turn on bit fd 1*/

FD_SET(4, &rset);     /*turn on bit fd 4*/

FD_SET(5, &rset);     /*turn on bit fd 5*/
```

The *maxfdp1* argument

- The *maxfdp1* argument specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested plus one. The descriptors 0, 1, 2, up 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 datatype. Its value is often 1024, but few programs use that many descriptors.
- The reason the *maxfdp1* argument exists, along with the burden of calculating its value, is for efficiency. Although each fd_set has room for many descriptors, typically 1,024, this is much more than the number used by a typical process. The kernel gains efficiency by not copying unneeded portions of the descriptor set between the process and the kernel, and by not testing bits that are always 0.

***readset*, *writeset*, and *exceptset* as value-result arguments**

- select modifies the descriptor sets pointed to by the *readset*, *writeset*, and *exceptset* 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 the FD_ISSET macro on return to test a specific descriptor in an fd_setstructure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set. To handle this, we turn on all the bits in which we are interested in all the descriptor sets each time we call select.

Return value of select

The return value from this function indicates the total number of bits that are ready across all the descriptor sets. If the timer value expires before any of the descriptors are ready, a value of 0 is returned. A return value of –1 indicates an error (which can happen, for example, if the function is interrupted by a caught signal).

Conditions for a Ready Descriptor

The following are specific about the conditions that cause select to return "ready" for sockets

1. **A socket is ready for reading** if any of the following four conditions is true:
    - The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer. A read operation on the socket will not block and will return a value greater than 0 (i.e.,

the data that is ready to be read). We can set this low-water mark using the SO_RCVLOWAT socket option. It defaults to 1 for TCP and UDP sockets.

- The read half of the connection is closed (i.e., a TCP connection that has received a FIN). A read operation on the socket will not block and will return 0 (i.e., EOF).
- The socket is a listening socket and the number of completed connections is nonzero.
- A socket error is pending. A read operation on the socket will not block and will return an error (–1) with errno set to the specific error condition. These pending errors can also be fetched and cleared by calling getsockopt and specifying the SO_ERROR socket option.

2. **A socket is ready for writing** if any of the following four conditions is true:
   - The number of bytes of available space in the socket send buffer is greater than or equal to the current size of the low-water mark for the socket send buffer and either: (i) the socket is connected, or (ii) the socket does not require a connection (e.g., UDP). This means that if we set the socket to nonblocking (Chapter 16), a write operation will not block and will return a positive value (e.g., the number of bytes accepted by the transport layer). We can set this low-water mark using the SO_SNDLOWAT socket option. This low-water mark normally defaults to 2048 for TCP and UDP sockets.
   - The write half of the connection is closed. A write operation on the socket will generate SIGPIPE (Section 5.12).
   - A socket using a non-blocking connect has completed the connection, or the connect has failed.
   - A socket error is pending. A write operation on the socket will not block and will return an error (–1) with errno set to the specific error condition. These pending errors can also be fetched and cleared by calling getsockopt with theSO_ERROR socket option.
3. A socket has an exception condition pending if there is out-of-band data for the socket or the socket is still at the out-of-band mark.

pselect Function

The pselect function was invented by POSIX and is now supported by many of the Unix variants.

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

```
#include <signal.h>
```

```
#include <time.h>
```

```
int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
        const struct timespec *timeout, const sigset_t *sigmask);
```

```
/* Returns: count of ready descriptors, 0 on timeout, –1 on error */
```

pselect contains two changes from the normal select function:

pselect uses the timespec structure (another POSIX invention) instead of the timeval structure. The tv_nsec member of the newer structure specifies nanoseconds, whereas the tv_usec member of the older structure specifies microseconds.

```
struct timespec {
  time_t tv_sec;       /* seconds */
  long   tv_nsec;      /* nanoseconds */
};
```

pselect adds a sixth argument: a pointer to a signal mask. This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now-disabled signals, and then call pselect, telling it to reset the signal mask.

With regard to the second point, consider the following example (discussed on APUE). Our program's signal handler for SIGINT just sets the global intr_flag and returns. If our process is blocked in a call to select, the return from the signal handler causes the function to return with errno set to EINTR. But when select is called, the code looks like the following:

```
if (intr_flag)
    handle_intr();       /* handle the signal */


/* signals occurring in here are lost */


if ( (nready = select( ... )) < 0) {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

The problem is that between the test of intr_flag and the call to select, if the signal occurs, it will be lost if select blocks forever.

With pselect, we can now code this example reliably as:

```
sigset_t newmask, oldmask, zeromask;
```

```
sigemptyset(&zeromask);

sigemptyset(&newmask);

sigaddset(&newmask, SIGINT);



sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* block SIGINT */

if (intr_flag)

   handle_intr();     /* handle the signal */

if ( (nready = pselect ( ... , &zeromask)) < 0) {

   if (errno == EINTR)  {

      if (intr_flag)

         handle_intr ();

   }

   ...

}
```

Before testing the intr_flag variable, we block SIGINT. When pselect is called, it replaces the signal mask of the process with an empty set (i.e., zeromask) and then checks the descriptors, possibly going to sleep. But when pselect returns, the signal mask of the process is reset to its value before pselect was called (i.e., SIGINT is blocked).

7. SHUTDOWN function

- Close one half of the TCP connection
    - send FIN to server, but leave the socket descriptor open for reading

- Limitations with close function
  - decrements the descriptor's reference count and closes the socket only if the count reaches 0
    - ✓ With shutdown, can initiate TCP normal connection termination regardless of the reference count
  - terminates both directions (reading and writing)
    - ✓ With shutdown, we can tell other end that we are done sending, although that end might have more data to send us.

Figure 2.14 shows the process of shutdown function.

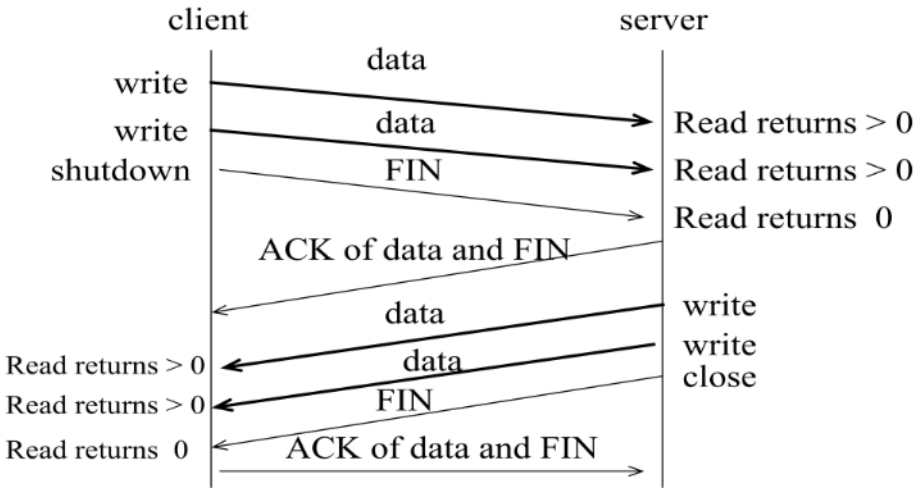


Fig. 2.14 Shutdown function

7.1 Syntax

#include<sys/socket.h>

int shutdown (int sockfd, int howto);

/* return : 0 if OK, -1 on error */

- ***howto*** argument

- **SHUT_RD**
  - ✓ read-half of the connection closed
  - ✓ Any data in receive buffer is discarded
  - ✓ Any data received after this call is ACKed and then discarded
- **SHUT_WR**
  - ✓ write-half of the connection closed (half-close)
  - ✓ Data in socket send buffer sent, followed by connection termination
- **SHUT_RDWR**
  - ✓ both closed

8. TCP Echo Server (with multiplexing)

The TCP echo server as a single process that uses select to handle any number of clients, instead of forking one child per client.

Before first client has established a connection

Before the first client has established a connection, the server has a single listening descriptor as shown in fig. 2.15.

- The server maintains only a read descriptor set (*rset*), shown in the following figure. Assuming the server is started in the foreground, descriptors 0, 1, and 2 are set to standard input, output, and error, so the first available descriptor for the listening socket is 3.
- We also show an array of integers named client that contains the connected socket descriptor for each client. All elements in this array are initialized to –1.

Fig. 2.15 First client establishing the connection

The only nonzero entry in the descriptor set is the entry for the listening sockets and the first argument to select will be 4.

After first client establishes connection

When the first client establishes a connection with our server, the listening descriptor becomes readable and our server calls accept. The new connected descriptor returned by accept will be 4. Fig. 2.16 shows this connection:

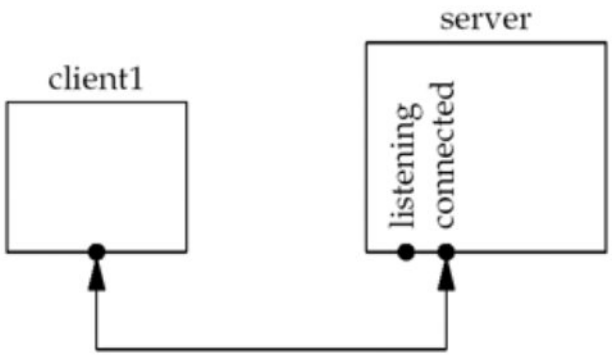


Fig. 2.16 After client establishment of connection

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 structures are shown in the fig. 2.17.

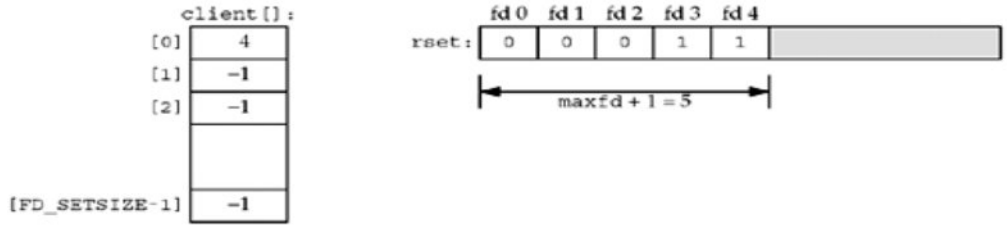


Fig. 2.17 Updated data structure

After second client connection is established

Sometime later a second client establishes a connection and we have the scenario shown below as in fig. 2.18.

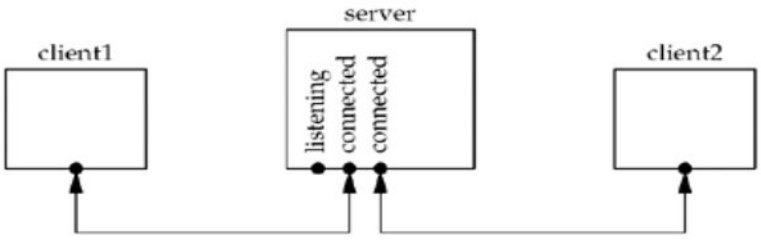


Fig. 2.18 After second client establishment

The new connected socket (which we assume is 5) must be remembered, giving the data structures shown below as in fig. 2.19.

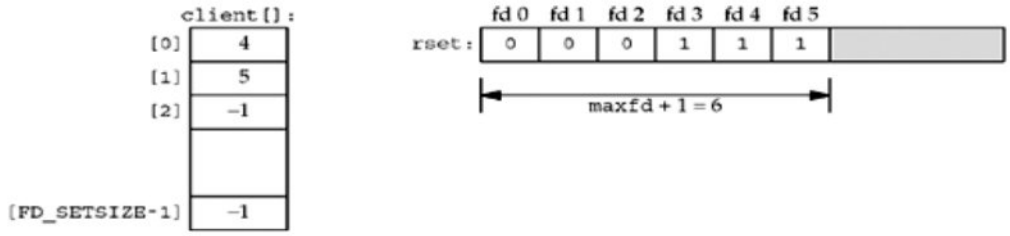


Fig. 2.19 Updated data structure

After first client terminates its connection

The first client terminates its connection. The client TCP sends a FIN, which makes descriptor 4 in the server readable. When our server reads this connected socket, read returns 0. We then close this socket and update our data structures accordingly. The value of client[0] is set to –1 and descriptor 4 in the descriptor set is set to 0. This is shown in the figure below. Notice that the value of maxfd does not change.

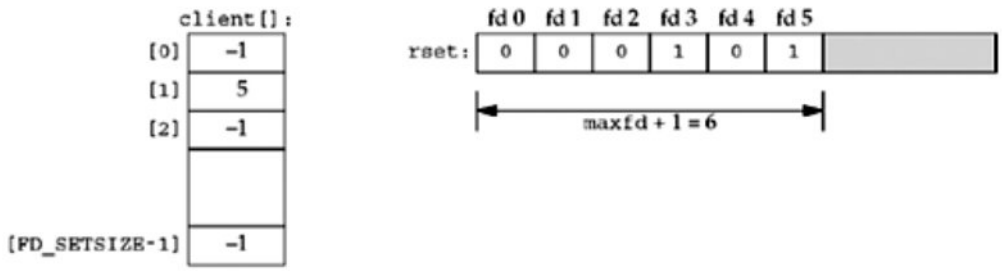


Fig. 2.20 First client terminating the connection

Summary of TCP echo server (revisited)

- As clients arrive, we record their connected socket descriptor in the first available entry in the client array (the first entry with a value of –1) and also add the connected socket to the read descriptor set.
- The variable maxi is the highest index in the client array that is currently in use and the variable maxfd (plus one) is the current value of the first argument to select.
- The only limit on the number of clients that this server can handle is the minimum of the two values FD_SETSIZE and the maximum number of descriptors allowed for this process by the kernel

```
/* include fig01 */
#include    "unp.h"

int
main(int argc, char **argv)
```

```
{
    int             i, maxi, maxfd, listenfd, connfd, sockfd;
    int             nready, client[FD_SETSIZE];
    ssize_t         n;
    fd_set          rset, allset;
    char            buf[MAXLINE];
    socklen_t       clilen;
    struct sockaddr_in  cliaddr, servaddr;

    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);

    maxfd = listenfd;           /* initialize */
    maxi = -1;                  /* index into client[] array */
    for (i = 0; i < FD_SETSIZE; i++)
        client[i] = -1;         /* -1 indicates available entry */
    FD_ZERO(&allset);
    FD_SET(listenfd, &allset);
/* end fig01 */

/* include fig02 */
    for ( ; ; ) {
        rset = allset;      /* structure assignment */
```

```
        nready = Select(maxfd+1, &rset, NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &rset)) {    /* new client connection */
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
    #ifdef  NOTDEF
            printf("new client: %s, port %d\n",
                    Inet_ntop(AF_INET, &cliaddr.sin_addr, 4, NULL),
                    ntohs(cliaddr.sin_port));
    #endif

            for (i = 0; i < FD_SETSIZE; i++)
                if (client[i] < 0) {
                    client[i] = connfd; /* save descriptor */
                    break;
                }
            if (i == FD_SETSIZE)
                err_quit("too many clients");

            FD_SET(connfd, &allset);    /* add new descriptor to set */
            if (connfd > maxfd)
                maxfd = connfd;         /* for select */
            if (i > maxi)
                maxi = i;               /* max index in client[] array */

            if (--nready <= 0)
                continue;               /* no more readable descriptors */
        }

        for (i = 0; i <= maxi; i++) {   /* check all clients for data */
            if ( (sockfd = client[i]) < 0)
```

```
            continue;
        if (FD_ISSET(sockfd, &rset)) {
            if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
                /* connection closed by client */
                Close(sockfd);
                FD_CLR(sockfd, &allset);
                client[i] = -1;
            } else
                Writen(sockfd, buf, n);

            if (--nready <= 0)
                break;          /* no more readable descriptors */
        }
    }
  }
}
```

The code does the following:

- **Create listening socket and initialize for select.** We create the listening socket using socket, bind, and listen and initialize our data structures assuming that the only descriptor that we will select on initially is the listening socket.
- **Block in select**. select waits for something to happen, which is one of the following:
  - The establishment of a new client connection.
  - The arrival of data on the existing connection.
  - A FIN on the existing connection.
  - A RST on the existing connection.
- **accept new connections**.
  - If the listening socket is readable, a new connection has been established.
  - We call accept and update our data structures accordingly. We use the first unused entry in the client array to record the connected socket.

- The number of ready descriptors is decremented, and if it is 0, we can avoid the next for loop. This lets us use the return value from select to avoid checking descriptors that are not ready.

- **Check existing connections**.
  - In the second nested for loop, a test is made for each existing client connection as to whether or not its descriptor is in the descriptor set returned by select, and a line is read from the client and echoed back to the client. Otherwsie, if the client closes the connection, read returns 0 and we update our data structures accordingly.
  - We never decrement the value of maxi, but we could check for this possibility each time a client closes its connection.

This server is more complicated than the earlier version, but it avoids all the overhead of creating a new process for each client and it is a nice example of select. Nevertheless, in, we will describe a problem with this server that is easily fixed by making the listening socket nonblocking and then checking for, and ignoring, a few errors from accept.

Denial-of-Service Attacks

There is a problem with the server in the above example. If a malicious client connects to the server, sends one byte of data (other than a newline), and then goes to sleep. The server will call read, which will read the single byte of data from the client and then block in the next call to read, waiting for more data from this client. The server is then blocked ("hung") by this one client and will not service any other clients, until the malicious client either sends a newline or terminates.

The basic concept here is that when a server is handling multiple clients, the server can never block in a function call related to a single client. Doing so can hang the server and deny service to all other clients. This is called a **denial-of-service** attack, which prevents the server from servicing other legitimate clients.

Possible solutions are:

- Use nonblocking I/O
- Have each client serviced by a separate thread of control (either spawn a process or a thread to service each client)
- Place a timeout on the I/O operations

9. Poll Function

Poll provides functionality that is similar to select, but poll provides additional information when dealing with STREAMS devices.

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

int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);

/* Returns: count of ready descriptors, 0 on timeout, –1 on error */

Arguments:

The first argument (*fdarray*) is a pointer to the first element of an array of structures. Each element is a pollfd structure that specifies the conditions to be tested for a given descriptor, fd.

```
struct pollfd {
  int     fd;       /* descriptor to check */
  short   events;   /* events of interest on fd */
  short   revents;  /* events that occurred on fd */
};
```

The conditions to be tested are specified by the events member, and the function returns the status for that descriptor in the corresponding revents member. This data structure (having two variables per descriptor, one a value and one a result) avoids value-result arguments (the middle three arguments for select are value-result). Each of these two members is composed of one or more bits that specify a certain condition. The following figure shows the constants used to specify the events flag and to test the revents flag against.

Constant	Input to *events* ?	Result from *revents* ?	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

The first four constants deal with input, the next three deal with output, and the final three deal with errors. The final three cannot be set in events, but are always returned in revents when the corresponding condition exists.

With regard to TCP and UDP sockets, the following conditions cause poll to return the specified revent. Unfortunately, POSIX leaves many holes (optional ways to return the same condition) in its definition of poll.

- All regular TCP data and all UDP data is considered normal.
- TCP's out-of-band data is considered priority band.
- When the read half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.
- The presence of an error for a TCP connection can be considered either normal data or an error (POLLERR). In either case, a subsequent read will return –1 with errno set to the appropriate value. This handles conditions such as the receipt of an RST or a timeout.
- The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.
- The completion of a nonblocking connect is considered to make a socket writable.

The number of elements in the array of structures is specified by the *nfds* argument.

The *timeout* argument specifies how long the function is to wait before returning. A positive value specifies the number of milliseconds to wait. The constant INFTIM (wait forever) is defined to be a negative value.

Return values from poll:

- –1 if an error occurred
- 0 if no descriptors are ready before the timer expires
- Otherwise, it is the number of descriptors that have a nonzero revents member.

If we are no longer interested in a particular descriptor, we just set the fd member of the pollfd structure to a negative value. Then the events member is ignored and the revents member is set to 0 on return.

10. TCP Echo Server (Revisited Again)

In the select version we allocate a client array along with a descriptor set named rset). With poll, we must allocate an array of pollfd structures to maintain the client information instead of allocating another array. We handle the fd member of this array the same way we handled the client array in the selection version: a value of –1 means the entry is not in use; otherwise, it is the descriptor value. Any entry in the array of pollfd structures passed to poll with a negative value for the fd member is just ignored.

```
/* include fig01 */
#include    "unp.h"
#include    <limits.h>      /* for OPEN_MAX */

int
main(int argc, char **argv)
{
   int             i, maxi, listenfd, connfd, sockfd;
   int             nready;
   ssize_t           n;
```

```
    char                buf[MAXLINE];
    socklen_t           clilen;
    struct pollfd       client[OPEN_MAX];
    struct sockaddr_in  cliaddr, servaddr;

    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);

    client[0].fd = listenfd;
    client[0].events = POLLRDNORM;
    for (i = 1; i < OPEN_MAX; i++)
        client[i].fd = -1;      /* -1 indicates available entry */
    maxi = 0;                   /* max index into client[] array */
/* end fig01 */

/* include fig02 */
    for ( ; ; ) {
        nready = Poll(client, maxi+1, INFTIM);

        if (client[0].revents & POLLRDNORM) {   /* new client connection */
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
#ifdef  NOTDEF
```

```
        printf("new client: %s\n", Sock_ntop((SA *) &cliaddr, clilen));
#endif

        for (i = 1; i < OPEN_MAX; i++)
            if (client[i].fd < 0) {
                client[i].fd = connfd;  /* save descriptor */
                break;
            }
        if (i == OPEN_MAX)
            err_quit("too many clients");

        client[i].events = POLLRDNORM;
        if (i > maxi)
            maxi = i;               /* max index in client[] array */

        if (--nready <= 0)
            continue;               /* no more readable descriptors */
    }

    for (i = 1; i <= maxi; i++) {   /* check all clients for data */
        if ( (sockfd = client[i].fd) < 0)
            continue;
        if (client[i].revents & (POLLRDNORM | POLLERR)) {
            if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
                if (errno == ECONNRESET) {
                        /* connection reset by client */
#ifdef  NOTDEF
                    printf("client[%d] aborted connection\n", i);
#endif
                    Close(sockfd);
                    client[i].fd = -1;
```

```
                } else
                    err_sys("read error");
            } else if (n == 0) {
                    /* connection closed by client */
#ifdef  NOTDEF
                printf("client[%d] closed connection\n", i);
#endif
                Close(sockfd);
                client[i].fd = -1;
            } else
                Writen(sockfd, buf, n);

            if (--nready <= 0)
                break;              /* no more readable descriptors */
        }
    }
  }
}
```

This code does the following:

- **Allocate array of pollfd structures.** We declare OPEN_MAX elements in our array of pollfd structures. Determining the maximum number of descriptors that a process can have open at any one time is difficult. One way is to call the POSIX sysconf function with an argument of _SC_OPEN_MAX (as described in APUE) and then dynamically allocate an array of the appropriate size.
- **Initialize.** We use the first entry in the client array for the listening socket and set the descriptor for the remaining entries to –1. We also set the POLLRDNORM event for this descriptor, to be notified by poll when a new connection is ready to be accepted. The variable maxi contains the largest index of the client array currently in use.

- **Call poll, check for new connection.** We call poll to wait for either a new connection or data on existing connection.
    - When a new connection is accepted, we find the first available entry in the client array by looking for the first one with a negative descriptor.
    - We start the search with the index of 1, since client[0] is used for the listening socket.
    - When an available entry is found, we save the descriptor and set the POLLRDNORM event.
- **Check for data on an existing connection.** The two return events that we check for are POLLRDNORM and POLLERR. We did not set POLLERR in the events member because it is always returned when the condition is true. The reason we check for POLLERR is because some implementations return this event when an RST is received for a connection, while others just return POLLRDNORM. In either case, we call read and if an error has occurred, it will return an error. When an existing connection is terminated by the client, we just set the fd member to –1.

11. TCP echo client (with multiplexing)

str_cli Function (Revisited Again)

The following code is our revised and correct version of the str_cli function that uses select and shutdown. In the function,select notifies us as soon as the server closes its end of the connection and shutdown lets us handle batch input correctly.

```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int         maxfdp1, stdineof;
    fd_set      rset;
    char        buf[MAXLINE];
    int     n;
```

```
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)) {  /* socket is readable */
        if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
            if (stdineof == 1)
                return;     /* normal termination */
            else
                err_quit("str_cli: server terminated prematurely");
        }

        Write(fileno(stdout), buf, n);
    }

    if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
        if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
            stdineof = 1;
            Shutdown(sockfd, SHUT_WR);  /* send FIN */
            FD_CLR(fileno(fp), &rset);
            continue;
        }

        Writen(sockfd, buf, n);
    }
}
```

```
}
```

- stdineof 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.
- **Normal and premature termination.** When we read the EOF on the socket, and:
    - If we have already encountered an EOF on standard input, this is normal termination and the function returns.
    - If we have not yet encountered an EOF on standard input, the server process has prematurely terminated. We now callread and write to operate on buffers instead of lines and allow select to work for us as expected.
- shutdown. When we encounter the EOF on standard input, our new flag, stdineof, is set and we call shutdown with a second argument of SHUT_WR to send the FIN. Here buffers are used instead of lines, using read and writen.

UNIT – III

SOCKET OPTIONS, ELEMENTRY UDP 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 – gethostbyaddr function – getservbyname and getservbyport functions.

3.1 Socket Options

Various types of options are available in a socket. There are various ways to get and set the options that affect a socket. They include,

- getsockopt and setsockopt functions
- fcntl function
- ioctl function

(a) getsockopt and setsockopt

Syntax

```
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);

Return Value : Both return 0 on OK, -1 on error.
```

sockfd – refer to an open socket descriptor

level –specifies the code in the system that interprets the option. (i.e) general socket code or protocol specific code (IPv4, IPv6,TCP)

optname – name of the option

optval – It 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 getsockopt.

Optlen – specifies the size of this variable.

(b) fcntl

fcntl stands for “file control”. This function performs various descriptor control operations.

Syntax

```
int fcntl (int fd, int cmd, ….int arg);

Return value: 0 if OK, -1 on error
```

(c) ioctl

ioctl stands for “IO control”.

Syntax

```
int ioctl (int fd, int request, ….void *arg);

Return value: 0 if OK, -1 on error
```

3.2 Generic Socket Options

These options are protocol independent options. These options are as follows.

(1) SO_BROADCAST

- This option enables or disables the ability of the process to send broadcast messages.
- Broadcasting is supported only for datagram sockets and only on networks that support the concept of broadcast message.
- An application must set this socket option before sending any broadcast message.
- If the destination address is a broadcast address and this socket option is not set, EACCESS is returned.

(2) SO_DEBUG

- This option is supported only by TCP.
- When this option is enabled for a TCP socket, the kernel keeps track of the detailed information about all the packets send and received by the TCP for the socket.
- These are kept in a circular buffer and can be examined with the trpt program.

(3) SO_DONTROUTE

- This option specifies that the outgoing packets are to bypass the normal routing mechanisms of the underlying protocol.
- According to the destination address given, the packets will be routed.
- So, a local interface will be identified and then the packet is routed.
- If the local interface cannot be identified, ENETUNREACH is returned.

- This option can also be applied to individual datagrams using MSG_DONTROUTE flag.
- This option is often used by the routing daemons to bypass the routing table and force a packet to be sent out a particular interface.

(4) SO_ERROR

- When an error occurs on a socket, the protocol module sets a variable named so_error for that socket to one of the standard unix Exxx values. This is called the pending error for the socket.
- This option can be fetched but cannot be set.
- The process can be notified about the error in one of the two ways.
    - If the process is blocked in a call to select on the socket, for either readability or writability.
    - If the process is using signal driven I/O, the SIGIO signal is generated for either the process or the process group.

(5) SO_KEEPALIVE

- The purpose of this option is to detect if the peer host crashes or become unreachable.
- When the keepalive option is set for a TCP socket and no data has been exchanged across the socket in either direction for 2 hours, TCP automatically sends a keep-alive probe to the peer.
- This probe is a TCP segment to which the peer must respond. One of the three scenarios result.
    - The peer responds with the expected ACK (If the peer is active)
    - The peer responds with an RST, which tells the local TCP that the peer host has been crashed and rebooted. So, the sockets pending error is set to ECONNRESET and the socket is closed.
    - There is no response from the peer to the keep-alive probe. TCP sends 8 additional probes, 75 seconds apart, trying to get a response from the peer. It will give up if there is no response within 11 minutes and 15 seconds from the time of sending the first probe. If there is no response, the sockets pending error is set to ETIMEDOUT and the socket are closed. If the peer host is unreachable, the pending error is set to EHOSTUNREACH.
- This option is normally used by servers, although clients can also use this option.

- Servers use this option because after establishment of connection, there may be a situation where the server may wait for the client request.
- But, if the client hosts crashes, powered off or connection drops, the server never knows about it and waits for the input that can never arrive. This is called a half open connection. The keep-alive option will detect these half open connections and terminate them.

(6) SO_LINGER

- This option specifies how the close function operates for a connection oriented protocol.
- By default, close returns immediately. But, if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer.
- But, the SO_LINGER changes this default case.
- It requires the following structure to be passed between the user process and the kernel.

```
Struct linger
{
        int l_onoff;    /* 0 = off, non-zero = on */
        int l_linger    /* linger time */
}
```

- When this socket option is set, any one of the following three scenarios takes place, depending on the values of the two structure members.
    - If l_onoff=0, the option is turned off. So, the value of l_linger is ignored and the TCP default applies (i.e) close returns immediately.
    - If l_onoff=nonzero and l_linger = 0, TCP aborts the connection when it is closed. (ie) TCP discards any data still remaining in the socket send buffer and sends a RST to its peer.
    - If l_onoff=nonzero and l_linger = nonzero, then the kernel will linger when the socket is closed. (i.e) If there is any data still remaining in the socket send buffer, the process is put to sleep until either,
        - All data is send and acknowledged by the peer TCP.
        - The linger time expires.
- Assume that the client writes data to the socket and then calls close. The following diagrams depict the various scenarios.

(a) Default situation

- By default, close returns immediately.

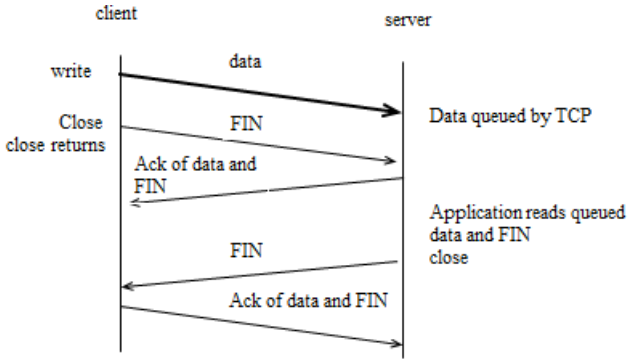


Fig 3.1 Default operation of close

We now need to look at exactly when *close* on a socket returns and what the actions and consequences are. In these cases, we assume that the client writes data to the socket and then calls close.

Fig. 3.1 shows the default scenario. Assume that when the client's 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 added.

But by default, the client's close returns immediately. As we see here, the client's close can return before the server reads the remaining data in its socket receive buffer. Therefore it is possible for the server host to crash before the server application reads this remaining data, and the client application will never know.

(b) SO_LINGER socket option is set and l_linger set to a positive value

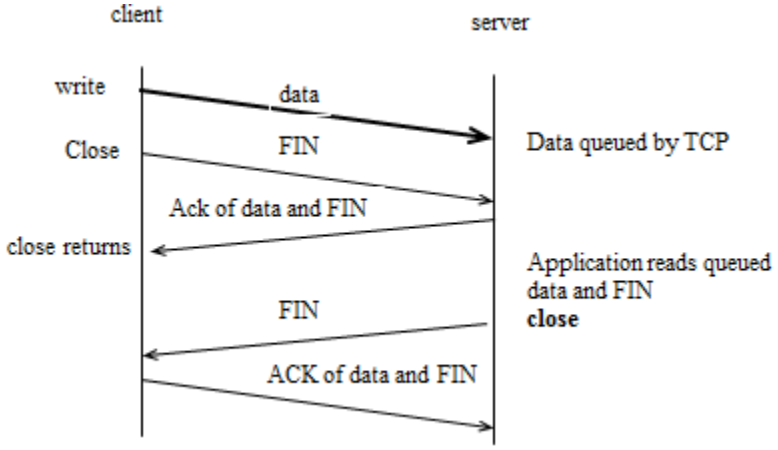


Fig 3.2 l_linger set to a positive value

In this scenario, the client sets the SO_LINGER option, specifying some positive linger time. When this occurs, the client's close does not return until all the client's data and its FIN have been acknowledged by the server TCP as shown in Fig 3.2.

The server host can crash before the server application reads its remaining data, and the client application will never know.

(c) SO_LINGER socket option set with l_linger set to small positive value

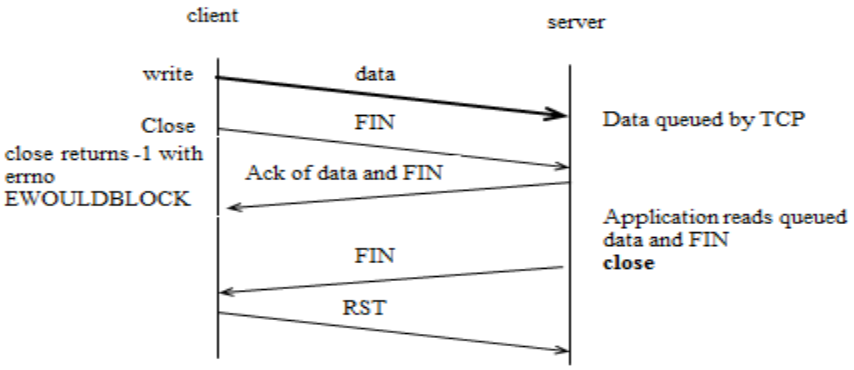


Fig 3.3 l_linger set to small positive value

Fig. 3.3 shows what can happen if the SO_LINGER option is set to a value that is too low. The basic principle here is that a successful return from close, with the SO_LINGER option set, only tells us that the data we sent (and our FIN) have been acknowledge by the peer TCP. It does not tell us whether the peer application has read the data. If we do not set the SO_LINGER option, we do not know whether the peer TCP has acknowledged the data.

(d) Using shutdown to show the peer has received the data

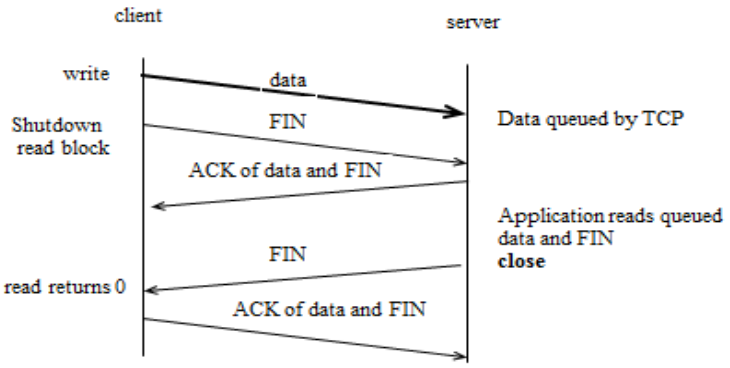


Fig. 3.4 Using shutdown

One way for the client to know that the server has read its data is to call shutdown (with SHUT_WR) instead of close and wait for the peer to close its end of the connection as shown in Fig. 3.4.

(e) Application ACK

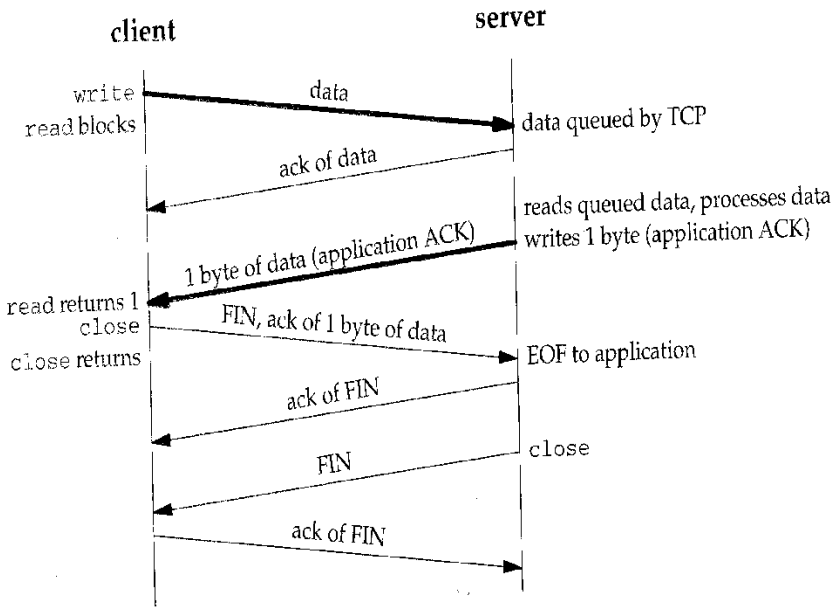


Fig. 3.5 Application ACK

Another way to know that the peer application has read our data is to use an application-level acknowledgment which requires coding in the server and client. In this case, the client waits for a 1 byte acknowledgment for each packet sent. Fig 3.5 shows the possible packet exchange.

(7) SO_OOBINLINE

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

(8) SO_RCVBUF and SO_SNDBUF

- Every socket has a send buffer and a receive buffer.
- Receive buffer - It is used to hold the received data until it is read by the application.
- Send buffer – It is used to hold the data to be send.

- The socket receive buffer has a limit in its window size. And, the peer can send data only upto the window size limit. The window size will be advertised to its peer while sending the SYN segment during connection establishment. This is TCPs flow control.
- If the peer ignores the advertised window and if it sends data beyond the window, the receiving TCP discards it.
- The default size of TCP send and receive buffers is 4096 bytes. But, newer systems use larger values from 8192 to 61440 bytes.
- The default size of UDP send buffer is 9000 bytes.
- The default size of UDP receive buffer is 40000 bytes.
- The main goal of this option is that these two options let us change the default sizes.

(9) SO_RCVLOWAT and SO_SNDLOWAT

- Every socket has a receive low water mark and send low water mark.
- These are used by the select function.
- Receive low water mark – It is the amount of data that must be in the socket receive buffer for the select to return readable.
- Send low water mark – It is the amount of available space that must exist in the socket send buffer for select to return writable.
- Default receive low water mark is 1.
- Default send low water mark is 2048.
- These two socket options, let us change these two low water marks.

(10) SO_RCVTIMEO and SO_SNDTIMEO

- These two socket options allow us to place a timeout on socket receive and send.
- This let us specify the timeout in seconds and microseconds.
- The timeout can be disabled by setting its value to 0 seconds and 0 microseconds.
- Both timeouts are disabled by default.
- The receive timeout affects the five input functions namely read, readv, recv, recvfrom and recvmsg.
- The send timeout affects the five output functions namely write, writev, send, sendto and sendmsg.

(11) S0_REUSEADDR and SO_REUSEPORT

- SO_REUSEADDR serves four different purposes.
  - It allows a listening server to start and bind its well known port even if previously established connections exist that use this port as their local port. This condition is typically encountered as follows.
    - A listening server is started.
    - A connection request arrives and a child process is spawned to handle that client.
    - The listening server terminates, but the child continues to service the client on the existing connection.
    - The listening server is restarted.
  - It allows a new server to be started on the same port as an existing server that is bound to the wildcard address as long as each instance binds a different local IP address.
  - It allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address.
  - It allows completely duplicate bindings : A bind of an IP address and port, when the same IP address and port are already bound to another socket, if the transport protocol supports it.
- This feature is supported only for UDP sockets.
- This feature is used with multicasting to allow the same application to be run multiple times on the same host.
- SO_REUSEADDR does the following.
  - It allows completely duplicate bindings, but only if each socket that wants to bind the same IP address and port specify this socket option.
  - It is considered equivalent to SO_REUSEPORT if the IP address being bound is a multicast address.
- Limitation : It is not supported by all systems.

(12) SO_TYPE

- This option returns the socket type.
- The integer value returned is a value SOCK_STREAM or SOCK_DGRAM or SOCK_RAW.

(13) SO_USELOOPBACK

- This option applies only to sockets in the routing domain.

- By default, this set to ON.
- When this option is enabled, the socket receives a copy of everything sent on the socket.

3.3 **IPv4 Socket Options**

- These socket options are processed by IPv4. These options include the following.

(1) IP_HDRINCL
  - If this option is set for a raw IP socket, we must build our own IP header for all the datagrams we send on the raw socket.
  - Normally kernel builds the IP header for all datagrams, but some applications require to build their own IP header.
  - When this option is set, we build a complete IP header, with the following exceptions.
    - IP always calculates and stores the IP header checksum.
    - If we set the IP identification field to 0, the kernel will set the field.
    - If the source IP address is INADDR_ANY, IP sets it to the primary IP address of the outgoing interface.
    - Setting IP options is implementation dependent.
    - Some fields must be in host byte order and some in network byte order. This is implementation dependent.

(2) IP_OPTIONS
  - Setting this option allows us to set IP options in the IPv4 header.
  - This requires intimate knowledge of the format of IP options in the IP header.

(3) IP_RECVDSTADDR
  - This option causes the destination IP address of a received UDP datagram to be returned as ancillary data by recvmsg.

(4) IP_RECVIF
  - This option causes the index of the interface on which a UDP datagram is received to be returned as ancillary data by recvmsg.

(5) IP_TOS
  - This option let us set the type of service in the IP header for a TCP, UDP socket.
  - TOS can be,
    - T – Throughput
    - R – Reliability

- D – Delay
- C - Cost

(6) IP_TTL

- TTL stands for Time to Live
- This option let us set and fetch the default TTL.

3.4 ICMPv6 Socket Option

- This socket option is processed by ICMPv6.

(1) ICMP_FILTER

- This option let us fetch and set an icmp6_filter structure that specifies which of the 256 possible ICMPv6 message types will be passed to the process on a raw socket.

3.5 IPv6 Socket Option

- These socket options are processed by IPv6. These options include the following.

(1) IPv6_CHECKSUM

- This option specifies the byte offset into the user data where the checksum field is located.
- If this value is non-negative, the kernel will,
  - Compute and store a checksum for all outgoing packets.
  - Verify the received checksum on input, discarding packets with an invalid checksum.
- If the value is -1 (default), the kernel will not calculate and store the checksum for outgoing packets on this raw socket and will not verify the checksum for received packets.

(2) IPv6_DONTFRAG

- Setting this option disables the automatic insertion of a fragment header for UDP and raw sockets.
- When this option is set, output packets larger than Maximum Transfer Unit (MTU) of the outgoing interface will be dropped.

(3) IPv6_NEXTHOP

- This option specifies the next hop address for a datagram as a socket address structure and is a privileged operation.

(4) IPv6_PATHMTU

- This option cannot be set, only retrieved.
- When this option is retrieved, the current MTU as determined by PATH_MTU discovery is returned.

(5) IPv6_RECVDSTOPTS

- Setting this option specifies that any received IPv6 destination options are to be returned as ancillary data by recvmsg.

(6) IPv6_RECVHOPLIMIT

- Setting this option specifies that the received hop limit field is to be returned as ancillary data by recvmsg.

(7) IPv6_RECVHOPOPTS

- Setting this option specifies that any received IPv6 hop-by-hop options are to be returned as ancillary data by recvmsg.

(8) IPv6_RECVPATHMTU

- Setting this option specifies that the path MTU of a path is to be returned as ancillary data by recvmsg.

(9) IPv6_RECVPKTINFO

- Setting this option specifies that the following two pieces of information about a received IPv6 datagram are to be returned as ancillary data by recvmsg.
  - The destination IPv6 address
  - Arriving interface index

(10) IPv6_RECVRTHDR

- Setting this option specifies that a received IPv6 routing header is to be returned as an ancillary date by recvmsg.

(11) IPv6_RECVTCLASS

- Setting this option specifies that the received traffic class is to be returned as ancillary data by recvmsg.

(12) IPv6_UNICAST_HOPS

- Setting this option specifies the default hop limit for outgoing datagrams sent on the socket, while fetching the socket option returns the value of the hop limit that the kernel will use for the socket.

(13) IPv6_USE_MIN_MTU

- Setting this option avoids fragmentation.
  - When this option is set to 1, path MTU discovery is not performed and packets are sent using minimum MTU.

- When this option is set to 0, causes path MTU discovery to occur for all destinations.
- When this option is set to -1, path MTU discovery is performed.

(14) IPv6_V6ONLY

- Setting this option restricts it to IPv6 communication only.

(15) IPv6_XXX

- UDP socket uses, recvmsg and sendmsg
- TCP socket uses, getsockopt and setsockopt

3.6 **TCP Socket Options**

(1) TCP_MAXSEG

- This socket option allows us to fetch or set the Maximum Segment Size (MSS) for a TCP connection.
- The value returned is the maximum amount of data that the TCP will send to the other end.
- The MSS is set while sending the SYN segment to the peer during connection establishment.
- The maximum amount of data that our TCP will send per segment can also change during the life of the connection if TCP supports path MTU discovery.
- If the route of the peer changes, this value will go up or down.

(2) TCP_NODELAY

- If this option is set, it disables TCP's Nagle algorithm.
- By default, this algorithm is enabled.
- Nagles algorithm avoids the syndrome caused in the sender side (i.e) if the sending side sends data too slowly, by sending each byte as a packet and waiting for the acknowledgment.

Nagle's Algorithm

(1) It sends the first byte as it is as a packet and waits for an acknowledgment.
(2) When it receives the ACK, it does not send the further byte as it is, provided it waits until a certain number of bytes gets accumulated or till the ACK for the previous is arrived.

- The purpose of Nagle's algorithm is to reduce the number of small packets in WAN.
- Small packet is any packet smaller than MSS.
- The two common generators of small packets are the Rlogin and Telnet clients, since they send each keystroke as a separate packet.
- In a fast LAN, we normally donot notice a Nagle's algorithm because the time required for a small packet to be acknowledged is typically a few milliseconds, far less than the time between two successive characters that we type.
- But in a WAN, it takes nearly a second to acknowledge a small packet, so we can notice a delay in the character echoing and this delay is often exaggerated by the Nagle's algorithm.
- Consider the following example,
  - We type the six character string "hello!" with exactly 250 ms between each character.
  - The Round Trip Time (RTT) to the server is 600 ms and the server immediately sends back the echo of each character.
  - Assuming the Nagle's algorithm is disabled, we have the 12 packets as shown below.

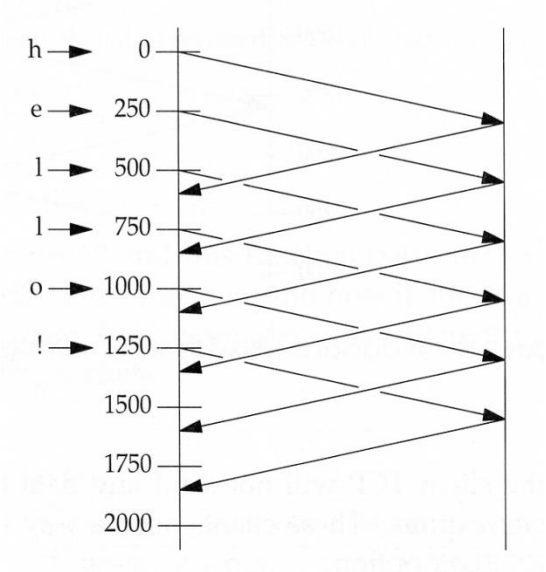


Fig 3.6 Nagle's algorithm (disabled)

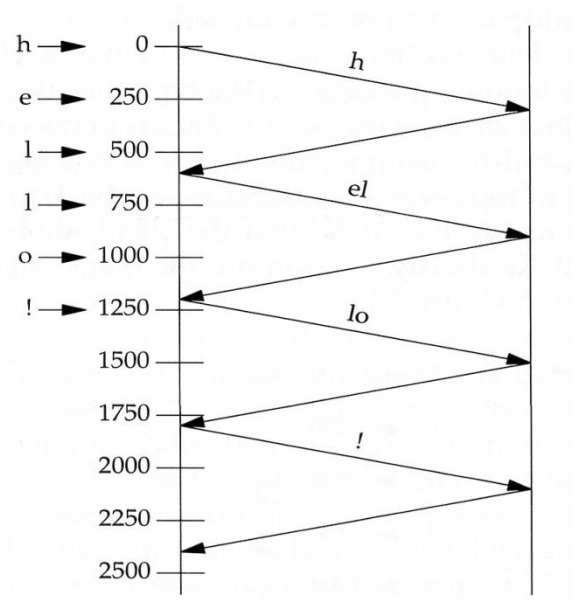


Fig. 3.7 Nagle's Algorithm (Enabled)

- The purpose of the Nagle algorithm is to reduce the number of small packets on a WAN.
- The algorithm states that if a given connection has outstanding data then no small packets will be sent on the connection in response to a user write operation until the existing data is acknowledged.
- Two common generators of small packets are the rlogin and telnet clients, since they normally send each keystroke as a separate packet.
- Fig. 3.6 shows the algorithm disabled. Fig. 3.7 shows the algorithm enabled.
- The characters are typed with 250 milliseconds between each.
- Round Trip Time (RTT) is 600 milliseconds.

3.7 Elementary UDP Sockets

Typical UDP client

- Client does not establish a connection with the server
- Client sends a datagram to the server using **sendto** function

Typical UDP server

- Does not accept a connection from a client

- Server calls **recvfrom** function which waits until data arrives from some client

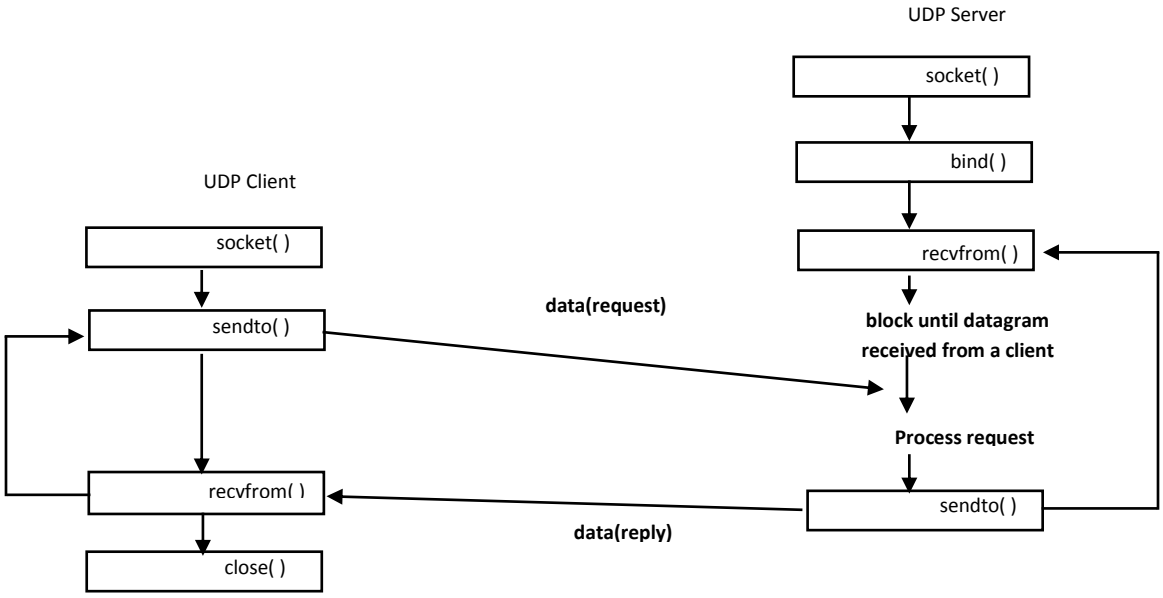


Fig. 3.8 Socket functions for UDP client server

Syntax

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

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,

                 struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,

                 const struct sockaddr *to, socklen_t addrlen);

//Both return: number of bytes read or written if OK,-1 on error
```

- **Both return the amount of user data in the datagram received**

3.8 Program

UDP Echo Server : main

```
#include “unp.h”

int main(int argc, char **argv)

{

    int sockfd;

    struck sockaddr_in servaddr,cliaddr;

    sockfd=Socket(AF_INET,SOCK_DGRAM,0);

    bzero(&servaddr,sizeof(servaddr));

    servaddr.sin_fammily=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));

}
```

UDP Echo server :dg_echo function

```
#include “unp.h”

void dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)

{

   int n;

   socklen_t len;

   char mesg[MAXLINE];
```

```
for( ; ; ) {
    len=clilen;
    n=Recvfrom(sockfd, mseg, MAXLINE, 0, pcliaddr, &len);
    sendto(sockfd, mesg, n, 0, pcliaddr, len);
} }
```

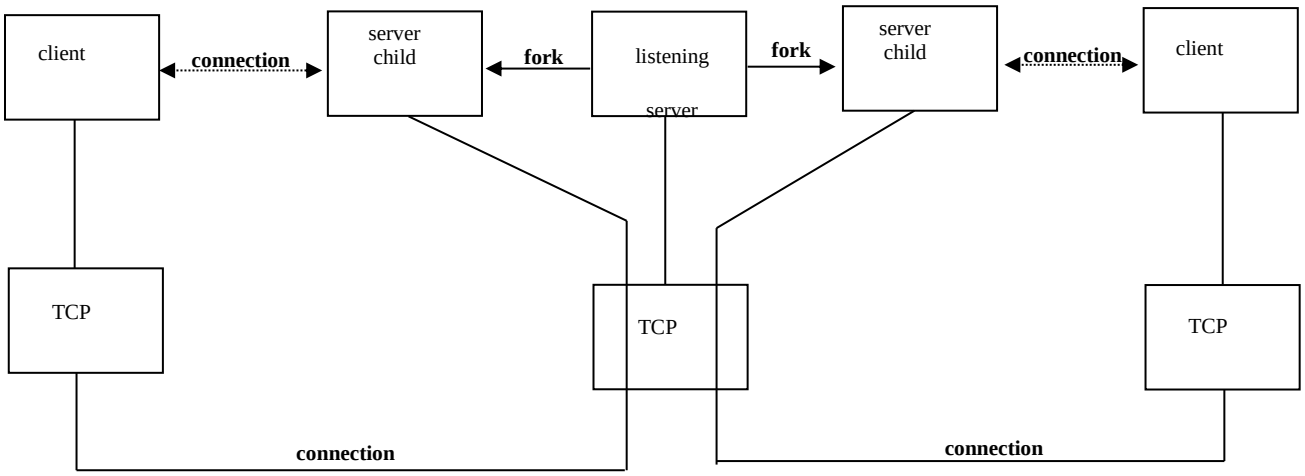


Fig. 3.9 Summary of TCP client-server with two clients

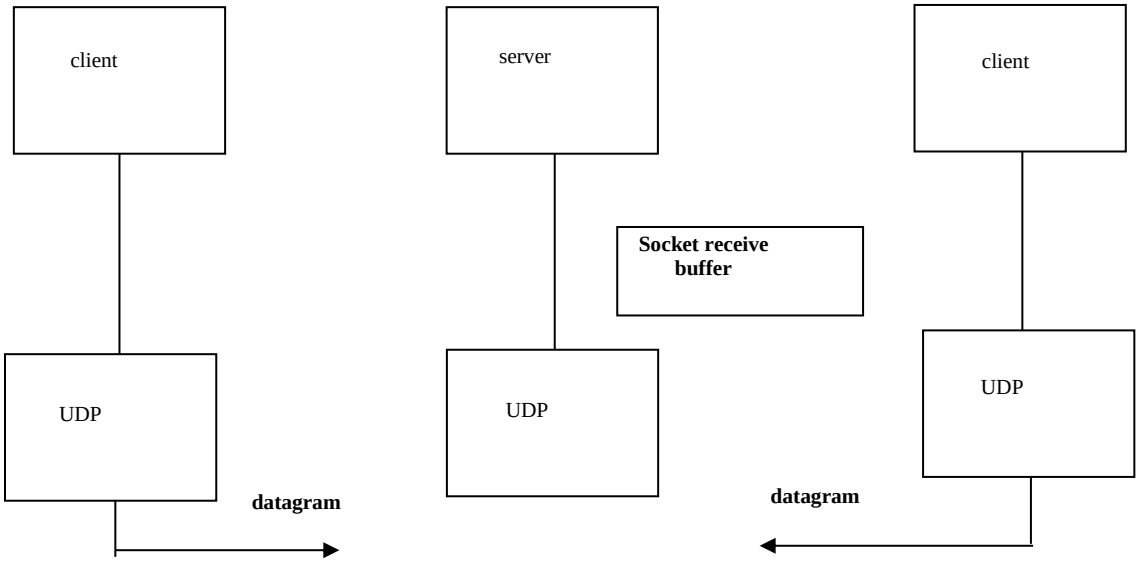


Fig. 3.9 Summary of UDP client-server with two clients

UDP Echo client : main

```
#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);

}
```

UDP Echo client : 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);

    }

}
```

3.9. 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);
/* create 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);
/* create 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.

3.10 Domain Name System (DNS)

Every machine, a host (or) a server is identified using the numeric address known as IP address and using the numeric port numbers. Remembering the numeric address for all servers is very difficult. So, inorder to remember the servers, names are assigned to them.

So, a mapping should be done to match the host names to their corresponding IP addresses. The system that is used for this mapping is known as DNS.

Functions used

1) gethostbyname – Converts names to their IP address
2) gethostbyaddr - Converts IP address to their corresponding host name
3) getservbyname – converts the service names to port no's
4) getservbyport - converts the port no to their corresponding service names

Resource Records

Entries in the DNS are known as resource records

Few types

1) A – An A record maps a host name into a 32-bit IPV4 address
2) AAAA – Called 'quad A' maps a host name into a 128-bit IPV6 address
3) PTR - 'Pointer Records' maps IP addresses into host names
    - For an IPV4 address,the conversion of IP address to decimal ASCII value is done first and then in_addr.arpa is appended.The resulting string obtained is the host name
    - For an IPV6 address,the conversion of IP address to hexadecimal ASCII value is done first and then ip6.arpa is appended.The resulting string obtained is the host name

4) MX – called 'Mail Exchanger' specifies a host to act as a mail exchanger for the specified host
5) CNAME – called 'canonical name.If people use these service names instead of host names,it is transparent when a service is moved to another host.

Typical arrangement of clients,servers and resolvers

Organizations run one or more name servers,often the program known as BIND(Barkeley Internet Name Domain)

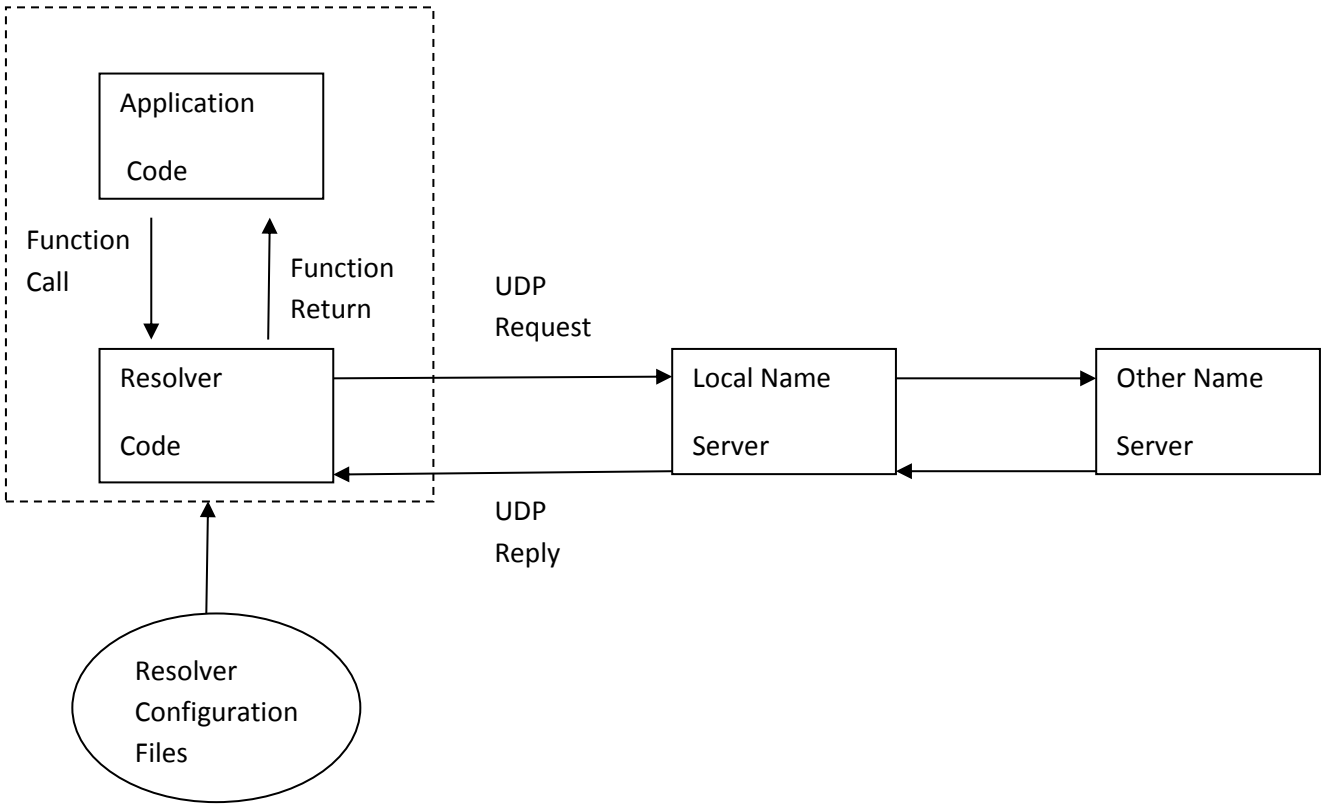


Fig. 3.10 Typical arrangement of clients, servers and resolvers

- Applications such as clients and servers contact a DNS server by calling functions in a library known as resolver. The common resolver functions are gethostbyname and gethostbyaddr.
- The resolver code can be inbuilt in the system library or it may be centralized where the applications can share it.
- Multiple name servers are often required for readability and redundancy.

Functions

1) gethostbyname Function

Host computers are often known by their human readable names.This function converts the hostname to their corresponding IP addresses

If successful,it returns a pointer to a hostnet structure that contains all the IPV4 addresses for the host

```
Struct hostent * gethostbyname(const char *
```

Returns : Non-null pointer if OK

Null on error

2) gethostbyaddr Function

This function takes a binary IPV4 address and tries to find the hostname corresponding to that address

```
Struct hostent * gethostbyaddr(const char * addr, socklen-t len, int family);
```

Returns : Non-null pointer if OK

Null on error

3) getservbyname Function

This function converts the service name to their corresponding port numbers

```
Struct servent * getservbyname(const char * servname, const char * protoname);
```

Returns : Non-null pointer if OK

Null on error

4) getservbyport Function

This function converts the port numbers to their corresponding service names

```
Struct servent * getservbynport(int port, const char * protoname);
```

Returns : Non-null pointer if OK
Null on error

3.11 IPv6 Support in DNS

The above four functions are protocol dependent. POSIX includes protocol independent functions namely getaddrinfo() getnameinfo().

- These functions provide name/address conversions as part of the sockets library.
- In the future it will be important to write code that can run on many protocols (IPV4, IPV6), but for now these functions are not widely available.
  - It's worth seeing how they work even though we probably can't use them yet!

(1) getaddrinfo

Syntax

int getaddrinfo(const char *hostname, const char *service, const struct addrinfo* hints, struct addrinfo **result);

- getaddrinfo() replaces both gethostbyname() and getservbyname()
- **hostname** is a hostname or an address string (dotted decimal string for IP).
- **service** is a service name or a decimal port number string.

Struct addrinfo

struct addrinfo {

int ai_flags;

int ai_family;

int ai_socktype;

int ai_protocol;

size_t ai_addrlen;

```
char *canonname;

struct sockaddr *ai_addr;

struct addrinfo *ai_next;

};
```

hints is an **addrinfo *** (can be **NULL**) that can contain:

- **ai_flags** (**AI_PASSIVE** , **AI_CANONNAME**)
- **ai_family** (**AF_*XXX***)
- **ai_socktype** (**SOCK_*XXX***)
- **ai_protocol (IPPROTO_TCP, etc.)**

result is returned with the address of a pointer to an **addrinfo** structure that is the head of a linked list.

It is possible to get multiple structures:

- multiple addresses associated with the **hostname**.
- The **service** is provided for multiple socket types.

(2) getnameinfo()

Syntax

int getnameinfo(const struct sockaddr *sockaddr, socklen_t addrlen char *host, size_t hostlen, char *serv, size_t servlen, int flags);

- getnameinfo() looks up a hostname and a service name given a sockaddr

QUESTIONS

PART - A

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. Explain gethostbyaddr function.
14. Explain gethostname function.
15. Explain getservbyname and getservbyport functions.

PART – B

1. 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.
2. Discuss any six generic socket options in detail.
3. Discuss about IPv4 socket option and ICMP socket options in detail with Suitable example.
4. Explain the purpose and usage of UDP sockets and their different functions.
5. Discuss about IPv6 socket option in detail
6. Briefly discuss about DNS.
7. Briefly discuss about UDP Echo server and client.
8. Explain in detail the TCP socket options.
9. Give a brief note on IPv6 support in DNS

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

4.1 IPv4 and IPv6 Interoperability

Over the coming years, there will probably be a gradual transition of the internet from IPv4 to IPv6. During this transition phase, it is important that existing Ipv4 applications continue to work with newer IPv6 applications. To handle this scenario, the hosts are running dual stacks (i.e) both an IPv4 protocol stack and IPv6 protocol stack.

There are four possible combinations of communication which include the following.

- IPv4 client, IPv6 server over dual-stack server host
- IPv6 client over dual-stack client host, IPv4 server
- IPv6 address macro, function and option
- Source code portability

4.1.1 IPv4 Client, IPv6 Server:

- A general propoerty of a dual stack host is that Ipv6 servers can handle both Ipv4 and Ipv6 clients. This is done using IPv4 mapped IPv6 addresses.
- Figure 4.1 shows an IPv4 client and an Ipv6 client on the left and IPv6 server on the right. Both clients send SYN segments to establish a connection with a server.
- The Ipv4 client host will send SYN in an IPv4 datagram and the IPv6 client host send the SYN in an IPv6 datagram. IPv6 dual stack server can handle both IPv4 and IPv6 clients. This is done using IPv4-mapped IPv6 address. The server create an IPv6 listening socket that is bound to the IPv6 wildcard address.

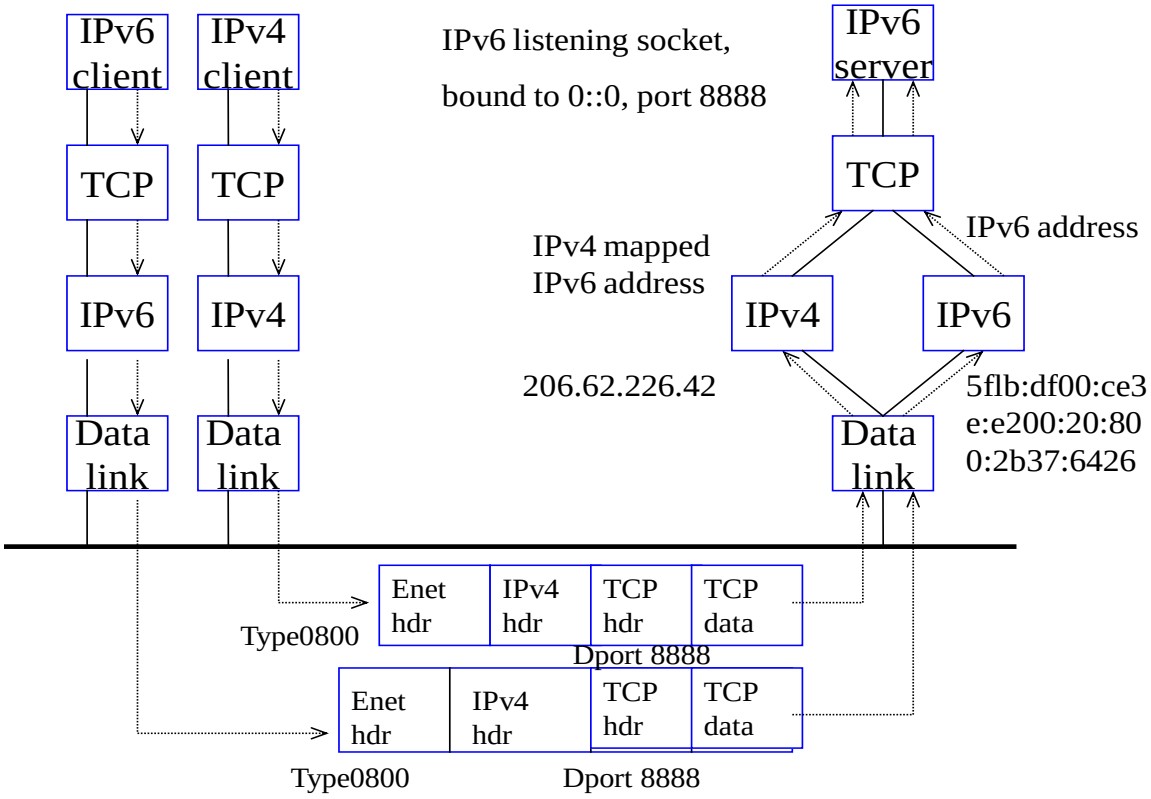


Fig 4.1 IPv6 server on dual stack host serving IPv4 and IPv6 clients

- The TCP segment from the Ipv4 client appears on the wires as an ethernet header followed by an Ipv4 header, a TCP header, and the TCP data.The ethernet header contains a type field of 0x0800 which identifies the frame as an Ipv4 frame.The TCP header contains an destination port which identifies the frame as an Ipv4 frame.

- The TCP segment from the Ipv6 client appears on the wire as an ethernet header folowed by an IPv6 header,an IPv6 header,a TCP header and the TCP data. The ehernet header contains a type field of 0x86dd which identifies the frame as an IPv6 frame.

- The **steps** that allow an IPv4 TCP client to communicate with an IPv6 server as follows:

  1. The IPv6 server starts,creates an IPv6 listening socket and it binds the wildcard address to the socket.

2. The IPv4 client calls gethostbyname and finds an A record for the server.
3. The clint calls connect and the client's host sends an IPv4 SYN to the server.
4. The server host receives the Ipv4 SYN directed to the Ipv6 listening socket and responds with an Ipv4 SYN/ACK.
5. When the server host sends to the Ipv4 mapped Ipv6 address ,its IP stack generates the Ipv4 datagrams to the Ipv4 address.
6. Unless the server explicitly checks whether this IPv6 address is an IPv4 mapped IPv6 address, the server never knows that it is communicationg with an IPv4 client. Similarly, the IPv4 client has no idea that it is communicating with an IPv6 server.

- The scenario is similar for an IPv6 UDP server. But, the address format can change for each datagram.

Summary

- If an IPv4 datagram is received for an IPv4 socket, nothing is done.
- If an IPv6 datagram is received for an IPv6 socket, nothing is done.
- When an IPv4 datagram is received for an IPv6 socket, the kernel returns the corresponding IPv4 mapped IPv6 address as the address returned by accept (TCP) or recvfrom (UDP).
- The converse is false (i.e) An IPv6 address cannot be represented as an IPv4 address.

Figure 4.2 summarizes how a received IPv4 or IPv6 datagram is processed, depending on the type of the receiving socket, for TCP and UDP, assuming a dual stack host.

Rules

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

- A listening IPv4 socket can accept incoming connections from only IPv4 clients.

- If a server has a listening IPv6 socket that has bound the wild card address and the IPv6_v6 ONLY socket option is not set, that socket can accept incoming connections from either IPv4 clients or IPv6 clients.

- If a server has a listening IPv6 socket that has bound an IPv6 address or the wild card address but has set the IPv6_v6ONLY socket option, that socket can accept incoming connections from IPv6 clients only.

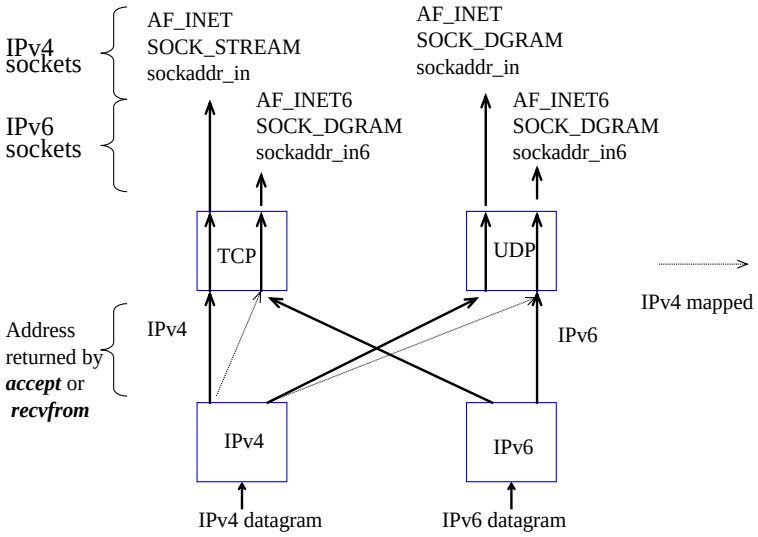


Fig. 4.2 Processing of received IPv4 or IPv6 datagrams, depending on type of receiving socket

4.1.2 IPv6 Client, IPv4 Server

Consider an IPv6 TCP client running on a dual stack host.

- An IPv4 server starts on an IPv4 only host and creates an IPv4 listening socket.
- The IPv6 client starts and calls getaddrinfo asking for only IPv6 addresses.IPv4 server host has only A records.
- The IPv6 client calls connect with the IPv 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 an IPv4 SYN/ACK, and the connection is established using IPv4 datagrams.

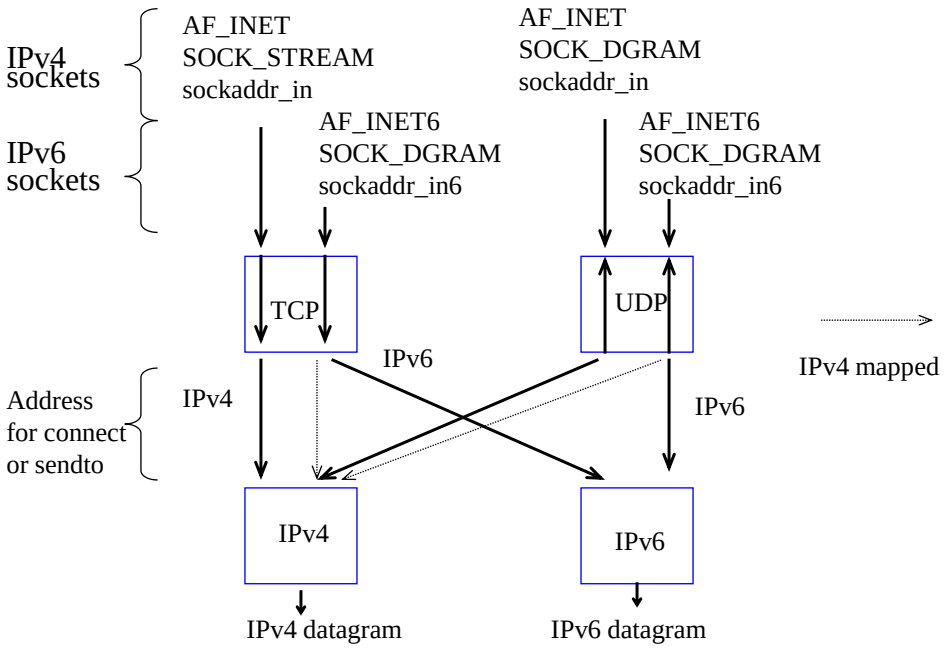


Fig. 4.3 Processing of client requests, depending on address type and socket type

Figure 4.3 summarizes this scenario as follows.

- ❖ If an IPv4 TCP client calls connect specifying an IPv4 address or if an IPv6 UDP client calls sendto specifying an IPv4 address, nothing special is done.
- ❖ 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.
- ❖ 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 datagram to be sent instead of an IPv6 datagram.
- ❖ An IPv4 client cannot specify an IPv6 address to either connect or sendto because a 16 byte IPv6 address doesnot fit in the 4 byte in_addr structure.

4.2 Threads

In the traditional UNIX model, when a process need something performed by another entity, it forks a child process and lets the child perform the processing. This is similar to the example of a concurrent server.

Problems with Fork

(1) Fork is expensive

- ❖ Memory is copied from the parent to the child and all the descriptors are duplicated in the child.
- ❖ This makes fork more expensive.

(2) Interprocess communication (IPC)

- ❖ IPC is required to pass information between the parent and the child after the fork.
- ❖ Passing the information before the fork is easy. But, returning information from the child to the parent takes more work.

These **problems** can be overcome using threads.

- Threads are light weight processes.
- Thread creation can be 10 to 100 times faster than a process creation.
- All threads within a process share the same global memory. So, sharing of information becomes easy between threads.
- The problem with threads is synchronization.

All the threads within a process shares the following.

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

Each thread has a unique,

- ➢ Thread ID
- ➢ set of registers, stack pointer

- stack for local variables, return addresses
- signal mask
- priority
- Return value: errno

Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.

4.2.1 **Basic thread functions (creation and termination)**

There are five basic thread functions. They are as follows.

(1) pthread_create()

- The pthread_create() function creates a thread.
- When a program is started, a single thread is created, called the initial thread.
- Additional threads are created by pthread_create.

Syntax

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*func)(void *), void *arg);
```

Parameters

thread->(Output) Pthread handle to the created thread

attr->(Input) The thread attributes object containing the attributes to be associated with the newly created thread. If NULL, the default thread attributes are used.

func->(Input) The function to be run as the new threads start routine

arg->(Input) An address for the argument for the threads start routine

Return Value

0->pthread_create() was successful.

Value->pthread_create() was not successful. value is set to indicate the error condition.

(2) pthread_join()

- The pthread_join() function waits for a thread to terminate, detaches the thread, then returns the threads exit status. (similar to waitpid)
- If the status parameter is NULL, the threads exit status is not returned.
- The meaning of the threads exit status (value returned to the status memory location) is determined by the application, except for the following conditions:

1. When the thread has been canceled using pthread_cancel(), the exit status of PTHREAD_CANCELED is made available.
2. When the thread has been terminated as a result of an unhandled OS/400 exception, operator intervention or other proprietary OS/400 mechanism, the exit status of PTHREAD_EXCEPTION_NP is made available.

Syntax

```
#include <pthread.h>
 int pthread_join(pthread_t thread, void **status);
```

Parameters

thread->(Input) Pthread handle to the target thread

status->(Output) Address of the variable to receive the thread's exit status

Return Value

0->pthread_join() was successful.

Value->pthread_join() was not successful. value is set to indicate the error condition.

(3) pthread_detach()

- The pthread_detach() function indicates that system resources for the specified thread should be reclaimed when the thread ends. If the thread is already ended, resources are reclaimed immediately. This routine does not cause the thread to end. After pthread_detach() has been issued, it is not valid to try to pthread_join() with the target thread.

- A thread is either joinable or detached. When a joinable thread terminates, its thread ID and exit status are retained until another thread calls pthread_join.
- A detached thread is like a daemon process. When it terminates, all its resources are released and we cannot wait it to terminate.

Syntax

```
#include <pthread.h>
 int pthread_detach(pthread_t thread);
```

Parameters

thread->(Input) Pthread handle to the target thread

Return Value

0->pthread_detach() was successful.

Value->pthread_detach() was not successful. value is set to indicate the error condition.

(4) pthread_self()

- The pthread_self() function returns the Pthread handle of the calling thread.
- The pthread_self() function does NOT return the integral thread of the calling thread. You must use pthread_getthreadid_np() to return an integral identifier for the thread.

Syntax

```
#include <pthread.h>
 pthread_t pthread_self(void);
```

Return Value

pthread_t->pthread_self() returns the Pthread handle of the calling thread.

(5) pthread_exit()

- The pthread_exit() function terminates the calling thread and makes the value value_ptr available to any successful join with the terminating thread.

- An implicit call to pthread_exit() is made when a thread other than the thread in which main() was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.
- The process exits with an exit status of 0 after the last thread has been terminated. The behaviour is as if the implementation called exit() with a zero argument at thread termination time.

Syntax

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

Return value

The pthread_exit() function cannot return to its caller.

4.3 TCP echo server using threads

1. In TCP echo server ,use one thread per client instead of one child process per client. When accep returns,call the pthread_create instead of fork.
2. The thread can execute the doit function. The thread share all descriptors with the main thread. The pointer to the connfd is the final argument of pthread_create.
3. A thread is created and doit function is scheduled to start executing.
4. Another connection is ready and the main thread runs again.Accept returns,connfd is stored and main thread calls pthread_create.

```
#include”unpthread.h”

Static void *doit(void *)

int main(int argc,char ** argv)

{
```

```
Int listenfd,connfd;

Pthread_t tid;

Socklen_t addrlen,len;

Struct sockaddr *cliaddr;

If(argc==2)

Listenfd=Tcp_listen(NULL,argv[1],&addrlen;

Else if If(argc==3)

Listenfd=Tcp_listen(argv[1],argv[2],&addrlen);

Else

Err_quit(“usage service or port”);

Cliaddr=malloc(addrlen);

for(;;)

{

len=addrlen;

connfd=Accept(listenfd,cliaddr,&len);

pthread_create(&tid,NULL,&doit,(void*) connfd);

}

}

Static void* doit(void *arg)

{

Pthread_detach(pthread_self());
```

```
Str_echo(int) arg);

Close((int)arg);

 Return(NULL);

}
```

If two threads are created,both will operate on the final value stored in connfd.It will create a problem. it can be solved by passing the value of connfto pthread_create instead of pointer to value.

4.4 Mutex

Mutual exclusion (often abbreviated to **mutex**) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections. A critical section is a piece of code in which a process or thread accesses a common resource. The critical section by itself is not a mechanism or algorithm for mutual exclusion. A program, process, or thread can have the critical section in it without any mechanism or algorithm which implements mutual exclusion.

Examples of such resources are fine-grained flags, counters or queues, used to communicate between code that runs concurrently, such as an application and its interrupt handlers. The synchronization of access to those resources is an acute problem because a thread can be stopped or started at any time.

To illustrate: suppose a section of code is altering a piece of data over several program steps, when another thread, perhaps triggered by some unpredictable event, starts executing. If this second thread reads from the same piece of data, the data, which is in the process of being overwritten, is in an inconsistent and unpredictable state. If the second thread tries overwriting that data, the ensuing state will probably be unrecoverable. These shared data being accessed by critical sections of code, must therefore be protected, so that other processes which read from or write to the chunk of data are excluded from running.

A **mutex** is also a common name for a program object that negotiates mutual exclusion among threads, also called a lock.

Syntax

```
# include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mptr);

int pthread_mutex_unlock (pthread_mutex_t *mptr);
```

Return Value

0 - If Ok

+ve value - On Error

4.5 Condition Variables

Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs. Condition variables are user-mode objects that cannot be shared across processes.

Condition variables enable threads to atomically release a lock and enter the sleeping state. They can be used with critical sections or slim reader/writer (SRW) locks. Condition variables support operations that "wake one" or "wake all" waiting threads. After a thread is woken, it re-acquires the lock it released when the thread entered the sleeping state.

Condition Variable provides a signaling mechanism.

Syntax

```
# include <pthread.h>

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

int pthread_cond_signal (pthread_cond_t *cptr);
```

Return Value

0 - If Ok

+ve value - On Error

Pthread_cond_signal awakens one thread that is waiting on the condition variable. There are instances where multiple threads should be awakened, in which case, **pthread_cond_broadcast** will wake up all the threads that are blocked on the condition variable. Pthread_cond_timedwait lets a thread place a limit on how long it will block.

Syntax

```
# 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);
```

Return Value

0 - If Ok

+ve value - On Error

4.6 Raw Sockets

It is a socket that takes packets, bypasses the normal TCP/IP processing and sends them to the application that wants them.

Features

a) These sockets let us read and write ICMPv4,IGMPv4 and ICMPv6 packets. It processes two ICMP messages (Router advertisement and Router solicitation) that the kernel knows nothing about.

b) With a raw socket, a process can read and write Ipv4 datagrams with an Ipv4 protocol field that is not processed by the kernel. This capability carries over to Ipv6 also.

c) With a raw socket, a process can build its own Ipv4 header using the IP_HDRINCL socket option.

Typical Uses

- ICMP messages
  - ping generates ICMP echo requests and received ICMP echo replies.
- Routing protocols
  - gated implements OSPF routing protocol.
  - Uses IP packets with protocol ID 89 – not supported by kernel.
- Hacking
  - Generating your own TCP/UDP packets with spoofed headers

4.6.1 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.

4.6.2 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.

4.6.3 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.

The following **tests** are performed for each raw socket and only if all three tests are true is the datagram delivered to the socket.

- 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.

4.7 Ping

- **The operation of ping**
- Ping program that works with both IPv4 IPv6.
- Very simple program that uses ICMP to send a ping to another machine over the Internet.
- Provides the option to send a defined number of packets
- It is used to understand the network programming concepts and techniques without being distracted by all these options.

Table 4.1 Format of ICMP messages

0 7 8 15 16 31

Type	Code	Check sum
Identifier	Sequence number	
Optional data		

Ping.h header is shown below.

```
#include        <netinet/in_systm.h>
#include        <netinet/ip.h>
#include        <netinet/ip_icmp.h>
#define         BUFSIZE                 1500
                        /* globals */
char    sendbuf[BUFSIZE];
int             datalen;                        /* # bytes of data following ICMP header */
char    *host;
int             nsent;                          /* add 1 for each sendto() */
pid_t   pid;                            /* our PID */
int             sockfd;
int             verbose;
                        /* function prototypes */
void    init_v6(void);
void    proc_v4(char *, ssize_t, struct msghdr *, struct timeval *);
void    proc_v6(char *, ssize_t, struct msghdr *, struct timeval *);
void    send_v4(void);
void    send_v6(void);
void    readloop(void);
void    sig_alrm(int);
void    tv_sub(struct timeval *, struct timeval *);
struct proto {
  void    (*fproc)(char *, ssize_t, struct msghdr *, struct timeval *);
  void    (*fsend)(void);
  void    (*finit)(void);
  struct sockaddr  *sasend;    /* sockaddr{} for send, from getaddrinfo */
  struct sockaddr  *sarecv;    /* sockaddr{} for receiving */
  socklen_t         salen;              /* length of sockaddr{}s */
  int               icmpproto;  /* IPPROTO_xxx value for ICMP */
} *pr;
#ifdef  IPV6
#include        <netinet/ip6.h>
```

```
#include        <netinet/icmp6.h>
#endif
```

The main function is given below

```
struct proto     proto_v4 = { proc_v4, send_v4, NULL, NULL, NULL, 0, IPPROTO_ICMP };
#ifdef  IPV6
struct proto     proto_v6 = { proc_v6, send_v6, init_v6, NULL, NULL, 0, IPPROTO_ICMPV6 };
#endif
int       datalen = 56;           /* data that goes with ICMP echo request */
int
main(int argc, char **argv)
{
        int                             c;
        struct addrinfo *ai;
        char *h;
        opterr = 0;             /* don't want getopt() writing to stderr */
        while ( (c = getopt(argc, argv, "v")) != -1) {
                switch (c) {
                case 'v':
                        verbose++;
                        break;
                case '?':
                        err_quit("unrecognized option: %c", c);
                }
        }
        if (optind != argc-1)
                err_quit("usage: ping [ -v ] <hostname>");
        host = argv[optind];
        pid = getpid() & 0xffff; /* ICMP ID field is 16 bits */
        Signal(SIGALRM, sig_alrm);
        ai = Host_serv(host, NULL, 0, 0);
        h = Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
        printf("PING %s (%s): %d data bytes\n",
                        ai->ai_canonname ? ai->ai_canonname : h,
                        h, datalen);
                /* 4initialize according to protocol */
        if (ai->ai_family == AF_INET) {
                pr = &proto_v4;
#ifdef  IPV6
        } else if (ai->ai_family == AF_INET6) {
                pr = &proto_v6;
                if (IN6_IS_ADDR_V4MAPPED(&(((struct sockaddr_in6 *)
                                                        ai->ai_addr)->sin6_addr)))
                        err_quit("cannot ping IPv4-mapped IPv6 address");
#endif
        } else
                err_quit("unknown address family %d", ai->ai_family);
        pr->sasend = ai->ai_addr;
```

```
        pr->sarecv = Calloc(1, ai->ai_addrlen);
        pr->salen = ai->ai_addrlen;
        readloop();
        exit(0);
}
```

4.8 Trace route Program

Traceroute lets us determine the path that IP datagrams follow from our host to some other destination

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.

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 */ 11 };
/* globals */
char    recvbuf   [BUFSIZE];
charsendbuf [BUFSIZE];
 int       datalen;        /* # bytes of data following ICMP header */ 16 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 */ 22 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 */
          int     ttllevel;           /* setsockopt () level to set TTL */
          int     ttloptname;         /* setsockopt () name to set TTL */
   } *pr;


   #ifdef IPV6
       #include    <netinet/ip6.h>
       #include    <netinet/icmp6.h>
   #endif
```

The main function is given below

```
struct proto      proto_v4 = { icmpcode_v4, recv_v4, NULL, NULL, NULL, NULL, 0,
                                                    IPPROTO_ICMP, IPPROTO_IP, IP_TTL };
```

```
#ifdef  IPV6
struct proto     proto_v6 = { icmpcode_v6, recv_v6, NULL, NULL, NULL, NULL, 0,
                                                    IPPROTO_ICMPV6, IPPROTO_IPV6,
IPV6_UNICAST_HOPS };
#endif
int              datalen = sizeof(struct rec);    /* defaults */
int              max_ttl = 30;
int              nprobes = 3;
u_short          dport = 32768 + 666;
int
main(int argc, char **argv)
{
        int                                     c;
        struct addrinfo *ai;
        char *h;
        opterr = 0;             /* don't want getopt() writing to stderr */
        while ( (c = getopt(argc, argv, "m:v")) != -1) {
                switch (c) {
                case 'm':
                        if ( (max_ttl = atoi(optarg)) <= 1)
                                err_quit("invalid -m value");
                        break;
                case 'v':
                        verbose++;
                        break;
                case '?':
                        err_quit("unrecognized option: %c", c);
                }
        }
        if (optind != argc-1)
                err_quit("usage: traceroute [ -m <maxttl> -v ] <hostname>");
        host = argv[optind];
        pid = getpid();
        Signal(SIGALRM, sig_alrm);
        ai = Host_serv(host, NULL, 0, 0);
        h = Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
        printf("traceroute to %s (%s): %d hops max, %d data bytes\n",
                  ai->ai_canonname ? ai->ai_canonname : h,
                  h, max_ttl, datalen);
                /* initialize according to protocol */
        if (ai->ai_family == AF_INET) {
                pr = &proto_v4;
#ifdef  IPV6
        } else if (ai->ai_family == AF_INET6) {
                pr = &proto_v6;
                if (IN6_IS_ADDR_V4MAPPED(&(((struct sockaddr_in6 *)ai->ai_addr)-
>sin6_addr)))
                        err_quit("cannot traceroute IPv4-mapped IPv6 address");
#endif
        } else
```

```
                err_quit("unknown address family %d", ai->ai_family);
        pr->sasend = ai->ai_addr;               /* contains destination address */
        pr->sarecv = Calloc(1, ai->ai_addrlen);
        pr->salast = Calloc(1, ai->ai_addrlen);
        pr->sabind = Calloc(1, ai->ai_addrlen);
        pr->salen = ai->ai_addrlen;
        traceloop();
        exit(0);
}
```

UNIT – V SNMP

The History of SNMP Management- Internet Organizations and Standards- The SNMP Model- The Organization Model- System Overview- The Information Model. SNMPv1 Network Management: Communication Model and Functional Models. Introduction to RMON, SNMP Management: Major Changes in SNMPv2- SNMPv2 System Architecture- SNMPv2 Structure of Management Information- The SNMPv2 Management Information Base- SNMPv2 Protocol- Compatibility with SNMPv1- SNMPv3 Architecture- SNMPv3 Applications- SNMPv3 Management Information Base.

5.1 Network Management System (NMS)

Goal

- To ensure that the users of network are provided services with a quality of service that they expect.
- It can be defined as Operations, Administration, Maintenance and Provisioning (OAMP) of network and services.

5.1.1 Network Management Dumbbell architecture

- The network management is concerned with network resources such as hubs, switches, bridges, routers and gateways and the connectivity between them via gateways.
- It is also concerned with the end to end connectivity between any two processors in the network.
- A network consists of network components and their interconnection.
- Each vendor who manufactures the network components is qualified to develop an NMS to manage the product or set of products.
- If a network uses products developed by different vendors then different NMS should be installed for each product. So, an NMS should be installed such that it can manage different vendor components of a network.
- Thus, common management system and interoperability between different vendor NMS plays a major role.
- Out of several standards, the two most prominent standards include, the **Internet (SNMP)** developed by IETF (Internet Engineering Task Force) and OSI developed by ISO.
- Network management Dumbbell architecture for interoperability is shown below, where two vendor systems A and B exchange common management messages.

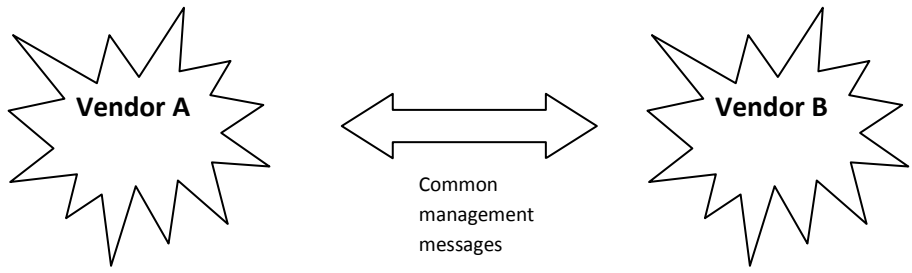


Fig 5.1 Dumbbell architecture for interoperability

- The message consists of ,
    - Management information data (Type ID, status of managed objects etc.)
    - Management controls (Setting and changing configuration of an object)
- Application services are the management related applications such as fault management and configuration management.
- Management protocols are CMIP for the OSI model and SNMP for the Internet model.
- Transport protocols are the first four OSI layers for the OSI model and TCP/IP over any of the first two layers for the Internet model.
- **SNMP** management is also known as internet management.
- Any network that uses TCP/IP protocol suite is an ideal candidate for SNMP management.
- SNMP is the most widely used NMS.
- If a new component like router or bridge that has an SNMP agent built in is added to the managed network, the NMS can automatically start monitoring the added component.

5.2 History of SNMP Management

- It began in 1970's.
- To remotely monitor and configure gateways, SGMP (Simple Gateway Monitoring Protocol) was developed.
- SGMP was enhanced and named as SNMP.

5.3 Internet Organizations and Standards

- IAB (Internet Advisory Board) – Recommended the development of SNMP. It was developed informally by researchers on TCP/IP networks in 1983.
- In 1989, IAB was renamed as Internet Architecture Board.
- It took the responsibility to manage two task forces.
    - IETF (Internet Engineering Task Force) – concerned with the development and standardization for IAB.
    - IRTF (Internet Research Task Force) – handles long term problems.
- InterNIC (Internet Network Information Center) – It is an organization that maintains several archives that contain documents related to the internet and the IETF activities. It includes,
    - RFC (Request for Comment)
    - STD (Standard RFC)
    - RFC (FYI) – For Your Information RFC
- IANA (Internet Assigned Numbers Authority) – central coordinator for assigning unique parameter values for internet protocols.

5.4 SNMP v1

5.4.1 SNMP Model

- An NMS acquires a new network element through a management agent or monitors the ones it has acquired.
- There is a relationship between the manager and agent.
- Since one manager is responsible for managing the designated functions of many agents, it is hierarchical in nature.
- Information is transmitted and received by both manager and the agent.

5.4.2 Organization Model

- The initial organization model of SNMP management is a simple two-tier model.
- It consists of a network agent process, which resides in the managed object, and a network manager process which resides in the NMS and manages the managed object.
- Both manager and agent are software modules.

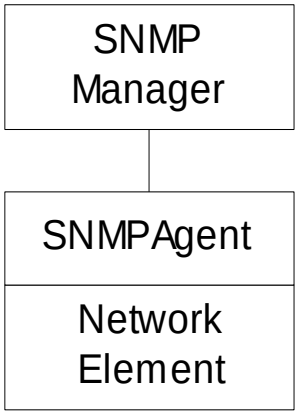
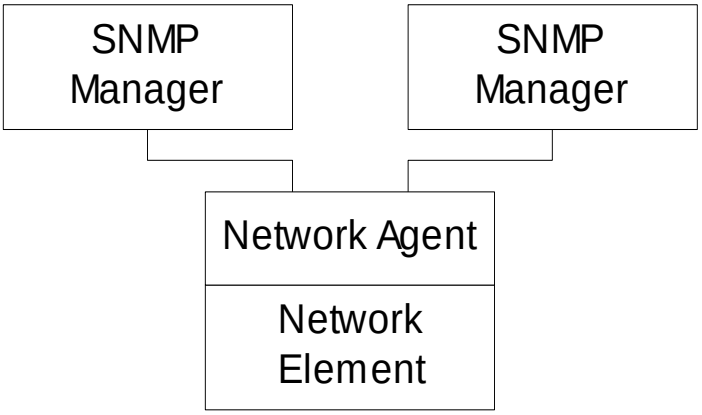


Fig. 5.2 One – Manager – One Agent Model



Multiple- Managers – One Agent Model

- In two-tier model, the network receives raw data from the agents and processes them. In certain situations, it is beneficial for the network manager to obtain preprocessed data.
- This introduces a three-tier architecture.

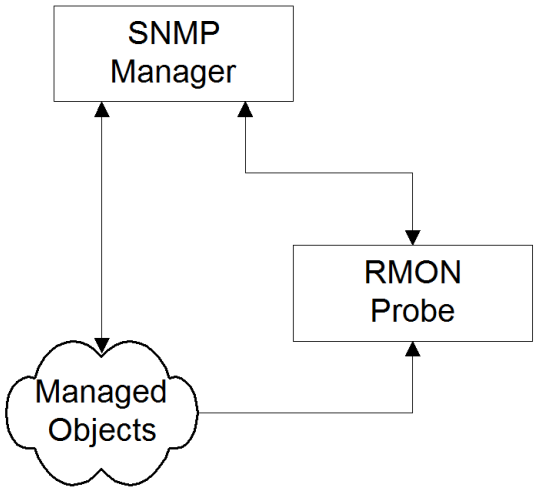


Fig. 5.3 Three-tier organization model

- An SNMP Manager can also manage a network element which does not have an SNMP agent. To do this a proxy server at the central location converts data into a set that is SNMP compatible and communicates with the SNMP manager.

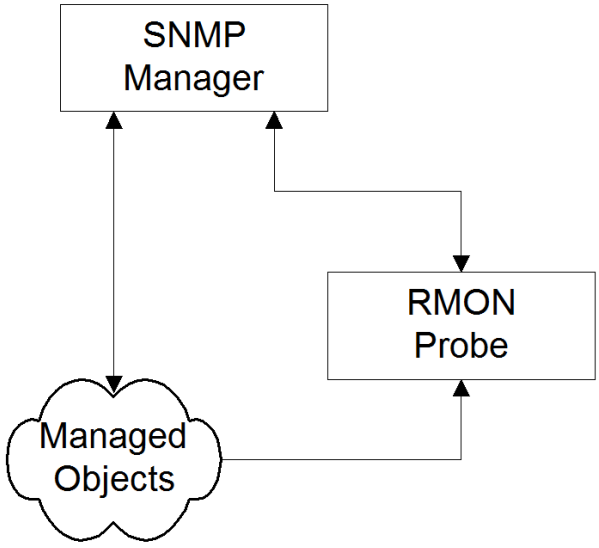


Fig. 5.4 Proxy server organization model

SNMP v1 System Architecture

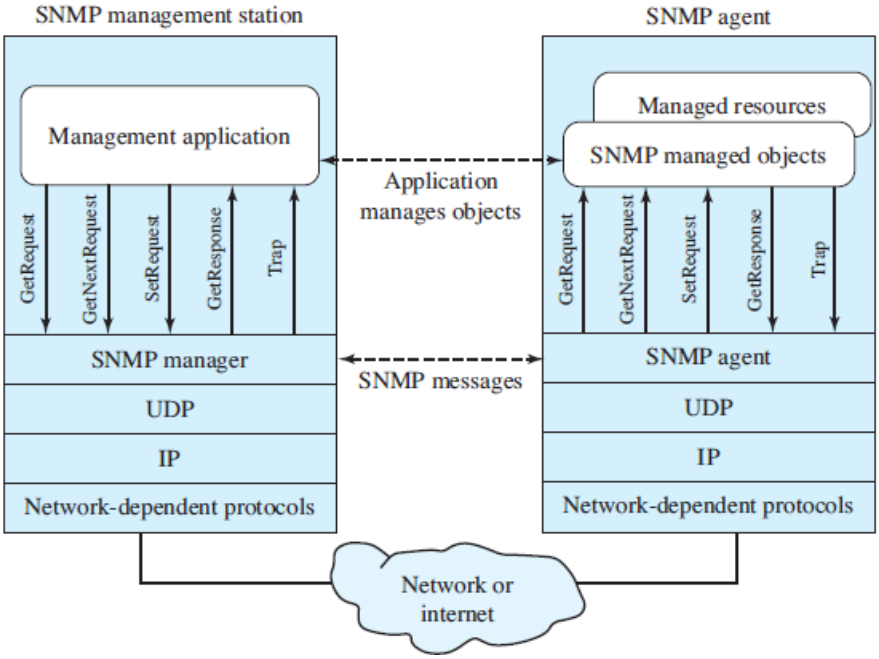


Fig. 5.5 SNMP v1 System Architecture

SNMP Messages

- Get Request:
  - Retrieve the values of objects in the MIB of an agent.
- Get-Next Request:
  - Retrieve the values of the next objects in the MIB of an agent.
- Get-Response:
  - Retrieve the response from the manager.
- Set Request:
  - Update the values of objects in the MIB of an agent.
- Trap Request

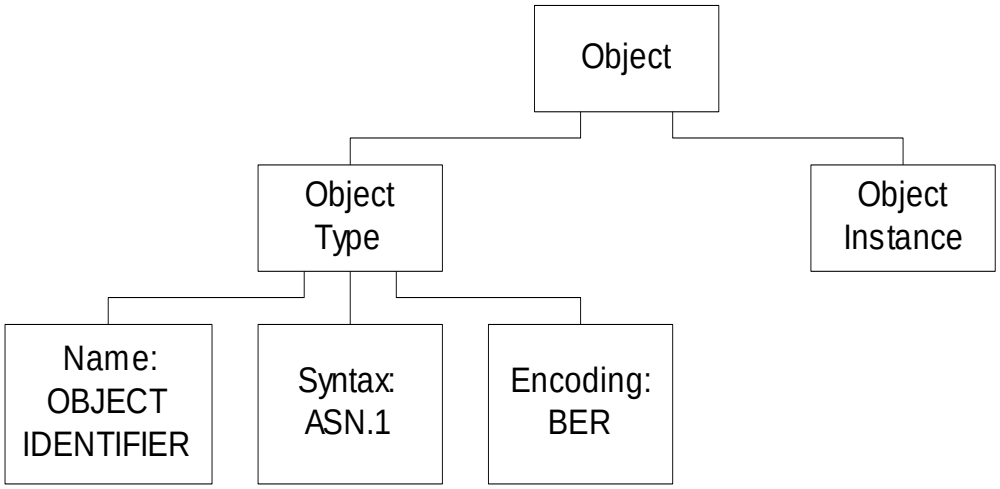
5.4.3 Information Model

- The information model deals with SMI (Structure of Management Information) and MIB (Management Information Base).
- This model deals with the structure and storage of information.
- For information to be exchanged intelligently between manager and agent process, there has to be a common understanding on both the syntax and semantics.
- The syntax used to describe management information is ASN.1 (Abstract Syntax Notation-version 1).
- ASN.1 syntax is based on Backus Naur Form (BNF) which looks like,
  <name>:=<definition>
  (eg) <digit> := 0|1|2|3|4|5|6|7|8|9

- The specification and organizational aspects of managed objects is called the Structure of Management Information (SMI) and is defined in RFC 1155.
- Specifications of managed objects and the grouping of and relationship between managed objects are addressed in the Management Information Base (MIB).

SMI (Structure of Management Information)

- A managed object can be considered to be composed of an object type and an object instance.



- SMI is concerned only with the object type and not the object instance.

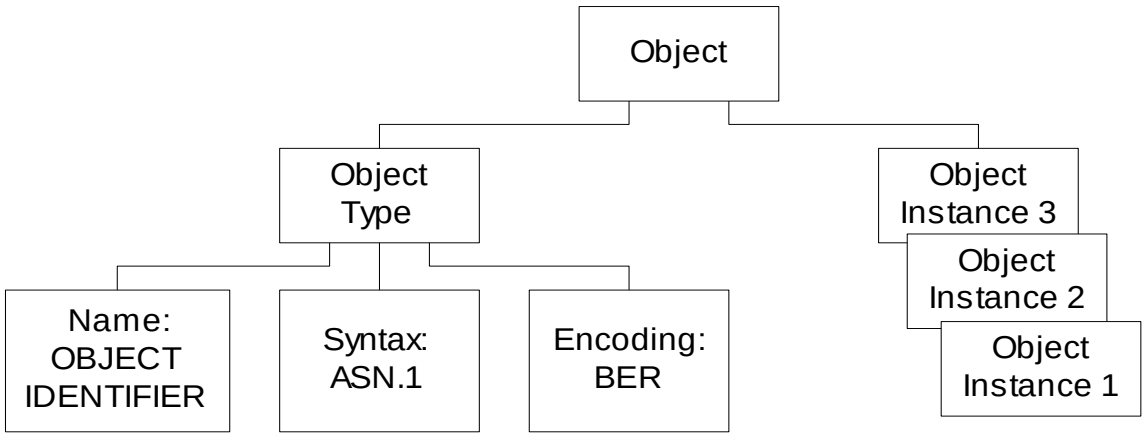


Fig.5.6 Managed Object – multiple instances

- Object is uniquely defined by
  - DESCRIPTOR
  - OBJECT IDENTIFIER

```
internet OBJECT IDENTIFIER ::=
                        {iso org(3) dod(6) 1 }.
```

internet OBJECT IDENTIFIER ::= {iso(1) standard(3) dod(6) internet(1)}
internet OBJECT IDENTIFIER ::= {1 3 6 1}

internet OBJECT IDENTIFIER ::= {iso standard dod internet }

internet OBJECT IDENTIFIER ::= { iso standard dod(6) internet(1) }

internet OBJECT IDENTIFIER ::= { iso(1) standard(3) 6 1 }

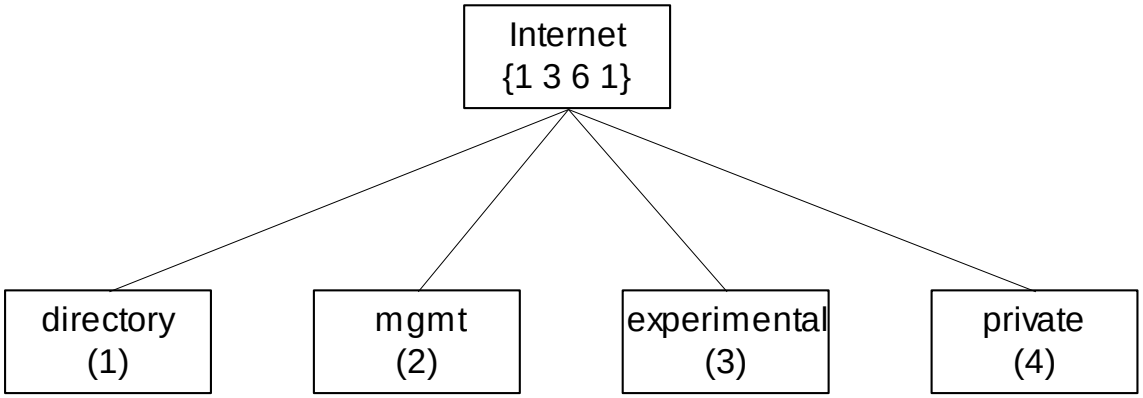


Fig.5.7 Internet Sub nodes

directory OBJECT IDENTIFIER ::= {internet 1}

mgmt OBJECT IDENTIFIER ::= {internet 2}

experimental OBJECT IDENTIFIER ::= {internet 3}

private OBJECT IDENTIFIER ::= {internet 4}

- The figure below shows the example of four commercial vendors CISCO, HP, 3COM and Cabletron who registered as nodes 9,11,43 & 52 respectively.

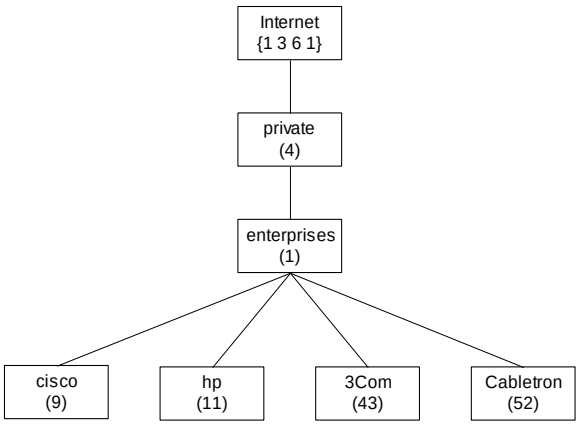


Fig. 5.8 Private MIB

5.4.3 SNMP Communication Model

- This model defines specification of four aspects of SNMP communication.
  - Architecture
  - Administrative model that defines data access policy.
  - SNMP protocol
  - SNMP MIB

1. Architecture

SNMP Messages

- Get-Request

- Get-Next-Request
- Set-Request
- Get-Response
- Trap
    - Generic trap
    - Specific trap

2. Administrative Model

- Based on community profile and policy

SNMP Entities:

a. SNMP application entities
   - Reside in management stations and network elements
   - Manager and agent

b. SNMP protocol entities
   - Communication processes (PDU handlers)
   - Peer processes that support application entities

- The application entity residing in the management station is known as SNMP Manager.
- The application entity in the network element is known as SNMP agent.
- The pairing of the two entities is called as an SNMP community.
- Multiple pairs can belong to same community.

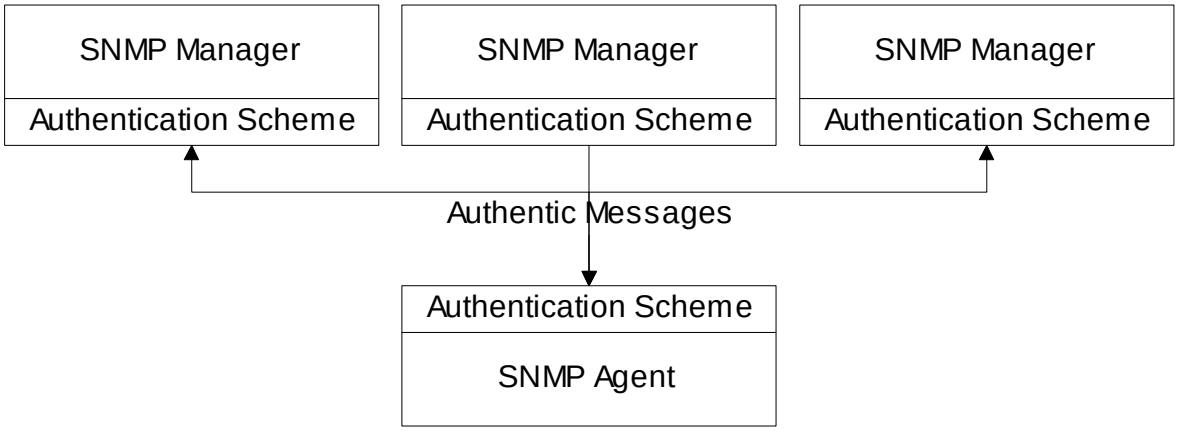


Fig. 5.9 SNMP Community

- A network element consists of many managed objects – both standard and private. However, a management agent will be permitted to view only a subset of network elements managed objects. This is called community MIB view.

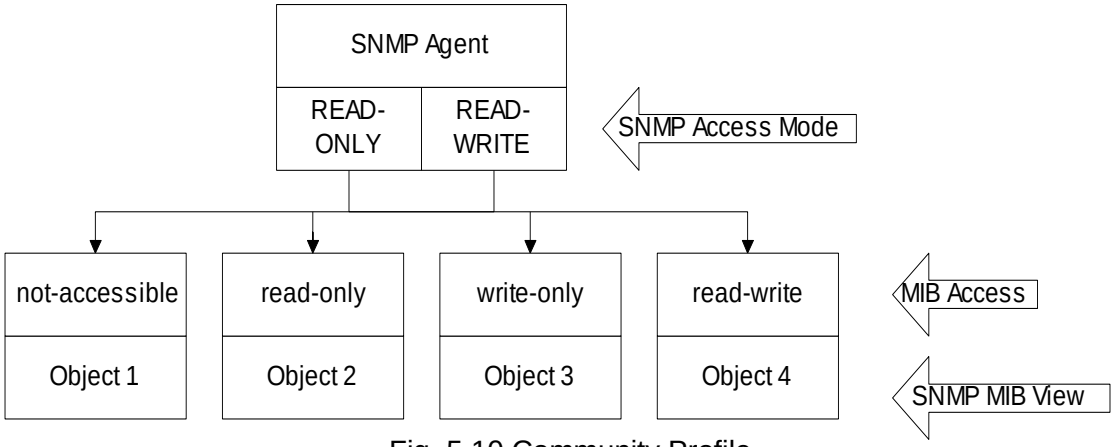


Fig. 5.10 Community Profile

- A pairing of SNMP MIB view with an SNMP access code is called a community profile.
- The pairing of an SNMP community with an SNMP community profile is defined as an SNMP access policy. This defines the administrative model of SNMP management.

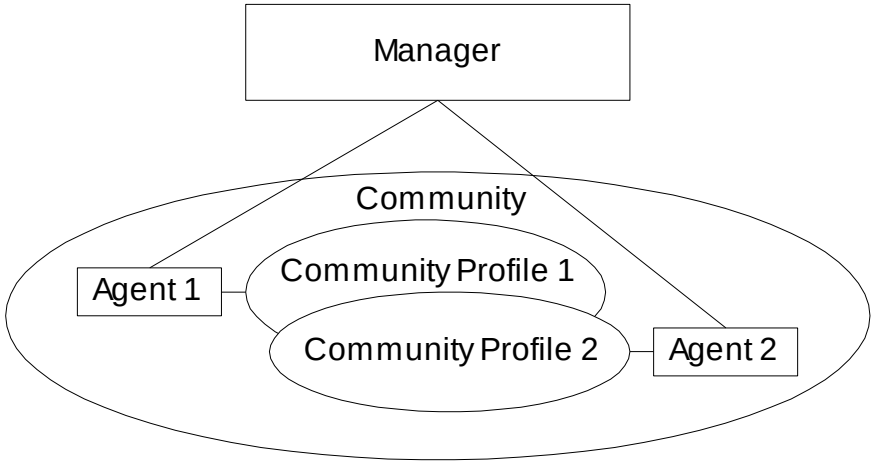


Fig. 5.11 Access policy

3. SNMP Protocol

- Communication among protocol entities is done using messages encapsulated in UDP datagram.
- SNMP message consists of,
  - Version identifier
  - SNMP community name
  - PDU (Protocol Data Unit)

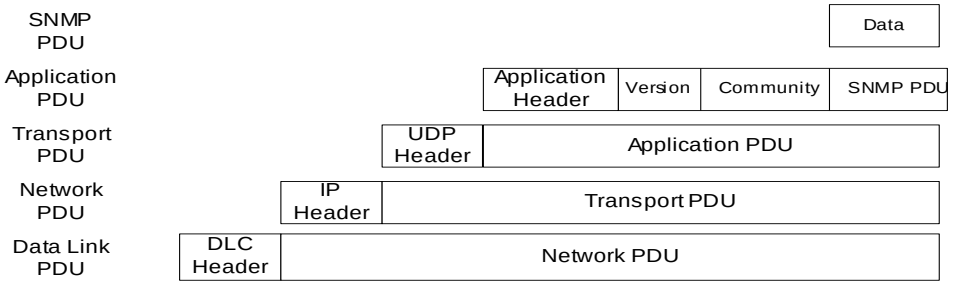


Fig. 5.12 Protocol entities

4. SNMP MIB

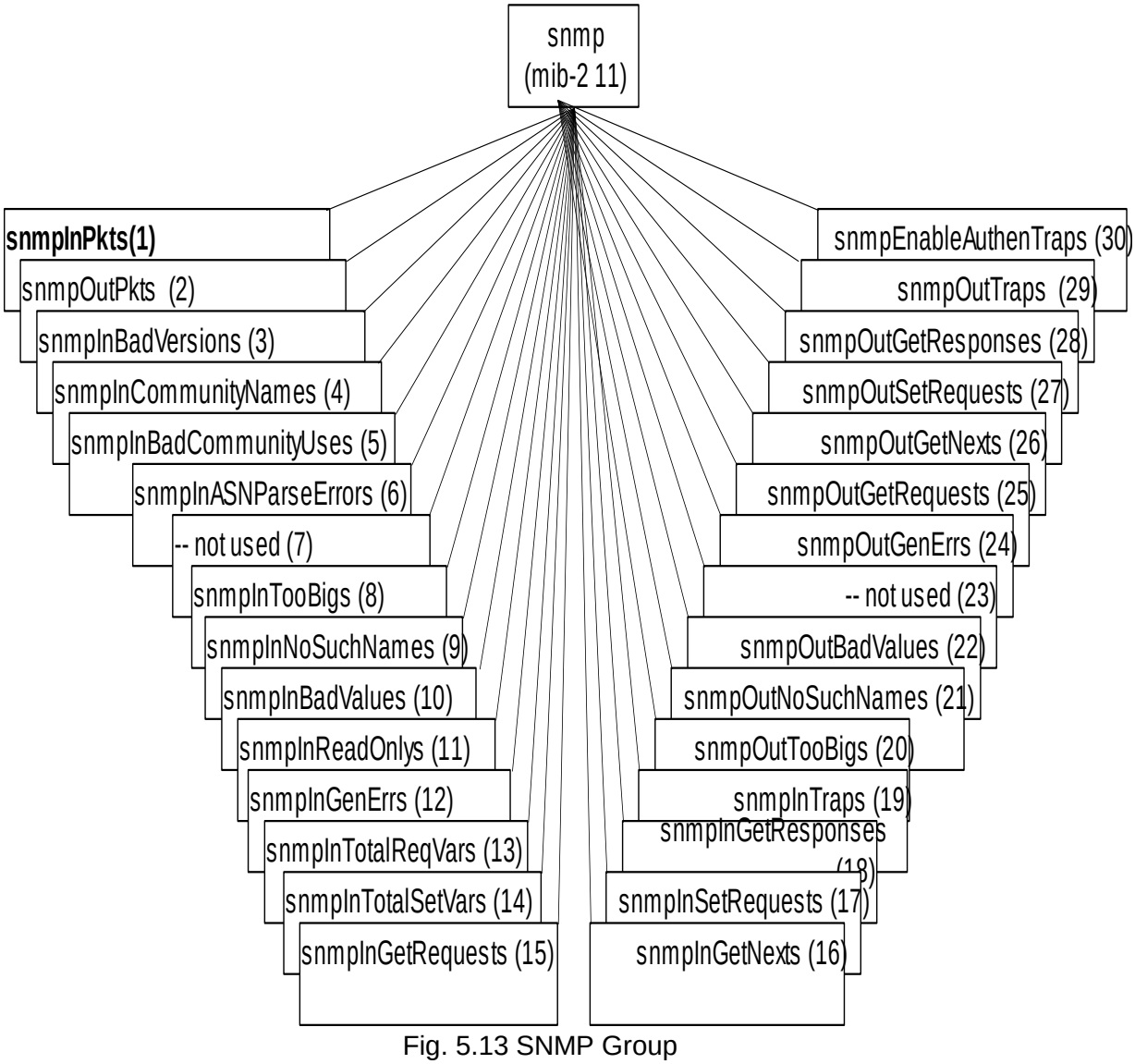


Fig. 5.13 SNMP Group

5.5 SNMP v2

The basic components of network management in SNMPv2 are the same as in version 1. They are agent, manager both performing the same functions.

Improvements in SNMPv2 when compared to SNMPv1

1. Bulk data transfer message

- Ability to request and receive bulk data using the get bulk message.

2. Manager to Manager message
   - Deals with interoperability of two network management systems.
3. SMI (Structure of Management Information)
   - SMI in version 2 is divide into three parts.
     - Module definitions
     - Object definitions
     - Trap definitions
4. Textual conventions
   - These are designed to help define new data types.
5. Conformance statements
   - These help the customer compare the feature of various products.
6. MIB enhancements
7. Table enhancements
8. Transport mappings

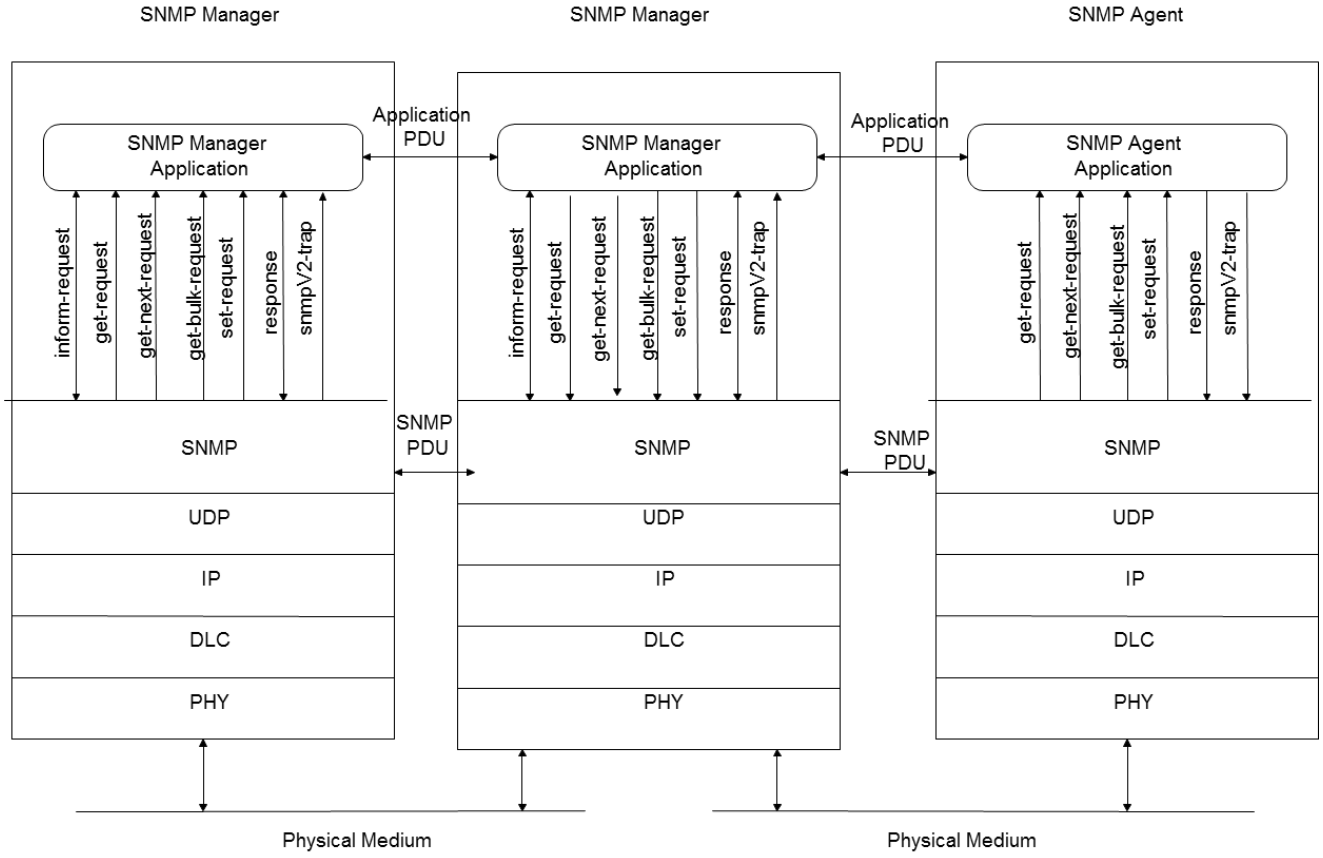


Fig. 5.14 SNMP v2 System Architecture

Additional Messages

- ❑ inform-request
  - ❑ manager-to-manager message
  - ❑ The receiving manager responds with a response message

- Enhances interoperability

- get-bulk-request
  - transfer of large data, e.g. retrieval of table data
- SNMPv2-trap

Similar to trap messages in SNMPv1

SMIv2 – Module Definitions

- Defines and describe semantics of an information module (info. related to network management)
  - added to provide administrative information regarding the informational module and the revision history
- MODULE-IDENTITY macro defines the module definitions

```
MODULE-IDENTITY    MACRO ::=
BEGIN
        TYPE NOTATION ::=
                        "LAST-UPDATED" value (Update UTCTime)
                        "ORGANIZATION" Text
                        "CONTACT-INFO" Text
                        "DESCRIPTION" Text
                        RevisionPart
        VALUE NOTATION ::=
                        value (VALUE OBJECT IDENTIFIER)
        RevisionPart ::=      Revisions | empty
        Revisions ::= Revision | Revisions Revision
        Revision ::=
                "REVISION" value (UTCTime)
                "DESCRIPTION" Text
        -- uses the NVT ASCII character set
        Text ::= """" string """"
END
```

MODULE-IDENTITY Macro

SMI v2 – Object Definitions

- OBJECT IDENTIFIER, OBJECT-IDENTITY, OBJECT-TYPE
  - OBJECT IDENTIFIER defines the *administrative identification* of a node in the MIB
  - OBJECT-IDENTITY macro (defines info. about OID) *assigns* an object identifier to a class of managed objects in the MIB (e.g., defining a class of routers!)
    - The object itself is not managed
  - OBJECT-TYPE macro defines the *type* of a managed object (e.g., a specific router type)
    - Focuses on the details of implementation
  - NOTE:
    - OBJECT-IDENTITY is high level description

- OBJECT-TYPE details description needed for implementation

Differences when compared to version 1

1. There are seven messages instead of five.
2. Two manager applications can communicate with each other at peer level.

5.6 SNMPv3

Features

- Modularization of documentation and architecture.
- Improved security
- The access policy used in SNMP v1 and SNMP v2 is enhanced and formalized in the View based Access Control Model (VACM) in SNMPv3.

Architecture Overview

- An SNMP management network consists of several nodes each with an SNMP entity.
- They interact with each other in monitoring the network and its resources.
- The architecture of an SNMP entity is defined as the elements of that entity and the names associated with them.
- Conceptually SNMPv3 is nothing more than an extension of SNMP to address two major areas, administration and security. A major goal for SNMPv3, though, is to support a module architecture that can be easily extended. This way, for example, if new security protocols are advanced they can be supported by SNMPv3 by defining them as separate modules. Hopefully this will allow us to avoid having to buy books on SNMPv4 in the future.

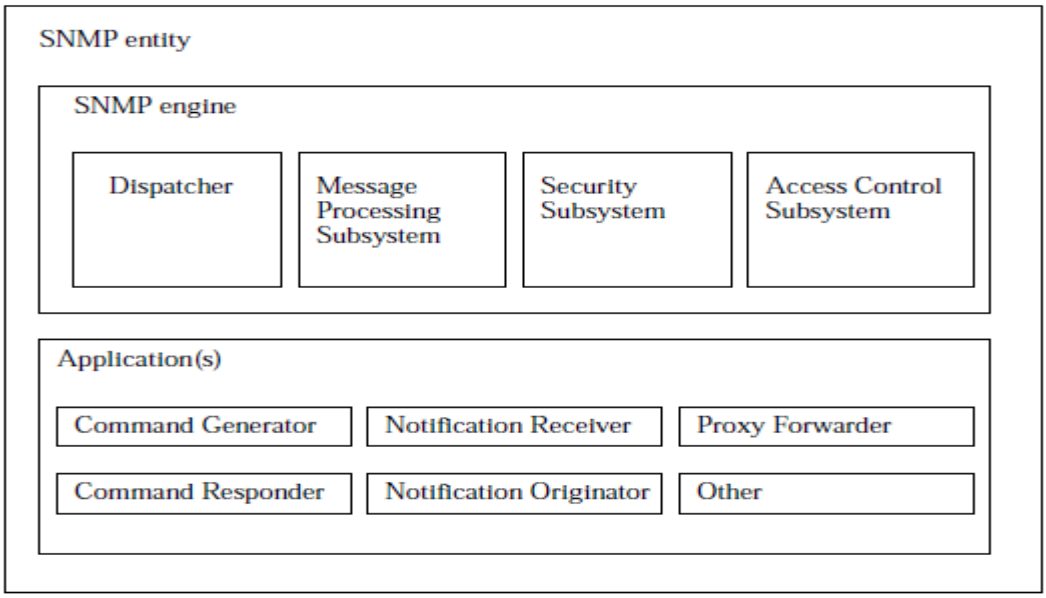


Fig.5.15 SNMPv3 Architecture

SNMP Engine

As you can see from the above diagram, an SNMP engine is made up of the following components:

• Dispatcher

• Message Processing Subsystem

• Security Subsystem

• Access Control Subsystem

Dispatcher

The Dispatcher is responsible for sending and receiving messages. When a message is received, the Dispatcher tries to determine the version number of the message and then passes the message to the appropriate Message Processing Model. If the message cannot be parsed so that the version can be determined, then the snmpInASNParseErrs counter is incremented and the message is discarded. If the version is not supported by the Message Processing Subsystem, then the snmpInBadVersions counter is incremented and the message is discarded. The dispatcher is also responsible for dispatching PDUs to applications, and for selecting the appropriate transports for sending messages.

Message Processing Subsystem

- The Message Processing Subsystem is made up of one or more Message Processing Models.

The following diagram shows a Message Processing Subsystem that supports models for SNMPv3, SNMPv1, SNMPv2c, and something that we will call “Other.”

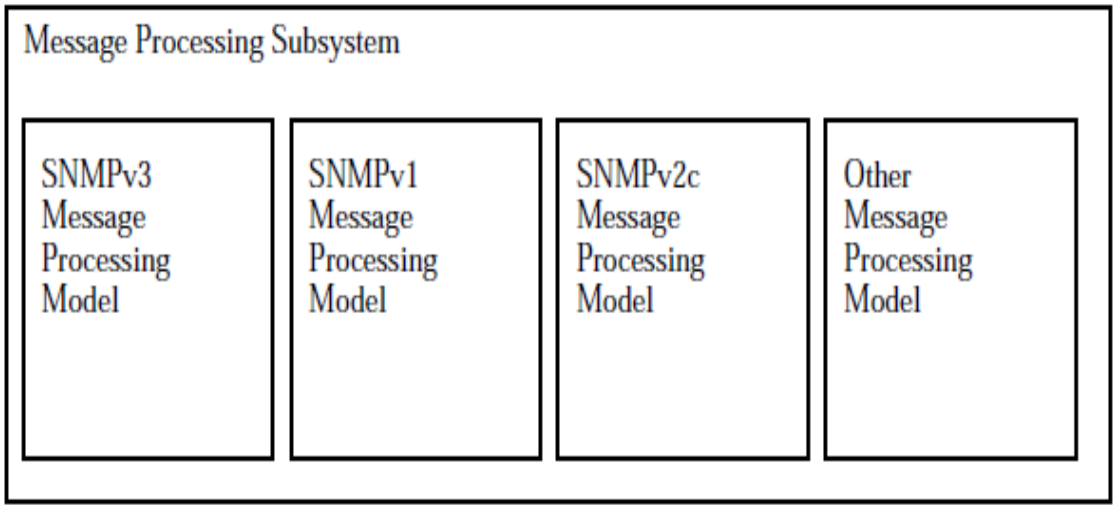


Fig. 5.16 Message Processing Sub system

The Message Processing Subsystem is responsible for

1. Preparing messages to be sent.
2. Extracting data from received messages.

Let’s walk through a simple case where the Dispatcher receives a valid SNMPv3 message from the line. The Dispatcher determines the version of the message and forwards it to the SNMPv3 Message Processing Model. The SNMPv3 Message Processing Model then processes the message by extracting information from it. It then calls the Security Subsystem to decrypt the data portion of the message (if needed) and make sure the message is properly authenticated.

At that point the Dispatcher will forward the PDU portion of the message to the appropriate SNMP application (more about that later).

This architecture allows additional models (like “Other”) to be added. These additional models may be enterprise specific or future standards. In any case, the Dispatcher will need to be able to parse the messages to determine the version (and then map the version number to a Message Processing Model).

• An unauthorized user trying to masquerade as an authorized user. For example, someone might try to perform management operations (such as change the operational state of a port) that they don’t have authorization for by pretending to be an authorized user.

• Modifying the message stream. SNMP is typically based on UDP, which is a connectionless transport service. Messages could potentially be captured and reordered, delayed, or possibly replayed at a later time. For example, if a Set operation were captured and
replayed in the future, it could conceivably change the desired configuration. By checking the timeliness of messages, this threat can be minimized.

• Eavesdropping. By allowing messages to be encrypted, someone eavesdropping on the line won’t be able to make sense of what they see. This feature is essential for carriers that
need to protect against sensitive data, such as billing information, from being eavesdropped on.

The User-Based Security model currently defines the use of HMAC-MD5-96 and HMACSHA- 96 as the possible authentication protocols and CBC-DES as the privacy protocol. Future authentication and privacy protocols may be added.

SNMPv1 and SNMPv2c Security Models provide only weak authentication (community names) and no privacy.

This architecture allows additional Security Models (like “Other”) to be added. These
additional models may be enterprise specific or future standards. Authentication and privacy protocols supported by Security Models are uniquely identified using Object Identifiers. Any IETF standard protocols for authentication should have an identifier defined within the snmpAuthProtocols subtree. Any IETF standard protocols for privacy should have an identifier defined within the snmpPrivProtocols subtree. Enterprise specific protocols should have their identifiers defined within the enterprise subtree.

Applications

For SNMPv3, when we refer to applications, we are referring to internal applications within an SNMP entity as opposed to what you might normally think of, such as a network management application to do trending or configuration. These internal applications do things like
generate SNMP messages, respond to received SNMP messages, generate notifications, receive notifications, and forward messages between SNMP entitites. Currently there are five types of applications defined:

1. Command Generators — generate SNMP commands to collect or set management data.

2. Command Responders — provide access to management data. For example, processing Get, Get-Next, Get-Bulk and Set PDUs are done by a Command Responder application.

3. Notification Originators — initiate Trap or Inform messages.

4. Notification Receivers — receive and process Trap or Inform messages.

5. Proxy Forwarders — forward messages between SNMP entities.

The SNMPv3 Framework allows other applications to be defined over time. From this list, you can see that Command Generators and Notification Receivers are what we used to think of as part of an SNMP Manager, while Command Responders and Notification Originators are what we used to think of as part of an SNMP Agent.

4.2.3 Snmp Message Processing Model

This type is used to identify the message processing model used to process an SNMP message.

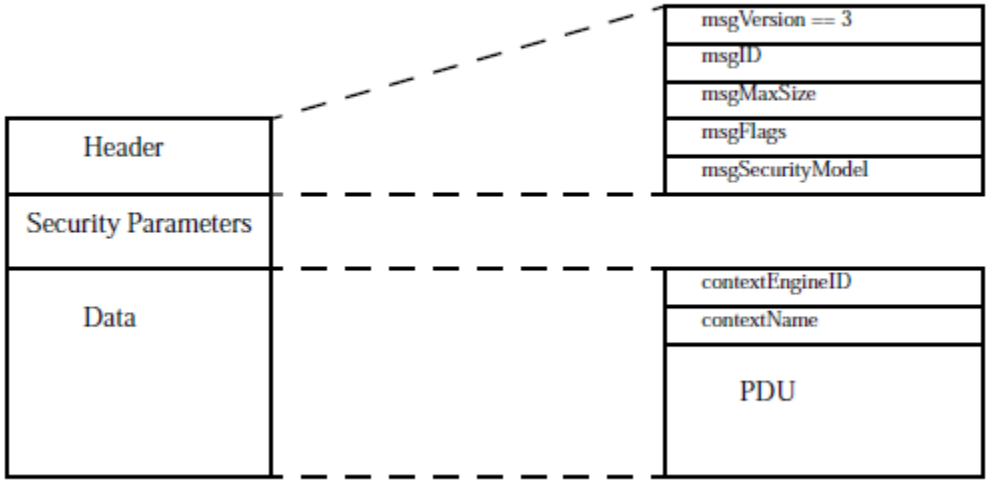
It resolves to an INTEGER and can have one of the following values:

- 0, SNMPv1.
- 1, SNMPv2c.
- 2, SNMPv2u and SNMPv2*.
- 3, SNMPv3.
- 4 - 255, reserved for standards-track message processing models. These values will be managed by the Internet Assigned Numbers Authority (IANA).
- Values greater than 255 are handled exactly the same way as with the SnmpSecurityModel type to allow enterprise-specific message processing models. An enterprise- specific message processing model can be defined as enterpriseNumber * 256 + messageProcessingModel

Again, as with the security model example, since Cisco's enterprise number is 9, Cisco could define enterprise-specific message processing models with identifiers in the range of 2304 through 2559. And as with the security model, this scheme allows enterprises to define up to 255 enterprise-specific message processing models.

SNMPv3 Message Format

A new format has been defined for SNMPv3 messages. An SNMPv3 message contains among other things an SNMPv2 PDU either encrypted or in plain text, security information, and the context the message should be processed in. The format for the message is



The header is made up of the following:

- msgVersion, a value of 3 identifies the version of the message as an SNMPv3 message.
- msgID (message identifier), this is an integer value that is used to coordinate request and response messages between two SNMP entities. The use of this is similar to the use of the request identifier within a PDU. The request identifier is used by SNMP applications to identify the PDU. The msgID is used by the engine to identify the message which carries a PDU.

Note: One of the security threats that SNMPv3 tries to protect against is where a valid message is captured and replayed later. By guaranteeing that msgID values are not reused and that each message is identified by a unique value, this threat can be eliminated. One possible implementation to generate unique msgID values is to use the low-order bits of snmpEngineBoots as the high-order portion of the msgID value and a counter value for the low-order portion of msgID. This will protect against an SNMP entity generating the same msgID value after a device reboots. It will also guarantee that msgID values won't repeat until after 65,535 messages (216-1) have been generated.

- msgMaxSize (maximum message size), an integer value which indicates the maximum message size that the sender can support. This value is used to determine how big a response to a request message can be. This can have values ranging from 484 through 231-1.
- msgFlags (message flags), a 1-byte value that contains flags that indicate whether the message can cause a Report to be generated and the security level the sender had applied to the message before it was sent on the wire. The 3 bits defined are reportableFlag, authFlag, and the privFlag.

If the reportableFlag is set, then a Report PDU can be sent back to the original sender (more on Report PDUs later). All messages that can be responded to (such as a Get PDU or an Inform PDU) are automatically treated as if reportableFlag is set to

1. All messages that are unacknowledged (such as a Report PDU, a Response PDU, or an SNMPv2-trap PDU) are automatically treated as if reportableFlag is set to 0.

The reportableFlag is only used if the PDU portion of a message cannot be decoded, for example, if a PDU cannot be decrypted because of an invalid encryption key.

The authFlag and privFlag are used to indicate the security level. This can indicate the message was sent with no authentication and no privacy, authentication and no privacy, or authentication and privacy. The receiver of the message must apply this same security level when the contents are processed.

- msgSecurityModel (message security model), an integer value which identifies the message security model that the sender used to generate this message. The receiver, obviously, must use the same security model to perform security processing for the message. The possible values for this are defined by the SnmpSecurityModel type.

Since enterprise-specific security models may be implemented, the mapping of this value to the desired security model within an SNMP engine may need to be done in an implementation-dependent way.

SNMPv3 MIB Views

The SNMPv3 protocol allows you to configure MIB views for users and groups. The MIB tree is defined by RFC 1155 (Structure of Management Information).

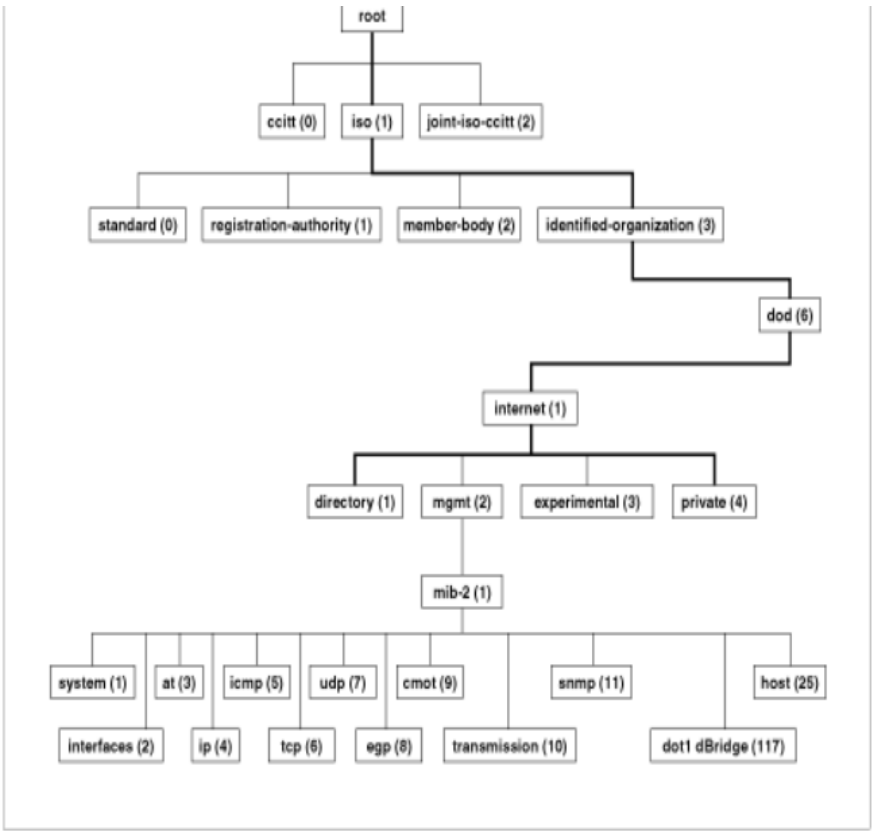


Fig.5.18 MIB Tree

You can define a MIB view that the user can access or a MIB view that the user cannot access. When you want to permit a user to access a MIB view, you include a particular view. When you want to deny a user access to a MIB view, you exclude a particular view.

After you specify a MIB subtree view you have the option of further restricting a view by defining a subtree mask. The relationship between a MIB subtree view and a subtree mask is analogous to the relationship between an IP address and a subnet mask. The switch uses the subnet mask to determine which portion of an IP address represents the network address and which portion represents the node address. In a similar way, the subtree mask further refines the subtree view and enables you to restrict a MIB view to a specific row of the OID MIB table. You need a thorough understanding of the OID MIB table to define a subtree mask.