UNIT 3 SOCKET PROGRAMMING

Structure **Page Nos.**

3.0 Introduction 42
3.1 Objectives 42
3.2 Advance System call 42
  3.2.1 Data Transfer
  3.2.2 Byte Operations and Addressing
  3.2.3 Socket Options
  3.2.4 Select System Call
3.3 Raw Socket 53
3.4 Multiple Recipients 55
  3.4.1 Unicasting
  3.4.2 Broadcasting
  3.4.3 Multicasting
3.5 Quality of Service Issues 56
3.6 Summary 58
3.7 Solutions/Answers 58
3.8 Further Readings 58

3.0 INTRODUCTION

In the previous unit we have discussed different elementary system calls required for connection establishment, termination and data transfer in TCP and UDP client server architecture. But because of the complexity and the requirements of network communication like, byte ordering problem, addressing schemes, multiple recipients communication and the need of personal protocols development we need many more system calls for network communication that we will discuss in this unit.

3.1 OBJECTIVES

Our objective is to introduce you the advanced socket calls, which will provide you the in-depth knowledge of the functions and their characteristics that are required for developing network applications. After successful completion of this unit, you should be able to:

- have a reasonable understanding of the Advance System calls;
- understand the use of advance data transfer calls;
- describe the Byte Operations and Addressing functions;
- know the use and applications of socket options;
- understand the concept of raw sockets;
- demonstrate an understanding of the network programs for multiple recipients, and
- know the Quality of Service issues in TCP/IP.

3.2 ADVANCE SYSTEM CALL

This section will describe the syntax and semantics of some of the important system calls, organised by their functionalities. The syntax of these calls are given in C language, where the meaning and use of each system call is defined. Then the syntax with necessary parameters and header files are given. They usually return a negative value to indicate an error and a non-negative return value to indicate success. .

3.2.1 Data Transfer

readv(): The **readv()** function is used to input data from a socket in scatter mode when the input is to be noncontiguous.

```
#include <socket.h>
#include <uio.h>
int readv ( sockfd, buff, buffcnt)
int sockfd;
struct buffec *buff;
int buffcnt;
```

```
The buffec structure is defined as:
struct buffec
{
   caddr–t    buff_base;
   int        buff_len;
};
```

The readv() function can be called with either a socket or file descriptor. The readv() function is same as read(), but scatters the input data into the buffcnt buffers specified by the members of the buff array: buff[0], buff[1], ..., buff[buffcnt-1]. Each buffec entry specifies the ***base address*** and ***length*** of an area in memory where data should be placed. readv() function returns the total number of bytes read on success, 0 is returned if an end-of-file condition exits, otherwise, the value -1 is returned with error code.

writev(): The **writev()** function is used to output data from a socket in gather mode when the data are not contiguous.

```
#include <socket.h>
#include <uio.h>
int writev (sockfd, buff, buffcnt )
int sockfd;
struct buffec *buff;
int iovcnt;
```

writev() is also as write(), but gathers the output data from the buffcnt buffers specified by the members of the buff array: buff [0], buff [1], .., buff [buffcnt-1]. writev() function always writes a complete area before proceeding to the next buff entry. It returns the total number of bytes actually written on success, otherwise the value -1 is returned with error code.

recvmsg() : The recvmsg() is used to receive incoming data that has been queued for a connected or unconnected sockets.

```
#include <socket.h>
#include <uio.h>
int recvmsg ( sockfd, messg, flags )
int sockfd;
struct messghdr messg [ ];
int flags;
```

```
struct messghdr
{  caddr_t  .       messg_name;
   int              messg_namelen;
   struct buffec    *messg_buff;
   int              messg_bufflen;
   caddr_t          messg_accrights;
   int
messg_accrightslen;
};
```

Data is received in "scatter" mode and placed into noncontiguous buffers. The argument **messg** is the pointer to an object of byte structure, messghdr, used to minimize the number of directly supplied parameters. Here **messg_name** and **messg_namelen** specify the source.address if the socket is unconnected. The **messg_buff** and **messg_bufflen** describe the "scatter" locations, as described in **read()**. A buffer to receive any access rights sent along with the message is specified in **messg_accrights**, which has length **messg_accrightslen.** It returns the number of bytes received if successful. 0 is returned if an end-of-file condition exits, otherwise, the value -1 is retuned with error code.

sendmsg(): It is used to send outgoing data on a connected or unconnected socket.

```
#include <socket.h>
#include <uio.h>
int sendmsg ( sockfd, messg, flags )
int sockfd;
struct messghdr messg [ ];
int flags;
```

Data is sent in gather mode from a list of noncontiguous buffers. The argument messg is a pointer to an object of byte structure messghdr. This structure is Sam as defined in recvmsg() above. If successful, it returns the number of bytes sent. Otherwise, the value -1 is returned with error code.

3.2.2 Byte Operations and Addressing

Byte Order Conversions: Different types of computers can store data in different ways. A common data format was decided upon for communication between hosts via TCP/IP. The following functions convert data from the host format to the network format, or vice-versa. Before going into the details of the data conversion first we can understand the use of these functions.

Function Name	Descriptions
htons	"host to network, short", for converting a two-byte integer from host order format to network order
htonl	"host to network, long", for converting a four-byte integer from host order to network order
ntohs	"network to host, short", for converting a two-byte integer from network order to host order
ntohl	"network to host, long", for converting a four-byte integer from network order to host order.

Note the pattern of the function names.

Operation	From format	"to"	To format	Data length
htonl	host	to	network	long(4byte)
htons	host	to	network	short(2byte)
ntohl	network	to	host	long(4byte)
ntohs	network	to	host	short(2byte)

There are two type of byte ordering:

- Big endian / host byte ordering: most significant bit first.
- Little endian / Network byte ordering: least significant bit first

In the first form, also called 'Big Endian', The Most Significant Byte (MSB) is kept in the lower address, while the Least significant Byte (LSB) is kept in the higher address. In the second form, also called 'Little Endian', the MSB is kept in the higher address, while the LSB is kept in the lower address. As we know (recall unit 5) different computers use different byte ordering (or different endianess), usually depending on the type of CPU they have. The same problem arises when using a long integer: which word (2 bytes) should be kept first in memory? the least significant word, or the most significant word?

In a network protocol, however, there must be a predetermined byte and word ordering. The IP protocol defines what is called 'the network byte order', which must be kept on all packets sent across the Internet. The programmer on a Unix machine is not saved from having to deal with this kind of information. We'll see how the translation of byte orders is solved when we get down to programming.

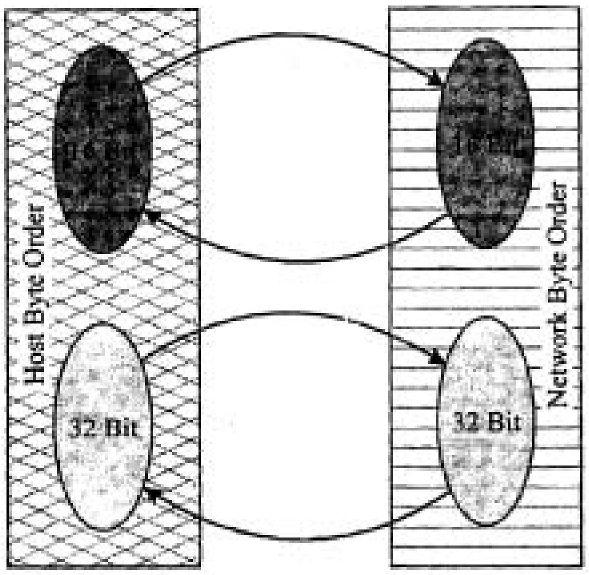


Figure 1: Byte Order Conversion

Here we will take one example to show you how to convert an integer before sending it through a socket and after reading how it **can** be coverted back into the original **form:**

1) **J=htonl(i); /*** converting a four-byte integer from host order to network order ***/**
2) **write_data(sockfd, &i, sizeof(i));** I* write data on the socket **sockfd*/**
3) **read_data(sockfd,** &i, **sizeof(i)); /*** read the data*/
4) **i=ntohl(i); /*** convert a four-byte integer from network order to host order ***/**

If your host order is different from network order and you fail to compensate for it, then your sockets will not work. The sockets API provides a number of conversion functions, including ones that will convert host-order values into network order and vice versa. The details of these functions for converting byte order are given in the following section:

htonl() : It converts 32-bit quantities from host **byte** order to network byte **order.**

```
#include <sys/types.h>
#include <inet.h>
unsigned long htonl ( hostlong );
unsigned long hostlong;
```

The **htonl()** function **is provided to** .maintain compatibility with application programs ported from other systems that byte-swap memory. These systems store 32-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols. If successful, returns the converted value.

htons() : It converts 16-bit quantities from host byte order to network byte order.

```
#include <sys/types.h>
#include <inet.h>
unsigned short htons ( hostshort );
unsigned short hostshort;
```

The **htons()** function is provided to maintain compatibility with application programs ported from other systems that byte-swap memory. These systems store 16-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols. If successful, returns the converted value.

ntohl() : It converts 32-bit quantities from network byte order to host byte order.

```
#include <sys/types.h>
#include <inet/in.h>
unsigned long ntohl( netlong );
unsigned long netlong;
```

ntohl is portable to other environments, including most UNIX systems, that implement BSD sockets. If successful, returns the converted value.

ntohs() : It converts 16-bit quantities from network byte order to host byte order.

```
#include <sys/types.h>
#include <inet/in.h>
unsigned short ntohs(netshort);
unsigned short netshort;
```

Ntohs () is portable to other environments, including most UNIX systems, that implement BSD sockets. If successful, returns the converted value.

Address Conversion

An Internet address is usually written and specified in the dotted-decimal notation. Internally it becomes part of a structure of type in–addr. To convert between these two representations functions are available. In the following section we will discuss these address conversion function in details.

- inet_aton: ASCII to number (string to binary)
- inet_ntoa: number to ASCII (binary to string)
- inet_addr: like inet_aton.

inet() : The inet functions are a set of routines that construct Internet addresses or break Internet addresses down into their component parts. Routines that convert between the binary and ASCII ("dot" notation) form of Internet addresses are included. Here, in the given sections, these functions are briefly defined.
unsigned long **inet_addr** (cp) : inet_addr() interpret character strings (char *cp) representing numbers expressed in the Internet standard dotted (".") notation, returning numbers suitable for use as Internet addresses.

unsigned long **inet_network** (cp) : inet_network() interpret character strings representing numbers expressed in the Internet standard dotted (".") notation, returning numbers suitable for use as network numbers.

char **inet_ntoa ()** : inet_ntoa() takes an Internet address and returns an ASCII string representing the address in dot notation.

unsigned long **inet_makeaddr** (net, **lna)** : inet_makeaddr() takes an Internet network number(net) and a local network address (Ina) or host number and constructs an Internet address from it.

int **inet_lnaof** () : inet_netof() break apart Internet host addresses, returning the network number.

int **inet_netof ()** : inet_lnaof() break apart Internet host addresses, returning the local network.

You must remember that all the Internet addresses are returned in network byte order and all network numbers and local address parts are returned as integer values in host byte order. Values specified using the Internet standard dotted notation take one of these forms:

1) **a.b.c.d** → If 4 parts are specified, each part is interpreted as a byte of data and assigned, from **left** to right, to the four bytes of an Internet address.

2) **a.b.c** → In this case the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as **128.net.host.**

3) a.b → In this case when two parts are specified the last part is 'interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as **net.host.**

4) a → When only one part is given, the value is stored directly in the network address without any byte rearrangement.

The **inet_addr()** function returns an lnternet address if successful, or a value of -1 if the request was unsuccessful. The **inet_network()** function returns a network number if successful, or a value of -1 if the request was malformed.

inet_aton()

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>
int inet_aton( cp , in)
const char    *cp;
struct in–addr    *in;
```

As you know **inet_aton()** converts an ASCII representation of an lnternet address to its network internet address. **inet_aton()** interprets a character string representing numbers expressed in the lnternet standard. notation, returning a number suitable for use as an lnternet address.

inet_ntoa()

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>
char *inet_ntoa ( in )
const struct in–addr in;
```

The **inet_ntoa()** converts an Internet network address to its **ASCII "d.d.d.d"** representation. **inet_ntoa()** returns a pointer to a string in the base 256 notation **"d.d.d.d"**. lnternet addresses are returned in network byte order (bytes ordered from left to right). **inet_ntoa()** points to a buffer which is overwritten on each call.

Get useful information

There are number of situations when we need to obtain the official name of a host and we should know about its network address or vice versa. Similarly, sometime we need the name of the local host, the port number associated with a service or the name assigned to a socket there may be many more other cases when we want to get some information of client or server using some known information for this purpose. We have lots of get functions in our socket library which you can use accordingly. Here we have given the details of some get* system call description.

gethostbyaddr(): It is used to obtain the official name of a host when its network address is known.

```
#include <netdb.h>
struct hostent *gethostbyaddr ( addr, len, type )
char *addr;
int len, type;
```

A name server is used to resolve the address if one is available; otherwise, the name is obtained (if possible) from a database maintained on the local system. gethostbyaddr() returns a pointer to an object with the structure hostent.

```
struct hostent
{
char *h_name;
char **h_aliases;
int h_addrtype;
int h_length;
char **h_addr_list;
}; #define h_addr h_addr_list [0]
```

Parameters:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A zero-terminated array of network addresses for the host.
h_addr	The first address in h_addr_list; this is for backward compatibility.

The gethostbyaddr() function returns a pointer to the hostent structure, if successful. If unsuccessful, a null pointer is returned, and the external integer h_errno is set to indicate the nature of the error.

gethostbyname() : It is used to obtain the network address or list of addresses of a host when its official name is known.

```
#include <netdb.h>
struct hostent *gethostbyname (name)
char *name;
```

A name server is used to resolve the name if one is available; otherwise, the address is obtained (if possible) from a database maintained on the local system. The gethostbyname() function returns a pointer to an object with the structure hostent as defined above. The gethostbyname() function returns a pointer to the hostent structure, if successful. If unsuccessful, a null pointer is returned, and the external integer h_errno is set to indicate the nature of the error.

gethostname() : It is used to obtain the name of the local host.

```
#include <socket.h>
#include <uio.h>
int gethostname ( name, namelen )
char *name;
int namelen;
```

Here name is the *official* name by which other hosts in the communications domain reference it. Generally, a host belonging to multiple communication domains, or connected to multiple networks within a single domain, has one **official** name by which it is known. The **gethostname()** function returns the standard host name for the local system. The parameter namelen specifies the size of the name array. If **gethostname()** is **successful,** a value of 0 is returned. A return value of -1 indicates an error.

getservbyname() : It is used to obtain the port number associated with a service when its **official** name is known.

```
#include <netdb.h>
struct servent *getservbyname ( name, proto )
char *name, *proto;
```

This information is acquired from a service database maintained by the local system. **getservbyname()** returns a pointer to an object with this given structure:

```
struct servent
{
   char     *s_name;
   char     **s_aliases;
   int     s_port;
   char     *s_proto;
};
```

Parameters:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
sgort	The port number at which the service resides.
sgroto	The name of the protocol to use when contacting the service.

The **getservbyname()** function sequentially searches from the beginning of the network service database until a matching service **name** is found, or until the end of the database is reached. It returns a pointer to the servent structure, if successful. If unsuccessful, a null pointer is returned.

getsockname() : It obtains the name assigned to a socket, which is the address of the local endpoint and was assigned with a bind() function.

```
#include <socket.h>
#include <uio.h>
int getsockname ( sockfd, name, namelen )
int sockfd;
struct sockaddr *name;
int *namelen;
```

getsockname() returns the current name for the specified socket. The namelen parameter should be initialized to indicate the amount of space pointed to by name. On return, it contains the actual size of the name returned (in bytes). If **getsockname()** is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer **errno** indicates the nature of the error.

Check Your Progress 1

1) readv() & writev() functions are used when data are?

a) Synchronous

b) Asynchronous

c) Contiguous

d) Non-Contiguous

....

2) Which of the following function is used for converting a two-byte integer from network order to host order? .

a) h tons()
b) h tonl()
c) n tons()
d) n tohl()

....

3) When we want to know the official name of a host and its network address is known, which of the following function is used?

a) getservby name
b) getsockname()
c) gethostbyname()
d) gethostbyaddr()

....

3.2.3 Socket Options

It is possible to set and get a number of options on sockets via the ***setsockopt*** and ***getsockopt*** system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

setsockopt(sockfd, level, optname, optval, optlen);

and

getsockopt(sockfd, level, optname, optval, optlen);

The parameters to calls are as follows: ***sockfd*** is the socket on which the option is to be applied. ***Level*** specifies the protocol layer on which the option is to be applied; The actual option is specified in ***optname. Optval*** and ***Optlen*** point to the value of the option and the length of the value of the option, respectively. For ***getsockopt, optlen*** is a value-result parameter, initially set to the size of the storage area pointed to by ***optval,*** and modified upon return to indicate the actual amount of storage used. An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs described below will perform this task.

```
#include <sys/types.h>
#include <sys/socket.h>
int type, size;
size = sizeof (int);
if (getsockopt(sockfd, SOL–SOCKET, SO–TYPE, (char *).
&type, &size) < 0) {
```

SO–TYPE get the socket option after the getsockopt call, type will be set to the value of the socket type. Because the socket were a datagram socket, ***type*** would have the value corresponding to SOCK–DGRAM.

Lets see the details of **getsockopt()** and **setsockopt()** system calls in the following section. Then you will understand the above example in more detail:

setsockopt() : The **setsockopt()** function is used to manipulate options associated with a socket.

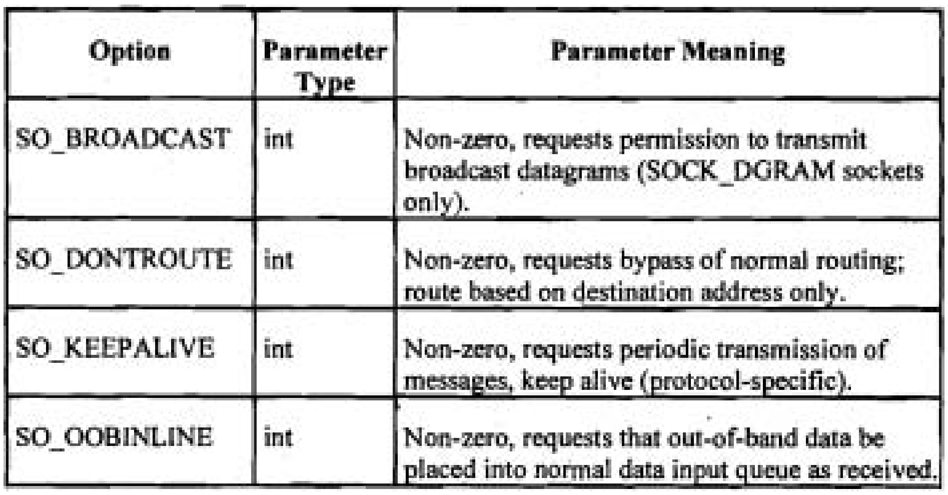
```
#include <socket.h>
#include <uio.h>
#include <inet.h>
#include <tcp.h>
int setsockopt ( sockfd, level, optnarne, optval, optlen )
int sockfd, level, optnarne;
char *optval;
int optlen;
```

Options can exist at multiple protocol levels; **they** are always present at the uppermost socket level. Remember when you are manipulating socket options, the level at which the option resides and the name of the option must be specified properly. level is specified as SOL–SOCKET to manipulate options at the socket level. To manipulate options at any other level, the protocol number of the appropriate protocol is supplied. For explaining it, let's take an example, the case for changing TCP protocol option the following parameters of **setsockopt()** should be passed as given below:

level	set to the **protocol** number of TCP.
optval	identify a buffer that contains the option value.
optlen	length of the option value in bytes.
optnarne	parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation.

Options at **other protocol** levels vary in format and name. These options are recognized at the socket level. Each can be set with **setsockopt()** and examined with getsockopt.

There are a number of socket options which either specialise the behaviour of a socket or provide useful information. These options may be set at different protocol levels and are always present at the **uppermost** "socket" level. These options (some of these options are given below) allow an application program to customize the behaviour and characteristics of a socket to provide the desired effect.



Option	Parameter Type	Parameter Meaning
SO_BROADCAST	int	Non-zero, requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only).
SO_DONTROUTE	int	Non-zero, requests bypass of normal routing; route based on destination address only.
SO_KEEPALIVE	int	Non-zero, requests periodic transmission of messages, keep alive (protocol-specific).
SO_OOBINLINE	int	Non-zero, requests that out-of-band data be placed into normal data input queue as received.

getsockopt(): The **getsockopt()** function is used to retrieve options currently associated with a socket.

```
#include <socket.h>
#include <uio.h>
int getsockopt (sockfd, level, optname,
optval, optlen)
int sockfd, level, optname;
char *optval;
int *optlen;
```

When retrieving socket options, the level at which the option resides and the name of the option must be specified. In continuation of the previous example to indicate that an option is to be returned by the TCP protocol the following parameters of **getsockopt()** should be indicated as given below:

Level	set to the protocol number of **TCP**.
Optval	identify a buffer in which the value for the requested option is to be returned.
Optlen	It initially contains the size of the buffer pointedto by the optval and modified on return to indicate the actual size of the value returned. If no option value is to be returned, optval can be supplied as 0.
optname	is passed uninterpreted to the appropriate protocol module for interpretation.

3.2.4 Select System Call

The select() system call is used to synchronise processing of several sockets operating in non-blocking mode. When an application calls recv or recvfrom it is blocked until data arrives for that socket. While the incoming data stream is empty program can do some other job or the situation when a program receives data from multiple sockets. The select function call solves this problem by allowing the program to choose all the socket handles to see if they are available for non-blocking reading and writing operations.

```
#include <socket.h>
#include <uio.h>
int select ( nfds, readfds, writefds, exceptfds, timeout )
int nfds;
fd–set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
int fd;
fd–set fdset;
FD_SET ( fd, &fdset ) /* Includes a particular descriptor fd in fdset. */
FD_CLR ( fd, &fdset ) /* Removes fd from fdset. */
FD_ISSET ( fd, &fdset ) /* non-zero if fd is a member of fdset, zero
otherwise */
FD_ZERO ( &fdset ) I* Initializes a descriptor set fdset to the null set. */
int fd;
fd–set fdset;
```

Sockets that are ready for reading, ready for writing, or have a pending exceptional condition can be selected. If no sockets are ready for processing, the select() function can block indefinitely or wait for a specified period of time (which may be zero) and then return. Select() examines the I/O descriptor sets whose addresses are passed in readfds, Writefds, and exceptfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfds descriptors are checked in each set (in other words, the descriptors from 0 through nfds-1 in the descriptor sets are examined). readfds - A pointer to a set of file and socket descriptors that are to be polled for non-blocking reading and writing operations. writefds, exceptfds are same as readfds, except these sets cdntain the file/socket handles to poll for non-blocking writing operations and error detection. On return, select() replaces the given descriptor sets with subsets consisting of those clescriptors that are ready for the requested operation. The total number of ready clescriptors in all the sets is returned.

If timeout is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a zero pointer, the select blocks indefinitely. To affect a poll, the timeout argument should be non-zero, pointing to a zero-valued timeval structure.

If successful, select() returns the number of ready descriptors that are contained in the descriptor sets, or 0 if the time limit expires. Otherwise, the value -1 is returned, and the error code.

Example:

```
fd_set fdset;
struct timeval tv;         // sock is an intialized socket handle
tv.tv_sec = 2;
tv.tv_usec = 500000;  // tv now represents 2.5 seconds
FD_ZERO(&fdset);
FD_SET(sock, &fdset); // adds sock to the file descriptor set
/* wait 2.5 seconds for any data to be read from any single socket */
select(sock+1, &fdset, NULL, NULL, &tv);
if (FD_ISSET(sock, &fdset))
recvfrom(s, buffer, buffer–len, 0, &sa, &sa_len);
else
/* do some other work */
```

3.3 RAW SOCKET

Raw sockets are those sockets, which offer the programmer the possibility to have absolute control over the data, being sent or received through the network. They are very useful when someone needs to create his own protocol but can you think what is a need of creating special protocols? You know with raw sockets you can have your own encrypted traffic tunnels which will enhance your security, you can also optimize the voice and video conference protocols, which may increase the quality and the performance of the sessions.

In this section, you will learn the basics of using raw sockets to insert an IP protocol based datagram into the network traffic. You will learn how to build raw socket scanners or how to perform operations that need to send out raw sockets. Basically, you can send any packet at any time, whereas using the interface functions for your systems IP-stack (connect, write, bind, etc.).

The basic concept of low-level sockets is to send a single packet at one time, with all the protocol headers filled in by the program. Unix provides two kinds of sockets that permit direct access to the network.

- SOCK–PACKET
- SOCK–RAW

SOCK–PACKET receives and sends data on the device link layer. This means, the NIC specificheader is included in the data that will be written or read. For most networks, this is the ethernet header. Of course, all subsequent protocol headers will also be included in the data.

The socket type we are going to use, is SOCK–RAW, which includes the IP headers and all subsequent protocol headers and data. A standard command to create a datagram socket is:

```
socket (PF_INET, SOCK–RAW, IPPROTO_UDP);
```

The moment it is created, you can send any IP packets over it, and receive an IP packets.

Note that even though the socket is an interface to the IP header, it is transport layer specific. That means, for listening to TCP, UDP and ICMP traffic, you have to create 3 separate raw sockets, using IPPROTO–TCP, IPPROTO_UDP and IPPROTO_ICMP.

With this knowledge, we can, for example, already create a small sniffer, that dumps out the contents of all tcp packets we receive.

As you see, we are skipping the IP and TCP headers which are contained in the packet, and print out the payload, the data of the session/application layer, only.

```
int fd = socket (PF_INET, SOCK–RAW, IPPROTO–TCP);
char buffer[8192]; /* single packets are usually not bigger than 8192 bytes */
while (read (fd, buffer, 8192) > 0)
printf ("Caught tcp packet: %s\n",
buffer+sizeof(struct iphdr)+sizeof(struct tcphdr));
```

To inject your own packets, all you need to know is the structures of the protocols that need to be included. It is recommended to build your packet by using a struct, so you can comfortably fill in the packet headers. Unix systems provide standard structures in the header files (eg. <netinet/ip.h>). You can always create your own structs, as long as the length of each option is correct. Now, by putting together the knowledge about the protocol header structures with some basic C functions, it is easy to construct and send any datagram(s). To make use of raw packets, knowledge of the basic IP stack operations is essential.

Check Your Progress 2

1) How Many parameters are in setsockopt () system call?
   a) three
   b) four
   c) five
   d) two

....

....

....

....

2) Which of the following option should be used for by passing the normal routing?
   a) SO–KEEPALIVE
   b) SO–BYPASS
   c) SO_BYPASSROUTER
   d) SO_DONTROUTE

....

....

....

....

3) Which of the following sockets has permission to directly access the network?

a) SOCK–STREAM
b) SOCK–DATAGRAM
c) SOCK–PACKET
d) SOCK–TCP

....

....

3.4 MULTIPLE RECIPIENTS

There are different ways of transmitting a message over a network, one way in which a data is sent from a single source to a specified destination is known as unicasting, another way is multicasting in which data is sent to a specified group of destinations and in broadcasting where data is sent to all connected destinations. Let's discuss these transmission ways in detail:

3.4.1 Unicasting

Unicast is the term used to describe communication where a piece of information is sent from one point to another point. In this case there is just one sender, and one receiver. In unicast transmission a packet is sent from a single source to a specified destination, is still the predominant form of transmission on LANs and within the Internet. For example in our earlier sections we have given the implementation of unicasting.

3.4.2 Broadcasting

Broadcast is the term which is very familiar to all of us, and it has a traditional meaning associated with TV and radio. It generally means as a transmission that can be received by everyone having the correct equipment. In this case there is just one sender, but the information is sent to all connected receivers. Broadcast transmission is supported by most of the LANs, for example the ARP (address resolution protocol) uses this to send an address resolution query to all computers on a LAN.

Sending Broadcasting message

To send a broadcast message, first of all a datagram socket should be created as given below:

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

Then the socket option should be set for allowing broadcasting:

```
int b_on = 1;
setsockopt(sockfd, SOL-SOCKET, SO-BROADCAST, &b_on, sizeof (on));
/*as we have studied earlier in the course*/
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(sockfd, (struct sockaddr *) &sin, sizeof (sin));
```

3.4.3 Multicasting

Multicasting is the term used to describe communication where a piece of information is sent from one point to a set of other points. In this case there is one sender, and the information is distributed to a set of receivers. One example of this we can see in email and chatting groups on the 'Internet.

IP multicast provides dynamic many-to-many connectivity between a set of senders (at least 1) and a group of receivers. The format of IP multicast packets are identical to that of unicast packets and is distinguished only by the use of a special class of destination address (class D IP address), which denotes a specific multicast group.

Most of the LAN's are able to support the multicast transmission mode. It is inherently supported by the shared LANs, because in that case all packets reach all network interface cards connected to the LAN.

Sending multicasting message

To send a multicast datagram, specify an IP multicast address in the range **224.0.0.0** to **239.255.255.255.** A socket option is available to override the default for subsequent transmissions from a given socket:

```
struct in–addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof (addr));
```

Where "addr" is the local IP address of the desired outgoing interface.
If amulticast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop,
sizeof(loop));
```

where loop is set to 0 to disable loopback, and set to 1 to enable loopback. This option improves performance for applications that may have no more than one instance on a single host (such as a router demon), by eliminating the overhead of receiving their own transmissions.

To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified (i.e., INADDR_ANY). To receive multicast datagrams sent to a particular group and port, bind the local port, the local address set to the multicast group address. Once bound to a multicast address, the socket cannot be used for sending data.

More than one process may bind to the same SOCK_DGRAM **UDP** port or the same multicast group and port if the ***bind*** call is preceded by:

```
int on = 1;
setsockopt(sock, SOL–SOCKET, SO–REUSEPORT, &on, sizeof(on));
```

All processes sharing the port must enable this option. Every incoming multicast or broadcast UDP datagram destined to the shared port is delivered to all sockets bound with the port. For backwards compatibility reasons, this does not apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port.

3.5 QUALITY OF SERVICE ISSUES

"Quality of service" has become a big buzzword. This term conveys about as much useful information about what the technology offers as being told that it is "high performance".

The Internet is designed on top af the Internet Protocol, a packet-switching technology that is designed to get packets from point "A" to point "B" in whatever way is most effective, without the user necessarily having any ability to know what route will be taken. In fact, some packets in the same data stream may be sent along different routes. Packets may be stored for a while before being forwarded to their destination, or even dropped and retransmitted.

For most applications, such as simple file or message transfers, this is perfectly fine. However, there are applications where this sort of service is simply of "too low quality". In these cases, the nature of how the data is delivered is more important than merely how fast it is, and there is a need for technologies or protocols that offer "quality of service". This general term can encompass a number of related features; common ones include the following:

- Bandwidth Reservation: The ability to reserve a portion of bandwidth in a network or interface for a period of tirne, so that two devices can count on having that bandwidth for a particular operation. This is used for multimedia applications where data must be streamed in real-time and packet rerouting and retransmission would result in problems. This is also called resource reservation.
- Latency Management: A feature that limits the latency in any data transfer between two devices to a known value.
- Traffic Prioritization: In conventional networks, "all packets are created equal". A useful QoS feature is the ability to handle packets so that more important connections receive priority over less important one.
- Traffic Shaping: This refers to the use of buffers and limits that restrict traffic across a connection to be within a pre-determined maximum.
- Network Congestion Avoidance: This QoS feature refers to monitoring particular connections in a network, and rerouting data when a particular part of the network is becoming congested.

So, in essence, quality of service in the networking context is analogous to quality of service in the "real world". Some applications, especially multimedia such as voice, music and video, are time-dependent and require a constant flow of information more than the raw bandwidth.

Key Concept: The generic term quality of service describes the characteristics of how data is transmitted between devices, rather than just how quickly it is sent. Quality of service features seek to provide more predictable streams of data rather than simply faster ones. Examples of such features include bandwidth reservation, latency minimums, traffic prioritization and shaping, and congestion limitation. Quality of service is more important for special applications such as multimedia than for routine applications such as those that transfer files or messages.

To support quaiity of service requirements, many newer technologies have been developed or enhanced to add quality of service features to them. This includes the ability to support isochronous transmissions, where devices can reserve a specific amount of bandwidth over time to support applications that must send data in real time.

Check Your Progress 3

1) The transmission in which data is sent to a specific group of destination is called.

   a) Unicasting
   a) Multicasting
   b) Broadcasting
   c) All the above

   ....
   ....
   ....

2) Which of the following protocol uses the transmission in local area network?

   .a) ARP
   b) BRP
   c) TCP
   d) UDP

   ....
   ....
   ....

3) Which of the following condition is applied to disable the loop back in setsockopt () system call.

a) loop is set to 1
b) loop is set to 0
c) loop is set to 2
d) loop is set to 1

....

....

3.6 SUMMARY

In this unit we have discussed different practical issues about socket programming. In the first section we have covered the advance system call to provide you enhanced knowledge about socket programming. The address conversion and byte ordering routines were discussed with practical syntax of these routines. To help you for designing your own protocols raw sockets were discussed with their use in network programming. We have also covered the techniques to send data to all the connected computers or some group of computer5 in a network. In the end of this unit we have compared the services of TCP/IP protccols, which will definitely help you in TCP/IP programming. In the next block *Lab Manual* we have given you some exercises in the lab sessions based on these system calls.

3.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) D
2) C
3) D

Check Your Progress 2

1) C
2) D
3) C

Check Your Progress 3

1) B
2) A
3) B

3.8 FURTHER READINGS

1) Andrew S. Tanenbaum, ***Computer Networks***, Third edition.

2) Behrouz A. Forouzan, ***Data Communications*** and ***Networking***, Third edition.

3) Douglas E. Comer, ***Internetworking*** with ***TCP/IP Vol.1: Principles, Protocols, and Architecture*** (4th Edition).

4) William Stallings, Data ***and Computer Communications***, Seventh Edition.

5) W. Richard Stevens, The ***Protocols (TCP/IP Illustrated, Volume 1)***.

6) W. Richard Stevens, ***"UNIX Network Programming"***, Prentice Hall.