**UNIT 4**

##### Sockets

This chapter deals with sockets, which is the most fundamental technologies of computer networking. Most operating systems provide precompiled programs that communicate across a network. To understand some structs into this subject is necessary a deeper knowledge about the operating system and his networking protocols. This subject can be used as either beginners programmers or as a reference for experienced programmers Iis a case study, which describes in detail about how to implement Socket programs using TCP and UDP socket types.

**4.1. Socket Definition**

A socket is one end of a two-way communications link between two programs running on the network. It’s an abstraction, which is used to send and receive data 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

The chief difference between file descriptors and socket descriptors is that the operating system binds a file descriptor to a specific file when the application calls open, but it can create sockets without binding them to specific destination addresses.

Sockets perform exactly like UNIX files, so they can be used with traditional operations like read and write.

**4.2 IP Addressing**

An IP (Internet Protocol) address is a unique identifier for a node or host connection on an IP network. An IP address is a 32 bit binary number usually represented as 4 decimal values, each representing 8 bits, in the range 0 to 255 (known as octets) separated by decimal points. This is known as "dotted decimal" notation.

Example: 140.179.220.200

It is sometimes useful to view the values in their binary form.

140 .179 .220 .200

10001100.10110011.11011100.11001000

Every IP address consists of two parts, one identifying the network and one identifying the node. The Class of the address and the subnet mask determine which part belongs to the network address and which part belongs to the node address.

## 4.2.1 *Address Classes*

There are 5 different address classes. You can determine which class any IP address is in by examining the first 4 bits of the IP address.

* **Class A** addresses begin with **0xxx**, or **1 to 126** decimal.
* **Class B** addresses begin with **10xx**, or **128 to 191** decimal.
* **Class C** addresses begin with **110x**, or **192 to 223** decimal.
* **Class D** addresses begin with **1110**, or **224 to 239** decimal.
* **Class E** addresses begin with **1111**, or **240 to 254** decimal.

Addresses beginning with **01111111**, or **127** decimal, are reserved for loopback and for internal testing on a local machine. [You can test this: you should always be able to ping **127.0.0.1**, which points to yourself] Class D addresses are reserved for multicasting. Class E addresses are reserved for future use. They should not be used for host addresses.

**Note:** The loopback address is used to communicate between client and server where both are running in same node.

**4.3 Socket Address Structures**

Sockets use the sockaddr address structure to pass and to receive addresses. This structure does not require the socket API to recognize the addressing format.

Most socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begin with sockaddr\_ and end with a unique suffix for each protocol suite.

# *4.3.1 IPv4 Socket address structure*

An IPv4 socket address structure is commonly called an “Internet socket address structure,” is named *sockaddr\_in* and is defined by including the *<netinet/in.h>* header.

The POSIX (Portable Operating System for Unix Interface) definition is:

struct in\_addr

{

in\_addr\_t s\_addr; /\* 32-bit IPv4 address\*/

/\* network byte ordered\*/

};

struct sockaddr\_in {

uint8\_t sin\_len; /\* length of structure (16) \*/

sa\_family\_t sin\_family; /\* AF\_INET \*/

in\_port\_t sin\_port; /\* 16-bit TCP or UDP port number\*/

/\* network byte ordered \*/

struct in\_addr sin\_addr; /\* 32-bit IPv4 address \*/

/\* network byte ordered \*/

char sin\_zero[8]; /\* unused \*/

};

The datatypes required by the POSIX specification are:

Datatype Description Header

int8\_t Signed 8-bit integer <sys/types.h>

uint8\_t Unsigned 8-bit integer <sys/types.h>

int16\_t Signed 16-bit integer <sys/types.h>

uint16\_t Unsigned 16-bit integer <sys/types.h>

int32\_t Signed 32-bit integer <sys/types.h>

uint32\_t Unsigned 32-bit integer <sys/types.h>

sa\_family\_t Address family of socket address structure <sys/socket.h>

socklen\_t Length of socket address structure <sys/socket.h>

in\_addr\_t IPv4 address <netinet/in.h>

in\_port\_t TCP or UDP port <netinet/in.h>

***4.3.2 Generic socket address structure***

struct sockaddr

{

uint8\_t sa\_len;

sa\_family\_t sa\_family; /\* address family: AF\_xxx value \*/

char sa\_data[14]; /\* protocol-specific address \*/

};

***4.3.3. IPv6 socket address structure***

struct in6\_addr

{

uint8\_t s6\_addr[16]; /\* 128-bit IPv6 address \*/

/\* network byte ordered \*/

};

struct sockaddr\_in6

{

uint8\_t sin6\_len; /\* length of this struct (28) \*/

sa\_family\_t sin6\_family; /\* AF\_INET6 \*/

in\_port\_t sin6\_port; /\* transport layer port# \*/

/\* network byte ordered \*/

uint32\_t sin6\_flowinfo; /\* flow information, undefined \*/

struct in6\_addr sin6\_addr; /\* IPv6 address \*/

/\* network byte ordered \*/

uint32\_t sin6\_scope\_id; /\* set of interfaces for a scope \*/

};

New generic socket address structure:

struct sockaddr\_storage

{

uint8\_t ss\_len; /\* length of this struct (implementation dependent) \*/

sa\_family\_t ss\_family; /\* address family : AF\_xxx value \*/

/\* implementation dependent elements to provide:

\* a) alignment sufficient to fulfill the alignment requirements of all socket

\* address types that the system supports

\* b) enough storage to hold any type of socket address that the system supports

\*/};

**4.4 Byte Ordering Functions**

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

In a TCP segment, there is a 16-bit port number and a 32-bit IPv4 address. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields will be transmitted. The Internet protocols use big-endian byte ordering for these multibyte integers. This common byte order is called the *network byte order*.

To convert between host byte order and network byte order the following functions are used.

#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 host16bitvalue);

uint32\_t ntohl(uint32\_t host32bitvalue);

Both return: value in host byte order

**4.5 Byte Manipulation Functions**

There are two groups of functions that operate on multibyte field, without interpreting the data, and without assuming that the data is a null-terminated C string. We need these type of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses, which can obtain bytes of 0, but are not C character strings.

The first group of functions, whose name begins with *b* (for byte), are from 4.3BSD and the second group of functions, whose name begins with *mem* (for memory), are from the ANSI C standard.

The syntax for these functions are :

#include <strings.h>

void bzero(void *\*dest*, size\_t *nbytes*);

void bcopy(const void *\*src*, void *\*dest*, size\_t *nbytes*);

int bcmp(const void *\*ptr1*, const void *\*ptr2*, size\_t *nbytes*);

Returns: 0 if equal, nonzero if unequal#include <string.h>

void \*memset(void *\*dest*, int *c*, size\_t *len*);

void \*memcpy(void *\*dest*, const void *\*src*, size\_t nbytes);

int memcmp(const void \*ptr1, const void \*ptr2, size\_t nbytes);

Returns: 0 if equal, <0 or >0 if unequal

**bzero** sets the specified number of bytes to 0 in the destination.

**bcopy** moves the specified number of bytes from the source to the destination.

**bcmp** compares two arbitrary byte strings. The return vlue is 0 if the two byte strings are identical; otherwise, it is nonzero.

**memse**t sets the specified number of bytes to the value c in the destination.

**memcpy** is similar to bcopy, but the order of the two pointer arguments is swapped. **bcopy** correctly handles overlapping fields, while the behaviour of memcpy is undefined if the source and destination overlap.

**memcmp** compares two arbitrary strings and returns 0 if they are identical. If not identical, the return value is either greater than 0 or less than 0, depending on whether the first unequal byte pointed to by ptr1is greater than or less than the corresponding byte pointed to by ptr2.

**4.6. Address Conversion Functions**

***4.6.1. inet\_aton, inet\_addr, and inet\_ntoa Functions***

These address conversion functions convert Internet addresses between ASCII strings and network byte ordered binary values.

1. inet\_aton inet\_ntoa, and inet\_addr convert an IPv4 address from a dotted-decimal string (e.g., “206.128.34.2”) to its 32-bit network byte ordered binary value.
2. There are newer functions, inet\_pton and inet\_ntop, which handle both IPv4 and IPv6 addresses.

#include<arpa/inet.h>

int inet\_aton(const char \*strptr, struct in\_addr \*addrptr);

Returns: 1 if string was valid, 0 on error

in\_addr\_t inet\_addr(const char \*addrptr);

Returns: 32-bit network byte ordered IPv4 address; INADDR\_NONE if error

char \*inet\_ntoa(struct in\_addr inaddr\_;

Returns: pointer to dotted-decimal string

The first of these, inet\_aton converts the C character string pointer to by strptr into its 32-bit binary network byte ordered value inet\_addr does the same. The problem with this is that all 232 possible values are valid IP addresses (0.0.0.0 through 255.255.255.255), but the function returns the constant INADDR\_NONE(typically 32 one-bits) on an error. This means the string 255.255.255.255 cannot be handled by this function.

The inet\_ntoa function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.

**4.6.2. inet\_pton and inet\_ntop Functions**

These two functions works finely with both IPv4 and IPv6 addresses. Here the letters “p” and “n” stand for “ presentation” and “numeric” respectively.

#include<arpa/inet.h>

int inet\_pton(int family, const char \*strptr, void \*addrptr);

Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error

const char \*inet\_ntop(int family, const void \*addrptr, char \*strptr, size\_t len);

Returns: pointer to result of OK, NULL on error

The family argument for both the functions is either AF\_INET or AF\_INET6.

**4.6.3 Read and Write Functions**

For reading data and writing data from and to the sockets, several functions exists depending upon the type of the socket created. (Stream or Datagram socket).

The socket mechanism requires extensions to the traditional UNIX I/O system calls to provide the associated naming and connection semantics. Rather than overloading the existing interface, the developers used the existing interfaces to the extent that the latter worked without being changed, and designed new interfaces to handle the added semantics. The read and write system calls were used for byte-stream type connections, but six new system calls were added to allow sending and receiving addressed messages such as network datagrams. The system calls for writing messages include send, sendto, and sendmsg. The system calls for reading messages include recv, recvfrom, and recvmsg. In retrospect, the first two in each class are special cases of the others; recvfrom and sendto probably should have been added as library interfaces to recvmsg and sendmsg, respectively.

A read or write on a stream socket might input or output bytes from the socket.

ssize\_t read(int filedes, char \*buff, size\_t nbytes);

ssize\_t write(int filedes, const char \*buff, size\_t nbytes);

Return: number of bytes read or written, -1 on error

filedes is the socket descriptor, buff is the area to store or send data and nbytes is the size of the buffer.

**4.7 Elementary TCP sockets**

The following figure shows a timeline of the typical scenario that takes place between a TCP client and server.

socket()

socket()

connect()

write()

read()

close()

close()

read()

write()

read()

accept()

listen()

bind()

blocks until connection from client

process request

connection establishment

(TCP three-way handshake

data (request)

data (reply)

end-of-file notification

**TCP Client**

well-known port

**TCP Server**

***4.7.1. Socket Function***

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired.

#include<sys/socket.h>

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

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

The following are the possible values for each argument.

Protocol family constants:

|  |  |
| --- | --- |
| **Family** | **Description** |
| AF\_INET  AF\_INET6  AF\_LOCAL  AF\_ROUTE  AF\_KEY | IPv4 protocols  IPv6 protocols  Unix domain protocols  Routing sockets  Key socket |

Type of Socket for socket function

|  |  |
| --- | --- |
| **Type** | **Description** |
| SOCK\_STREAM  SOCK\_DGRAM  SOCK\_SEQPACKET  SOCK\_RAW | Stream socket  Datagram socket  Sequenced packet socket  Raw socket |

Protocol of sockets for **AF\_INET or AF\_INET6**

|  |  |
| --- | --- |
| **Protocol** | **Description** |
| IPPROTO\_TCP  IPPROTO\_UDP  IPPROTO\_SCTP | TCP transport protocol  UDP transport protocol  SCTP transport protocol |

Combinations of family and type for the socket function

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **AF\_INET** | **AF\_INET6** | **AF\_LOCAL** | **AF\_ROUTE** | **AF\_KEY** |
| **SOCK\_STREAM** | TCP/SCTP | TCP/SCTP | Yes |  |  |
| **SOCK\_DGRAM** | UDP | UDP | Yes |  |  |
| **SOCK\_SEQPACKET** | SCTP | SCTP | Yes |  |  |
| **SOCK\_RAW** | IPv4 | IPv6 |  | Yes | Yes |

***4.7.2. Connect Function***

The connect function is used by a TCP client to establish a connection with a TCP server.

#include<sys/socket.h>

int connect(int sockfd, const struct sockaddr \*servaddr, socklen\_t addrlen);

Return: 0 if OK, -1 on error

sockfd is a socket descriptor returned by the socket function. The second and third arguments are pointer to a socket address structure and its size (of the server).

***4.7.3. Bind Function***

The bind function assigns a local protocol address to a socket. The protocol address is a combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

#include<sys/socket.h>

int bind(int sockfd, const struct sockaddr \*myaddr, socklen\_t addrlen);

Return: 0 if OK, -1 on error

***4.7.4. Listen Function***

The listen function is called only by a TCP server and it performs two actions:

1. When a socket is created by the socket function, it is assumed to be an active socket, that is, a client socket that will issue a connect. The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
2. The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

#include<sys/socket.h>

int listen(int sockfd, int backlog);

Returns: 0 if OK, -1 on error

***4.7.5. Accept Function***

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

#include<sys/socket.h>

int accept(int sockfd, struct sockaddr \*cliaddr, socklen\_t \*addrlen);

Return: non-negative descriptor if OK, -1 on error

***4.7.6. Fork Function***

The fork function is the only way in Unix to create a new process.

#include<unistd.h>

pid\_t fork(void);

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

***4.7.7. Close Function***

The normal UNIX close function is also used to close a socket and terminate a TCP connection.

#include<unistd.h>

int close(int sockfd);

Returns: 0 if OK, -1 on error

**4.8 getsockname and getpeername Functions**

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

#include<sys/socket.h>

int getsockname(int sockfd, struct sockaddr \*localaddr, socklen\_t \*addrlen);

int getpeername(int sockfd, struct sockaddr \*peeraddr, socklen\_t \*addrlen);

Both Return: 0 if OK, -1 on error

**4.9 UDP SOCKETS**

There are some fundamantal differences between applications written in TCP versus those that use UDP because of the differences in the two transport layers. UDP sockets are used for connectionless service and it differs syntactically from TCP socket system call with its second argument ‘type’. SOCK\_DGRAM is used as type for UDP sockets. No connection is established from client or server and therefore no connect system call is used like TCP sockets. Listen and accept system calls are not used in UDP server side since it is not listening for any connection request from clients.

The client just sends a datagram to the server using the sendto function, which requires the address of the destination as a parameter. The server just calls the recvfrom function, which waits until data arrives from the client. Recvfrom returns the protocol address of the client along with the datagram, so the server can send a response to the correct client.

***4.9.1 Sendto and recvfrom functions***

ssize\_t sendto(int *socket*, const void \**message*, size\_t *length*, int *flags*, const struct

sockaddr \**dest\_addr*, socklen\_t *dest\_len*);

The *sendto*() function shall send a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message shall be sent to the address specified by *dest\_addr*. If the socket is connection-mode, *dest\_addr* shall be ignored.

The *sendto*() function takes the following arguments:

*socket*

Specifies the socket file descriptor.

*message*

Points to a buffer containing the message to be sent.

*length*

Specifies the size of the message in bytes.

*flags*

Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:

MSG\_EOR

Terminates a record (if supported by the protocol).

MSG\_OOB

Sends out-of-band data on sockets that support out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

*dest\_addr*

Points to a **sockaddr** structure containing the destination address. The length and format of the address depend on the address family of the socket.

*dest\_len*

Specifies the length of the **sockaddr** structure pointed to by the *dest\_addr* argument.

ssize\_t recvfrom(int *socket*, void \**buffer*, size\_t *length*, int *flags*,

struct sockaddr \**address*, socklen\_t \**address\_len*);

The *recvfrom()* function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

The function takes the following arguments:

*socket*

Specifies the socket file descriptor.

*buffer*

Points to the buffer where the message should be stored.

*length*

Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

*flags*

Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG\_PEEK

Peeks at an incoming message. The data is treated as unread and the next *recvfrom()* or similar function will still return this data.

MSG\_OOB

Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG\_WAITALL

Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, if the connection is terminated, if MSG\_PEEK was specified, or if an error is pending for the socket.

*address*

A null pointer, or points to a **sockaddr** structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.

*address\_len*

Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

**4.10. Value-Result arguments**

When a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.

Three functions, bind, connect, and sendto, pass a socket address structure from the process to the kernel. One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in

struct sockaddr\_in serv;

/\* fill in serv{} \*/

connect (sockfd, (SA \*) &serv, sizeof(serv));

Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel.

The below figure 4.1 shows the Socket address structure passed from kernel to the process.

Socket address st

User process

int

value

SAS

length

Protocol address

kernel

#### Figure 4.1 Value arguments

The datatype for the size of a socket address structure is actually socklen\_t and not int, but the POSIX specification recommends that socklen\_t be defined as uint32\_t.

Four functions, accept, recvfrom, getsockname, and getpeername, pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario. Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure.

The reason that the size changes from an integer to be a pointer to an integer is because the size is both a *value* when the function is called (it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in) and a *result* when the function returns (it tells the process how much information the kernel actually stored in the structure). This type of argument is called a *value-result* argument. The below figure 4.2 shows the structure passed from kernel to process

User process

SAS

Socket address st

length

Protocol address

value

result

kernel

**Figure 4.2 Result arguments**

Socket address structures being passed between the process and the kernel. For an implementation such as 4.4BSD, where all the socket functions are system calls within the kernel, this is correct. But in some implementations, notably System V, socket functions are just library functions that execute as part of a normal user process. How these functions interface with the protocol stack in the kernel is an implementation detail that normally does not affect us. Nevertheless, for simplicity, we will continue to talk about these structures as being passed between the process and the kernel by functions such as bind and connect.

Two other functions pass socket address structures: recvmsg and sendmsg . But, we will see that the length field is not a function argument but a structure member.

When using value-result arguments for the length of socket address structures, if the socket address structure is fixed-length, the value returned by the kernel will always be that fixed size: 16 for an IPv4 sockaddr\_in and 28 for an IPv6 sockaddr\_in6, for example. But with a variable-length socket address structure (e.g., a Unix domain sockaddr\_un), the value returned can be less than the maximum size of the structure.

***4.10.1 Setting up a destination address and port number***

An application program creates a variable of type *struct sockaddr\_in*, then assigns the destination address and port number to this variable. In sending or receiving data on the socket connection, this variable is passed as a parameter.

*struct sockaddr\_in server;  
  
/\* set up server name and port number \*/  
server.sin\_family = AF\_INET; /\* use TCP/IP \*/  
server.sin\_port = 800; /\* specify port 800 \*/  
server.sin\_addr.s\_addr = inet\_addr("156.59.20.50");*

***4.10.2 Binding the destination address***

Rather than specify the destination address in each call, the destination address can be bound to the socket.

*/\* set up the server connection side \*/*

*server.sin\_family = AF\_INET; /\* use TCP/IP \*/*

*server.sin\_port = 0; /\* use first available port \*/*

*server.sin\_addr.s\_addr = INADDR\_ANY;*

*if( bind( s, &server, sizeof(server) ) < 0 ) {*

*perror("Error, socket not bound.");*

*exit(3);*

*}*

***4.10.3 Sending data to the socket connection***

There are five possible system calls that an application program can use to send data to a socket. They are *send()*, *sendto()*, *sendmsg()*, *write()* and *writev()*.

The following code fragment sends data to the port.

char buf[32];  
  
strcpy( buf, "Hello" );  
sendto( s, buf, sizeof(buf)+1, 0, &server, sizeof(server));

**4.10.4 *Receiving data from the socket connection***

The following code fragment receives data from the port.

char buf[32];

int s, client\_address\_size;

struct sockaddr\_in client, server;

if( recvfrom( s, buf, sizeof(buf), 0, (struct sockaddr \*) &client,

&client\_address\_size) < 0 ) {

perror("Error getting data from socket connection.");

exit( 4 );

}

**4.11  *Writing Programs***

The C programs are composed of functions. The general structure of any C program is as follows.

Preprocessor Commands

Type definitions

Function prototypes

Variable Declarations

Function Definitions

Example

main()

{

printf (“ Hello, I am with C “);

}

This program, in fact, consists of a single piece of executable code known as a function. All C programs must include a function with the name main,execution of C programs always starts with the execution of the function main, if it is missing the program cannot run and most compilers will not be able to finish the translation process properly without the function called main.

**4.12. Example UDP Client Program and steps for executing it.**

This client program establishes a socket connection and sends the message "Hello" to the server application.

***42.12.1. Client Program***

#include <string.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

main( argc, argv)

int argc;

char \*\*argv;

{

int s;

unsigned short port;

struct sockaddr\_in server;

char buf[32];

/\* argv[1] is internet address of server

argv[2] is port number

Convert the port from ascii to integer and then

from host byte order to network byte order using

htons()

\*/

port = htons( atoi( argv[2] ));

/\* create datagram socket using UDP \*/

printf("Creating datagram socket.\n");

s = socket(AF\_INET, SOCK\_DGRAM, 0);

if( s == -1 )

printf("Socket was not created.\n");

else

printf("Socket created successfully.\n");

/\* set up the server name \*/

server.sin\_family = AF\_INET;

server.sin\_port = port;

server.sin\_addr.s\_addr = inet\_addr( argv[1] );

strcpy( buf, "Hello" );

printf("Sending data to the socket.\n");

sendto( s, buf, (strlen(buf)+1), 0, &server, sizeof(server ) );

printf("Data has been sent to the socket\n");

printf("Closing the socket connection.\n");

close(s);

printf("Socket closed.\n");

}

***4.12.2. Server Program***

This server program establishes a socket connection and receives data from a client application.

/\* server program, run this first \*/

#include <string.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

main()

{

int sockint, s, namelen, client\_address\_size;

struct sockaddr\_in client, server;

char buf[32];

/\* create datagram socket using UDP \*/

printf("Creating datagram socket.\n");

s = socket(AF\_INET, SOCK\_DGRAM, 0);

if( s == -1 )

printf("Socket was not created.\n");

else

printf("Socket created successfully.\n");

/\* set up the server name \*/

server.sin\_family = AF\_INET;

server.sin\_port = 0; /\* use first available port number \*/

server.sin\_addr.s\_addr = INADDR\_ANY;

if( bind(s, &server, sizeof( server )) < 0 ) {

printf("Error binding server.\n");

exit(3);

}

/\* find out what port was assigned \*/

namelen = sizeof( server );

if( getsockname( s, (struct sockaddr \*) &server, &namelen) < 0 ) {

perror("getsockname()\n");

exit(3);

}

printf("The assigned port is %d\n", ntohs( server.sin\_port));

/\* receive message on socket s in buf \*/

client\_address\_size = sizeof( client );

printf("Waiting for a message to arrive.\n");

if( recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr \*) &client, &client\_address\_size) < 0 )

{

printf("recvfrom()\n");

exit(4);

}

/\* print the message \*/

printf("Data has been sent to the socket\n");

printf("The message was\n");

printf("%s\n", buf );

printf("Closing the socket connection.\n");

close(s);

printf("Socket closed.\n");

}

### 4.13 Raw Sockets

Raw mode is basically there to allow you to bypass some of the way that your computer handles TCP/IP. Rather than going through the normal layers of encapsulation/decapsulation that the TCP/IP stack on the kernel does, you just pass the packet to the application that needs it. No TCP/IP processing -- so it's not a processed packet, it's a raw packet. The application that's using the packet is now responsible for stripping off the headers, analyzing the packet, all the stuff that the TCP/IP stack in the kernel normally does for you.

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

The application that takes the data from the raw socket has to do it. It puts a lot more responsibility on that program to create correctly formed headers, etc. It's that they get made by the program taking data from the socket, rather than the TCP/IP stack in the kernel.

### 4.14 Socket Options and Settings

There are various options which can be set for a socket and there are multiple ways to set options that affect a socket.

Of these, **setsockopt()** system call is the one specifically designed for this purpose. Also, we can retrieve the option which are currently set for a socket by means of **getsockopt()** system call.

***int setsockopt(int socket, int level, int option\_name, const void \*option\_value, socklen\_t option\_len);***

The socket argument must refer to an open socket descriptor. The level specifies who in the system is to interpret the option: the general socket code, the TCP/IP code, or the XNS code. This function sets the option specified by the option\_name, at the protocol level specified by the level, to the value pointed to by the option\_value for the socket associated with the file descriptor specified by the socket. The level argument specifies the protocol level at which the option resides. To set options at the socket level, we need to specify the level argument as **SOL\_SOCKET**. To set options at other levels, we need to supply the appropriate protocol number for the protocol controlling the option. The option\_name specifies a single option to set. The option\_name and any specified options are passed uninterpreted to the appropriate protocol module for interpretations. The list of options available at the socket level **(SOL\_SOCKET)** are:

* **SO\_DEBUG**

Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules. This option takes an int value. This is a boolean option.

* **SO\_BROADCAST**

Permits sending of broadcast messages, if this is supported by the protocol. This option takes an int value. This is a boolean option.

* **SO\_REUSEADDR**

Specifies that the rules used in validating addresses supplied to bind() should allow reuse of local addresses, if this is supported by the protocol. This option takes an int value. This is a boolean option.

* **SO\_KEEPALIVE**

Keeps connections active by enabling the periodic transmission of messages, if this is supported by the protocol. This option takes an int value. If the connected socket fails to respond to these messages, the connection is broken and processes writing to that socket are notified with a SIGPIPE signal. This is a boolean option.

* **SO\_LINGER**

Lingers on a close() if data is present. This option controls the action taken when unsent messages queue on a socket and close() is performed. If SO\_LINGER is set, the system blocks the process during close() until it can transmit the data or until the time expires. If SO\_LINGER is not specified, and close() is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option takes a linger structure, as defined in the <sys/socket.h> header, to specify the state of the option and linger interval.

* **SO\_OOBINLINE**

Leaves received out-of-band data (data marked urgent) in line. This option takes an int value. This is a boolean option.

* **SO\_SNDBUF**

Sets send buffer size. This option takes an int value.

* **SO\_RCVBUF**

Sets receive buffer size. This option takes an int value.

* **SO\_DONTROUTE**

Requests that outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an int value. This is a boolean option.

* **SO\_RCVLOWAT**

Sets the minimum number of bytes to process for socket input operations. The default value for SO\_RCVLOWAT is 1. If SO\_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned, e.g. out of band data). This option takes an int value. Note that not all implementations allow this option to be set.

* **SO\_RCVTIMEO**

Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. It accepts a timeval structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it returns with a partial count or errno set to [EAGAIN] or [EWOULDBLOCK] if no data were received. The default for this option is zero, which indicates that a receive operation will not time out. This option takes a timeval structure. Note that not all implementations allow this option to be set.

* **SO\_SNDLOWAT**

Sets the minimum number of bytes to process for socket output operations. Non-blocking output operations will process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option takes an int value. Note that not all implementations allow this option to be set.

* **SO\_SNDTIMEO**

Sets the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it returns with a partial count or with errno set to [EAGAIN] ore [EWOULDBLOCK] if no data were sent. The default for this option is zero, which indicates that a send operation will not time out. This option stores a timeval structure. Note that not all implementations allow this option to be set.

For boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.Options at other protocol levels vary in format and name.

Some of the options available for the **IP\_PROTO\_TCP** socket are:

* **TCP\_MAXSEG**

Returns the maximum segment size in use for the socket.The typical value for a 43.BSD socket using an Ethernet is 1024 bytes.

* **TCP\_NODELAY**

When TCP is being used for a remote login,there will be many small data packets sent from the client's system to the server.Each packet can contain a single character that the user enters which is sent to the server for echoing and processing.It might be desirable to reduce the number of such small packets by combining a number of them into one big packet.But this causes a delay between the typing of a character by the user and its appearance on its monitor.This is certainly not something the user will appreciate. For such services it is desirable that the client's packets be sent as soon as they are ready.The **TCP\_NODELAY** option is used for these clients to defeat the buffering algorithm, and allow the client's TCP to send small packets as soon as possible.

***int getsockopt(int socket, int level, int option\_name, void \*option\_value, socklen\_t \*option\_len);***

This function retrieves the value for the option specified by the option\_name argument for the socket. If the size of the option value is greater than option\_len, the value stored in the object pointed to by the option\_value will be silently truncated. Otherwise, the object pointed to by the option\_len will be modified to indicate the actual length of the value. The level specifies the protocol level at which the option resides. To retrieve options at the socket level, we need to specify the level argument as **SOL\_SOCKET.** To retrieve options at other levels, we need to supply the appropriate protocol number for the protocol controlling the option. The socket in use may require the process to have appropriate privileges to use the **getsockopt()** function. The list of options for option\_name is the same as those available for **setsockopt()** system call.

**4.15 I/O Multiplexing**

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

* When a client is handling multiple descriptors (normally interactive input and a network socket), I/O multiplexing should be used. This is the scenario we described previously.
* It is possible, but rare, for a client to handle multiple sockets at the same time.
* If a TCP server handles both a listening socket and its connected sockets,
* If a server handles both TCP and UDP, I/O multiplexing is normally used.
* If a server handles multiple services and perhaps multiple protocols

**4.15.1 I/O Models**

The five I/O models are:

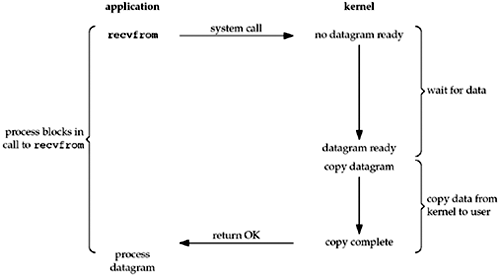
* blocking I/O
* nonblocking I/O
* I/O multiplexing (select and poll)
* signal driven I/O (SIGIO)
* asynchronous I/O (the POSIX aio\_functions)

Two distinct phases for an input operation:

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

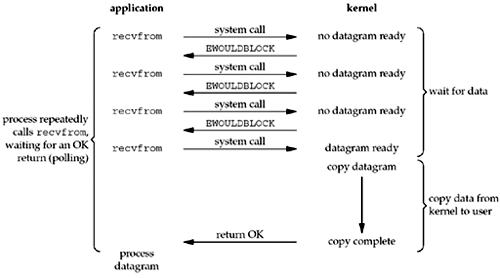
#### 4.15.1.1 Blocking I/O Model:

The most prevalent model for I/O is the blocking I/O model. 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. Process is blocked the entire time from when it calls recvfrom until it returns. When recvfrom returns successfully, our application processes the datagram



#### 4.15.1.2 Nonblocking I/O Model:

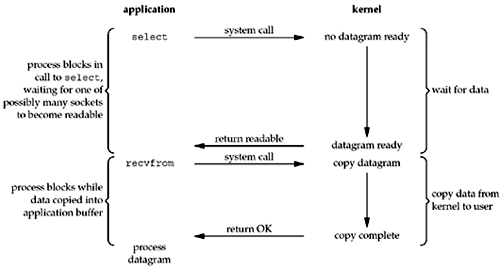
When we set a socket to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead." The first three times that we call recvfrom, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead. The fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. We then process the data.



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.

#### 4.15.1.3 I/O Multiplexing Model

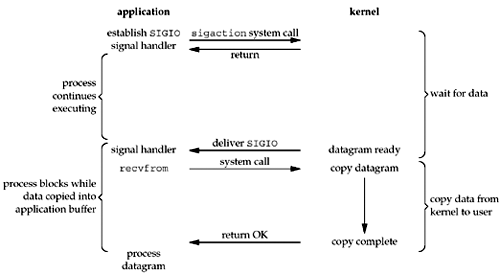
With I/O multiplexing, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call. We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer. The advantage in using select is that we can wait for more than one descriptor to be ready.



#### 4.15.1.4 Signal-Driven I/O Model

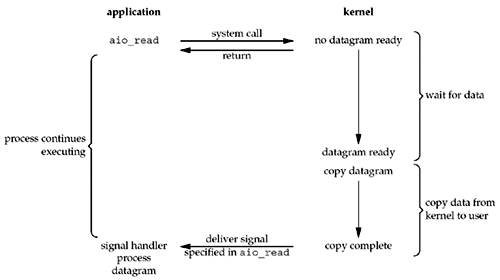
We can also use signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready. We call this signal-driven I/O. We first enable the socket for signal-driven I/O and install a signal handler using the sigaction system call. The return from this system call is immediate and 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, or we can notify the main loop and let it read the datagram.

Regardless of how we handle the signal, 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.



#### 4.15.1.5 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. In general, 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.



**4.15.2 Select Function call**

This 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.

| #include <sys/select.h> |
| --- |
| #include <sys/time.h> |
| int select(int maxfdp1, fd\_set \*readset, fd\_set \*writeset, fd\_set \*exceptset, const struct timeval \*timeout); |
| Returns: positive count of ready descriptors, 0 on timeout, –1 on error |

The final argument, which tells the kernel how long to wait for one of the specified descriptors to become ready. A timeval structure specifies the number of seconds and microseconds.

struct timeval {

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

};

There are three possibilities:

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

The wait in the first two scenarios is normally interrupted if the process catches a signal and returns from the signal handler. The three middle arguments, readset, writeset, and exceptset, specify the descriptors that we want the kernel to test for reading, writing, and exception conditions. There are only two exception conditions currently supported:

1. The arrival of out-of-band data for a socket.
2. The presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode. We do not talk about pseudo-terminals in this book.
3. All the implementation details are irrelevant to the application and are hidden in the fd\_set datatype and the following four macros:

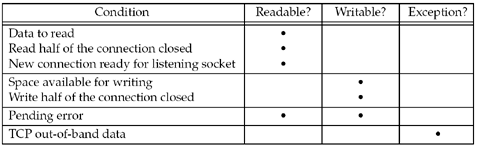
| void FD\_ZERO(fd\_set \*fdset); | /\* clear all bits in fdset \*/ |
| --- | --- |
| void FD\_SET(int fd, fd\_set \*fdset); | /\* turn on the bit for fd in fdset \*/ |
| void FD\_CLR(int fd, fd\_set \*fdset); | /\* turn off the bit for fd in fdset \*/ |
| int FD\_ISSET(int fd, fd\_set \*fdset); | /\* is the bit for fd on in fdset ? \*/ |

The constant FD\_SETSIZE, defined by including <sys/select.h>, is the number of descriptors in the fd\_set datatype.

**4.15.2.1 Under What Conditions Is a Descriptor Ready?**

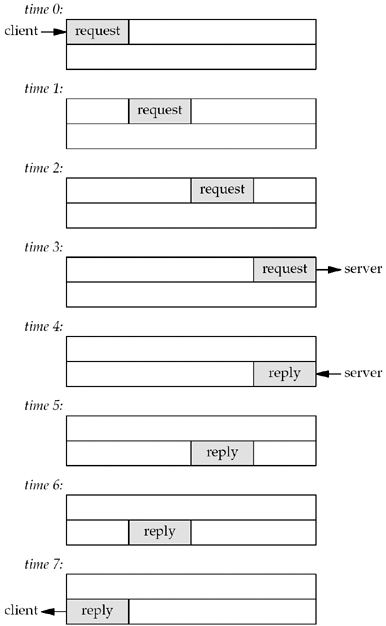
1. A socket is ready for reading if any of the following four conditions is true:
   1. The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer. A read operation on the socket will not block and will return 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.
   2. 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).
   3. The socket is a listening socket and the number of completed connections is nonzero. An accept on the listening socket will normally not block.
   4. 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:
   1. 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 , 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.
   2. The write half of the connection is closed. A write operation on the socket will generate SIGPIPE.
   3. A socket using a non-blocking connect has completed the connection, or the connect has failed.
   4. 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 the SO\_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.

Our definitions of "readable" and "writable" are taken directly from the kernel's soreadable and sowriteable macros on pp. 530–531 of TCPv2. Similarly, our definition of the "exception condition" for a socket is from the soo\_select function on these same pages.

****

### 4.15.3 Batch Input and Buffering

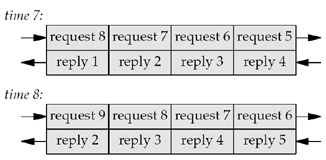
Consider the network between the client and server as a full-duplex pipe, with requests going from the client to the server and replies in the reverse direction.



A request is sent by the client at time 0 and we assume an RTT of 8 units of time. The reply sent at time 4 is received at time 7. We also assume that there is no server processing time and that the size of the request is the same as the reply. We show only the data packets between the client and server, ignoring the TCP acknowledgments that are also going across the network.

Since there is a delay between sending a packet and that packet arriving at the other end of the pipe, and since the pipe is full-duplex, in this example, we are only using one-eighth of the pipe's capacity. This stop-and-wait mode is fine for interactive input, but since our client reads from standard input and writes to standard output, and since it is trivial under the Unix shells to redirect the input and output, we can easily run our client in a batch mode. When we redirect the input and output, however, the resulting output file is always smaller than the input file (and they should be identical for an echo server).

In a batch mode we assume that after sending the first request, we immediately send another, and then another. We also assume that we can keep sending requests as fast as the network can accept them, along with processing replies as fast as the network supplies them



**4.15.4 shutdown Function**

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

1. close decrements the descriptor's reference count and closes the socket only if the count reaches 0. With shutdown, we can initiate TCP's normal connection termination sequence regardless of the reference count.
2. close terminates both directions of data transfer, reading and writing. Since a TCP connection is full-duplex, there are times when we want to tell the other end that we have finished sending, even though that end might have more data to send us.

| #include <sys/socket.h> |
| --- |
| int shutdown(int sockfd, int howto); |
| Returns: 0 if OK, –1 on error |

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

| SHUT\_RD | The read half of the connection is closed— No more data can be received on the socket and any data currently in the socket receive buffer is discarded. The process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP socket is acknowledged and then silently discarded.  By default, everything written to a routing socket loops back as possible input to all routing sockets on the host. Some programs call shutdown with a second argument of SHUT\_RD to prevent the loopback copy. An alternative way to prevent this loopback copy is to clear the SO\_USELOOPBACK socket option. |
| --- | --- |
| SHUT\_WR | The write half of the connection is closed— In the case of TCP, this is called a half-close (Section 18.5 of TCPv1). Any data currently in the socket send buffer will be sent, followed by TCP's normal connection termination sequence. As we mentioned earlier, this closing of the write half is done regardless of whether or not the socket descriptor's reference count is currently greater than 0. The process can no longer issue any of the write functions on the socket. |
| SHUT\_RDWR | The read half and the write half of the connection are both closed— This is equivalent to calling shutdown twice: first with SHUT\_RD and then with SHUT\_WR. |

**4.15.5 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:

1. pselect uses the timespec structure, another POSIX invention, instead of the timeval structure.

struct timespec {

time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

The difference in these two structures is with the second member: The tv\_nsec member of the newer structure specifies nanoseconds, whereas the tv\_usec member of the older structure specifies microseconds.

1. 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.

**4.15.6 poll** Function

| #include <poll.h> |
| --- |
| int poll (struct pollfd \*fdarray, unsigned long nfds, int timeout); |
| Returns: count of ready descriptors, 0 on timeout, –1 on error |

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

struct pollfd {

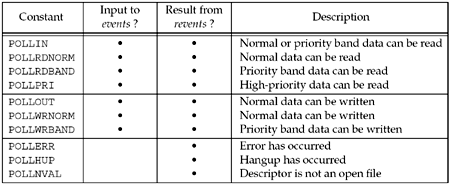
int fd; /\* descriptor to check \*/

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. (Having two variables per descriptor, one a value and one a result, avoids value-result arguments. Recall that 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.

****

There are three classes of data identified by poll: normal, priority band, and high-priority.With regard to TCP and UDP sockets, the following conditions cause poll to return the specified revent. Unfortunately, POSIX leaves many holes (i.e., 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

Domain Name System (DNS)

Map between hostname and IP address

Hostname

simple: netprog

fully qualified (FQDN): netprog.ccis.ksu.edu.sa

Resource Records

􀂄 A (Address) records

􀂅map hostname to IPv4 address

􀂅netprog IN A 10.5.2.150

􀂄 AAAA (quad A) records

􀂅map hostname to IPv6 address

􀂅netprog IN AAAA 3ffe:b80:1f8d:1:a00:20ff

􀂄 CNAME (Canonical Name) records

􀂅 alias for existing A record

􀂅used for common services (www, f

PTR (Pointer) records

􀂅map IP address to hostnames

􀂄 MX records

􀂅specify Main eXchanger host for domain

Address Conversion Functions

􀂄 **gethostbyname**

􀂅returns IPv4 address(es) for a host name

􀂄 **gethostbyaddr**

􀂅returns host name(s) for and IPv4 address

􀂄 **getservbyname**

􀂅returns port number for a service name

􀂅service names defined in /etc/services

**gethostbyname**

**struct hostent \*gethostbyname(**

**const char \*hostname);**

**struct hostent** is defined in netdb.h:

**#include <netdb.h>**

**struct hostent**

**struct hostent {**

**char \*h\_name;**

**char \*\*h\_aliases;**

**int h\_addrtype;**

**int h\_length;**

**char \*\*h\_addr\_list;**

**};**

*official name (canonical)*

*other names*

*AF\_INET or AF\_INET6*

*address length (4 or 16)*

*array of ptrs to addresses*

Getting at the address:

**char \*\*h\_addr\_list;**

**h = gethostbyname("joe.com");**

**sockaddr.sin\_addr.s\_addr =**

**\*(h->h\_addr\_list[0]);**

This won't work!!!!

**h\_addr\_list[0]** is a **char\*** !