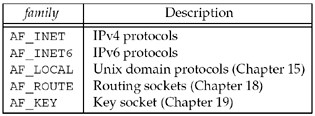
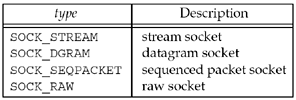
IA 2

1. **Explain the following arguments of the socket function:**
   1. **Family**
   2. **Type**
   3. **Protocol**

*family* specifies the protocol family and is one of the constants. This argument is often referred to as *domain* instead of *family*.

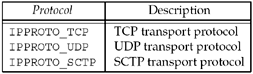


The socket *type* is one of the constants.

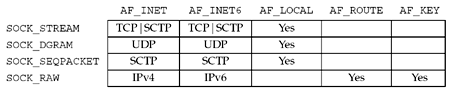


The *protocol* argument to the socket function should be set to the specific protocol type found in

Figure 4.4, or 0 to select the system's default for the given combination of *family* and *type*.



Combinations



1. **Explain the following functions of TCP socket:**
   1. **connect**
   2. **bind**
   3. **listen**
   4. **accept**
   5. **Close**

#include <sys/socket.h>

a.)

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

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

* *sockfd* is a **socket descriptor** returned by the socket function.
* The second and third arguments are a **pointer to a socket address structure** and its **size**,
* The socket address structure must contain the **IP address** and **port number** of the server

**b.)**

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

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

* bind assigns a protocol address to a socket, and what that protocol address means depends on the protocol.
* The second argument is a pointer to a protocol-specific address, and the third argument is the size of this address structure.

**C.)**

* The listen function is called only by a TCP server
* 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.
* The second argument to this function specifies the maximum number of connections
* the kernel should queue for this socket.

#int listen (int *sockfd*, int *backlog*);

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

**d.)**

* 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.

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

* The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client).
* *addrlen* is a value-result argument, before the call, we set the integer value referenced by \**addrlen* to the size of the socket address structure pointed to by *cliaddr*;
* on return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

e.)

The normal Unix close function is also used to close a socket and terminate a TCP Connection

#include <unistd.h>

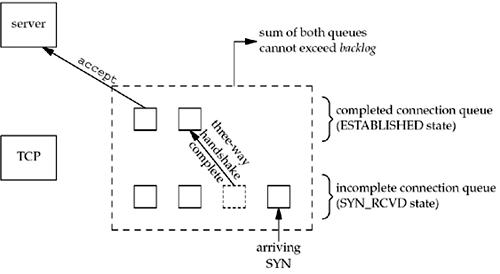
int close (int *sockfd*);

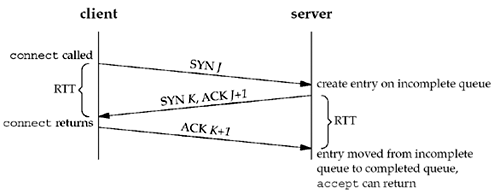
Returns: 0 if OK, -1 on erro

1. **With a neat block diagram explain the queues maintained by TCP for a listening socket. Also show the packets exchanged during the connection establishment with these two queues.**

the kernel maintains two queues:

1. An ***incomplete connection queue***, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN\_RCVD state.
2. A ***completed connection queue***, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.





**14.\_Illustrate the significance of fork and exec functions.**

**fork**

#include <unistd.h>

pid\_t fork(void);

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

* This function is the only way in Unix to create a new process.
* It is called *once* but it returns *twice*.
* It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child).
* It also returns once in the child, with a return value of 0.

There are two typical uses of fork:

1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is typical for **network servers**.
2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies (typically the child process) calls exec to replace itself with the new program. This is typical for programs such as **shells.**

**exec**

#include <unistd.h>

int execl (const char \**pathname*, const char \**arg*0, ... /\* (char \*) 0 \*/ );

int execv (const char \**pathname*, char \*const *argv*[]);

int execle (const char \**pathname*, const char \**arg*0, ...

/\* (char \*) 0, char \*const *envp*[] \*/ );

int execve (const char \**pathname*, char \*const *argv*[], char \*const *envp*[]);

int execlp (const char \**filename*, const char \**arg*0, ... /\* (char \*) 0 \*/ );

int execvp (const char \**filename*, char \*const *argv*[]);

All six return: -1 on error, no return on success

15.\_Outline the typical concurrent server with the help of pseudocode.

pid\_t pid;

int listenfd, connfd;

listenfd = Socket( ... );

/\* fill in sockaddr\_in{} with server's well-known port \*/

Bind(listenfd, ... );

Listen(listenfd, LISTENQ);

for ( ; ; ) {

connfd = Accept (listenfd, ... );

if( (pid = Fork()) == 0) {

Close(listenfd);

doit(connfd); /\* process the request \*/

Close(connfd); /\* done with this client \*/

exit(0); /\* child terminates \*/

}

Close(connfd); /\* parent closes connected socket \*/

}

16.\_Demonstrate the status of client/ server **before and after call to *accept* returns** with a neat block diagram

Status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

**before call to accept returns.**

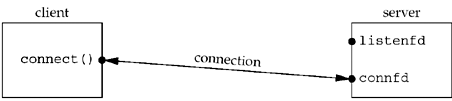


Immediately after accept returns.

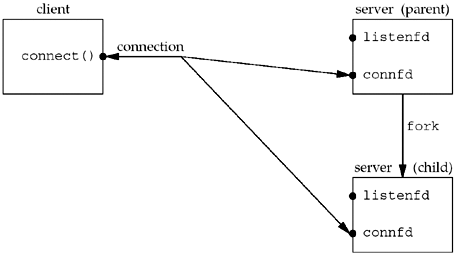
The connection is accepted by the kernel and a new socket, connfd, is created.

This is a connected socket and data can now be read and written across the connection.

**after return from accept.**



**17.\_Demonstrate the status of client/ server after fork returns with a neat block diagram.**



* Notice that both descriptors, listenfd and connfd, are shared between the parent and child.
* The next step is for the parent to close the connected socket and the child to close the listening socket.

19.\_Comment on the significance of *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*);

two functions are required for the following reasons:

1. After connect successfully returns in a TCP client that does not call bind, getsockname returns the local IP address and local port number assigned to the connection by the kernel.
2. After calling bind with a port number of 0 (telling the kernel to choose the local port number), getsockname returns the local port number that was assigned.
3. getsockname can be called to obtain the address family of a socket.

**20.\_Return the address family of a socket**

#include "unp.h"

int sockfd\_to\_family(int sockfd)

{

struct sockaddr\_storage ss;

socklen\_t len;

len = sizeof(ss);

if (getsockname(sockfd, (SA \*) &ss, &len) < 0)

return (-1);

return (ss.ss\_family);

}

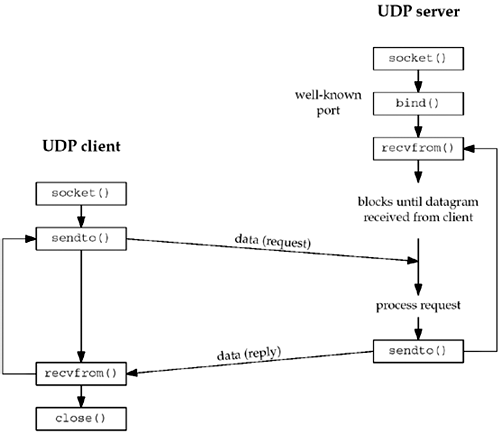
Unit 3:

1. **Illustrate the significance of socket functions for elementary TCP client/server with a neat block diagram.**

UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable

byte stream provided by TCP.

* Figure 8.1 shows the function calls for a typical UDP client/server.
* The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the sendto function , which requires the address of the destination (the server) as a parameter.
* The server does not accept a connection from a client. Instead, the server just calls the recvfrom function, which waits until data arrives from some client.
* recvfrom returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

 **Socket functions for UDP client/server.**

1. **Explain the following functions of UDP socket:**
   1. **recvfrom**
   2. **sendto**

These two functions are similar to the standard read and write functions, but three additional arguments are required.

#include <sys/socket.h>

ssize\_t recvfrom(int *sockfd*, void \**buff*, size\_t *nbytes*, int *flags*, struct sockaddr \*

*from*, socklen\_t \**addrlen*);

ssize\_t sendto(int *sockfd*, const void \**buff*, size\_t *nbytes*, int *flags*, const struct

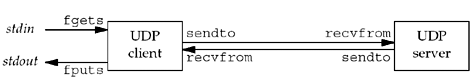
sockaddr \**to*, socklen\_t *addrlen*);

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

* The first three arguments, *sockfd, buff*, and *nbytes* are descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.
* The *to* argument for sendto is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent.
* The size of this socket address structure is specified by *addrlen*.
* The recvfrom function fills in the socket address structure pointed to by *from* with the protocol address of who sent the datagram.
* The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by *addrlen*.
* Note that the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).

**3.\_List and explain with a neat block diagram the steps associated with simple UDP echo client and server.**

Simple echo client/server using UDP.



**UDP Echo Server**

**Main function**

#include "unp.h"

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

{

int sockfd;

struct sockaddr\_in servaddr, cliaddr;

sockfd = Socket(AF\_INET, SOCK\_DGRAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

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

dg\_echo(sockfd, (SA \*) &cliaddr, sizeof(cliaddr));

}

***Create UDP socket, bind server's well-known port***

* We create a UDP socket by specifying the second argument to socket as SOCK\_DGRAM (a datagram socket in the IPv4 protocol).
* As with the TCP server example, the IPv4 address for the bind is specified as INADDR\_ANY and the server's well-known port is the constant SERV\_PORT from the unp.h header.
* The function dg\_echo is called to perform server processing.

**dg\_echo function**

#include "unp.h"

void dg\_echo(int sockfd, SA \*pcliaddr, socklen\_t clilen)

{

int n;

socklen\_t len;

char mesg[MAXLINE];

for ( ; ; ) {

len = clilen;

n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

Sendto(sockfd, mesg, n, 0, pcliaddr, len);

}

}

**Read datagram, echo back to sender**

* This function is a simple loop that reads the next datagram arriving at the server's port using recvfrom and sends it back using sendto.
* This function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.
* This function provides an *iterative server*, not a concurrent server as we had with TCP.
* There is no call to fork, so a single server process handles any and all clients.
* UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer this is how the queing takes place.

**UDP Echo Client**

Main function

#include "unp.h"

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

{

int sockfd;

struct sockaddr\_in servaddr;

if(argc != 2)

err\_quit("usage: udpcli <IPaddress>");

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(SERV\_PORT);

Inet\_pton(AF\_INET, argv[1], &servaddr.sin\_addr);

sockfd = Socket(AF\_INET, SOCK\_DGRAM, 0);

dg\_cli(stdin, sockfd, (SA \*) &servaddr, sizeof(servaddr));

exit(0);

}

**Fill in socket address structure with server's address**

* An IPv4 socket address structure is filled in with the IP address and port number of the server.
* This structure will be passed to dg\_cli, specifying where to send datagrams.
* A UDP socket is created and the function dg\_cli is called.

dg\_cli function

#include "unp.h"

void dg\_cli(FILE \*fp, int sockfd, const SA \*pservaddr, socklen\_t servlen)

{

int n;

char sendline[MAXLINE], recvline[MAXLINE + 1];

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

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

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

recvline[n] = 0; /\* null terminate \*/

fputs(recvline, stdout);

}

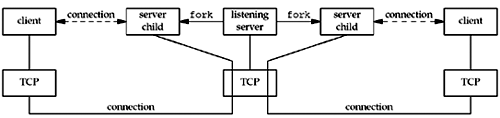
}

There are four steps in the client processing loop:

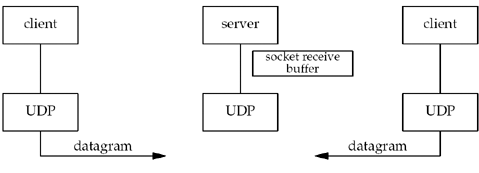
* read a line from standard input using fgets,
* send the line to the server using sendto,
* read back the server's echo using recvfrom,
* and print the echoed line to standard output using fputs.

**4.\_Outline the summary of TCP client/server with two clients.**

**5.\_Outline the summary of UDP client/server with two clients.**



There are two connected sockets and each of the two connected sockets on the server host has its own socket receive buffer.



There is only one server process and it has a single socket on which it receives all arriving datagrams and sends all responses. That socket has a receive buffer into which all arriving datagrams are placed.

**10. Develop the ‘C’ program for dg\_cli function that verifies returned socket address**

#include "unp.h"

void dg\_cli(FILE \*fp, int sockfd, const SA \*pservaddr, socklen\_t servlen)

{

int n;

char sendline[MAXLINE], recvline[MAXLINE + 1];

socklen\_t len;

struct sockaddr \*preply\_addr;

preply\_addr = Malloc(servlen);

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

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

len = servlen;

n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply\_addr, &len);

if (len != servlen || memcmp(pservaddr, preply\_addr, len) != 0) {

printf("reply from %s (ignored)\n", Sock\_ntop(preply\_addr, len));

continue;

}

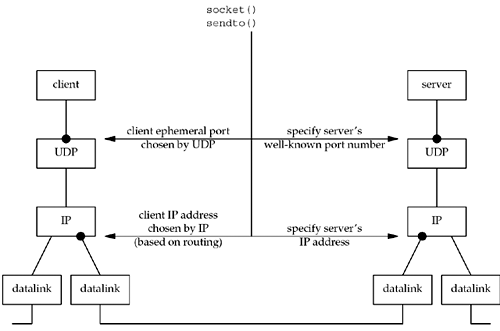
recvline[n] = 0; /\* null terminate \*/

Fputs(recvline, stdout);

}

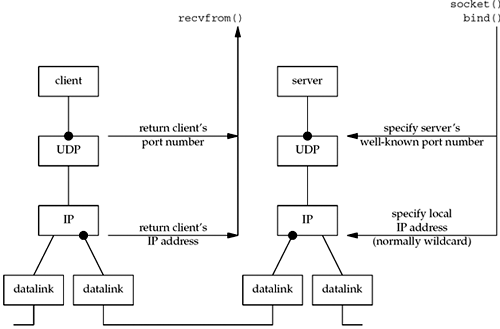
}

1. **\_Outline the summary of UDP client/server from client’s perspective with a neat block diagram**



* The client must specify the server's IP address and port number for the call to sendto.
* Normally, the client's IP address and port are chosen automatically by the kernel, although we mentioned that the client can call bind if it so chooses.
* Client's IP address can change for every UDP datagram that the client sends, assuming the client does not bind a specific IP address to the socket.
* If the client host is multihomed, the client could alternate between two destinations, one going out the datalink on the left, and the other going out the datalink on the right.
* In this worst-case scenario, the client's IP address, as chosen by the kernel based on the outgoing datalink, would change for every datagram.

**12.\_Outline the summary of UDP client/server from server’s perspective with a neat block diagram.**



There are at least four pieces of information that a server might want to know from an arriving IP datagram:

1. the source IP address,
2. destination IP address,
3. source port number,
4. and destination port number.

**13.\_Develop the ‘C’ program to demonstrate the UDP dg\_cli function that calls connect.**

#include "unp.h"

void dg\_cli(FILE \*fp, int sockfd, const SA \*pservaddr, socklen\_t servlen)

{

int n;

char sendline[MAXLINE], recvline[MAXLINE + 1];

Connect(sockfd, (SA \*) pservaddr, servlen);

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

Write(sockfd, sendline, strlen(sendline));

n = Read(sockfd, recvline, MAXLINE);

recvline[n] = 0; /\* null terminate \*/

Fputs(recvline, stdout);

}

}

**14.\_Develop the ‘C’ program to demonstrate the UDP dg\_cli function that writes a fixed number of datagrams to the server.**

#include "unp.h"

#define NDG 2000 /\* datagrams to send \*/

#define DGLEN 1400 /\* length of each datagram \*/

void dg\_cli(FILE \*fp, int sockfd, const SA \*pservaddr, socklen\_t servlen)

{

int i;

char sendline[DGLEN];

for (i = 0; i < NDG; i++) {

Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);

}

}

**15.\_Develop the ‘C’ program to demonstrate the UDP dg\_echo function that counts received datagrams.**

#include "unp.h"

static void recvfrom\_int(int);

static int count;

void dg\_echo(int sockfd, SA \*pcliaddr, socklen\_t clilen)

{

socklen\_t len;

char mesg[MAXLINE];

Signal(SIGINT, recvfrom\_int);

for ( ; ; ) {

len = clilen;

Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

count++;

}

}

static void recvfrom\_int(int signo)

{

printf("\nreceived %d datagrams\n", count);

exit(0);

}

**16.\_Develop the ‘C’ program to demonstrate the UDP dg\_echo function that increases the size of the socket receive queue.**

#include "unp.h"

static void recvfrom\_int(int);

static int count;

void dg\_echo(int sockfd, SA \*pcliaddr, socklen\_t clilen)

{

int n;

socklen\_t len;

char mesg[MAXLINE];

Signal(SIGINT, recvfrom\_int);

n = 220 \* 1024;

Setsockopt(sockfd, SOL\_SOCKET, SO\_RCVBUF, &n, sizeof(n));

for ( ; ; ) {

len = clilen;

Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

count++;

}

}

static void recvfrom\_int(int signo)

{

printf("\nreceived %d datagrams\n", count);

exit(0);

}