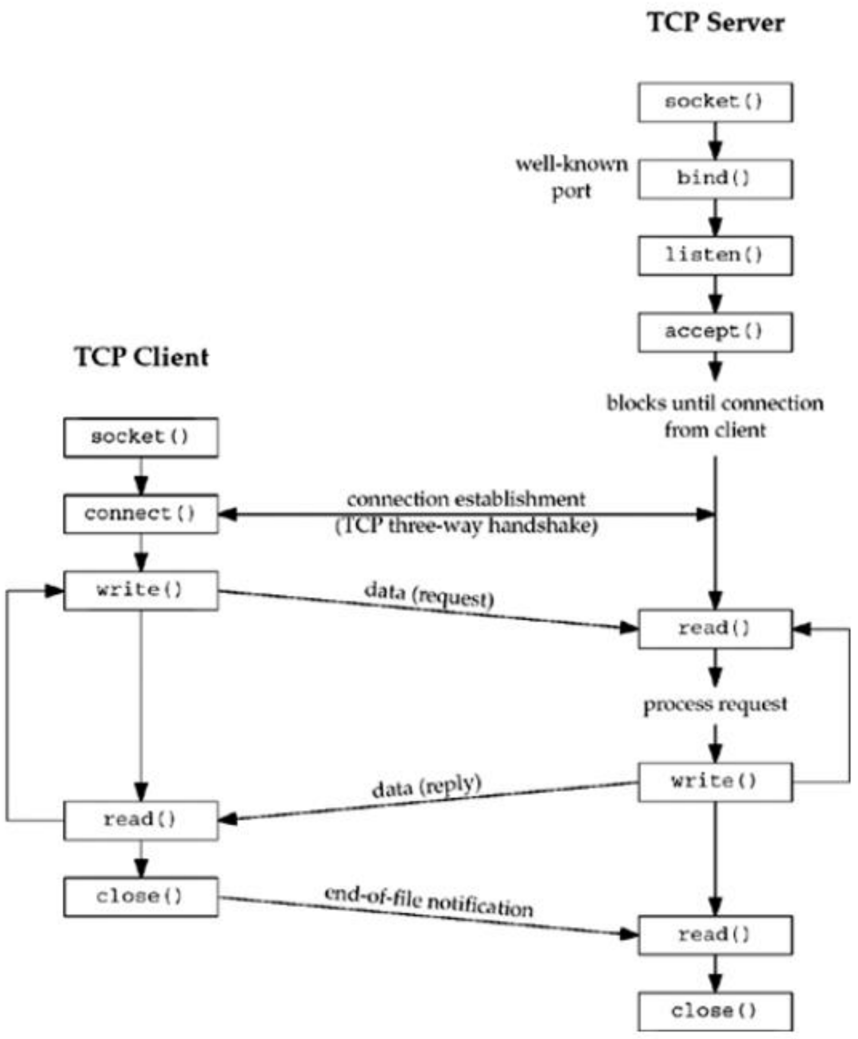
UNIT-3

1. Illustrate the significance of socket functions for UDP and TCP client server with a neat block diagram.

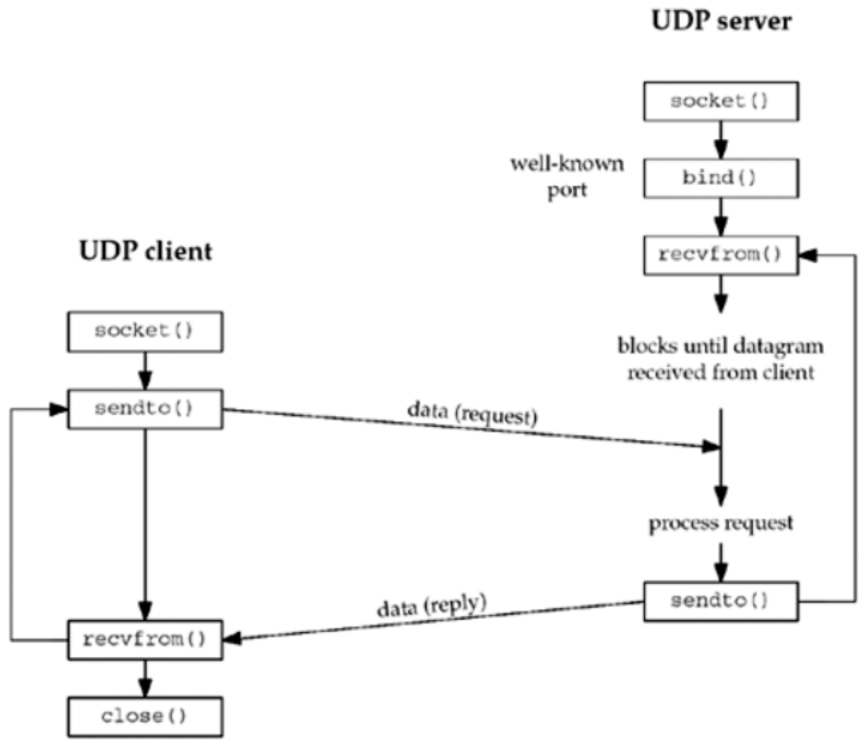
Ans: TCP:

- First, the server is started, then sometime later, a client is started that connects to the server.
- We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client.
- This continues until the client closes its end of the connection, which sends an end-of-file notification to the server.
- The server then closes its end of the connection and either terminates or waits for a new client connection.



UDP:

- The Figure 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 (described in the next section), which requires the address of the destination (the server) as a parameter.
- Similarly, 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.
- Figure shows a timeline of the typical scenario that takes place for a UDP client/server exchange.



2. Develop the “C” program for dg_cli function that verifies returned socket address

Ans:

```
#include “unp.h”

void dg_cli(FILE *fp, int sock, const SA *pseraddr, socklen_t servlen){

        int n;

        char sendline[MAXLINE], recvline[MAXLINE];

        socklen_t len;

        struct sockaddr *preply_addr;

        preply_addr = Malloc(servlen);

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

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

                Len = servlen;

                n = Recvfrom(sockfdm, recvline, MAXLINE, 0, preply_addr, &len);

                if(len != servlen || mememp(pservaddr, preply_addr, len) != 0){

                        printf(“reply from %s (ignore)\n”, Sock_ntop(preply_addr, len));

                        continue;

                }

                recvline[n] = 0;

                fputs(recvline, stdout);

        }

}
```

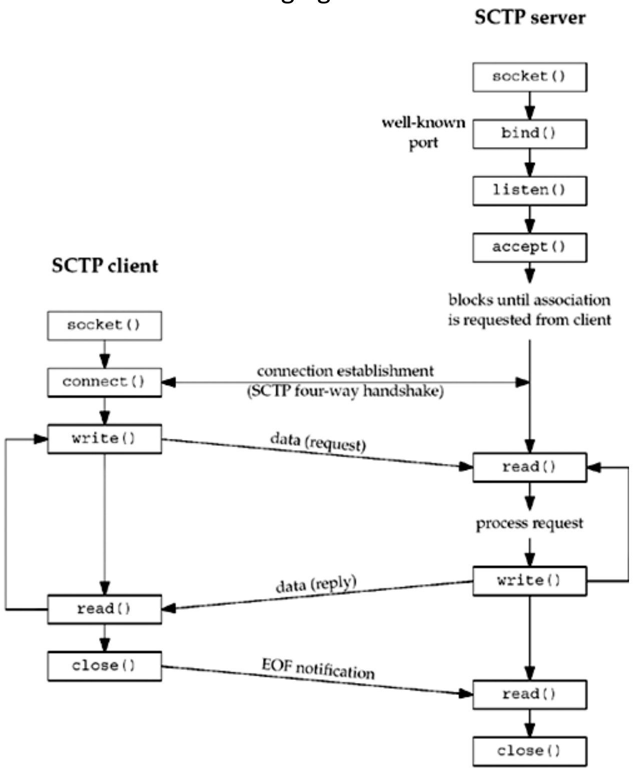
3. Explain briefly about the different interface models that are used in SCTP Protocol

Ans:

There are two interface models in SCTP protocol: the one-to-one style and the one-to-many style.

One-to-One Style:

- The one-to-one style in SCTP is mostly compatible with existing TCP applications, making it easier to migrate from TCP to SCTP.
- It provides a similar model to TCP, allowing for a single SCTP association per socket.
- This style is suitable for applications that require a one-to-one connection between endpoints.
- The one-to-one style is shown in the following figure

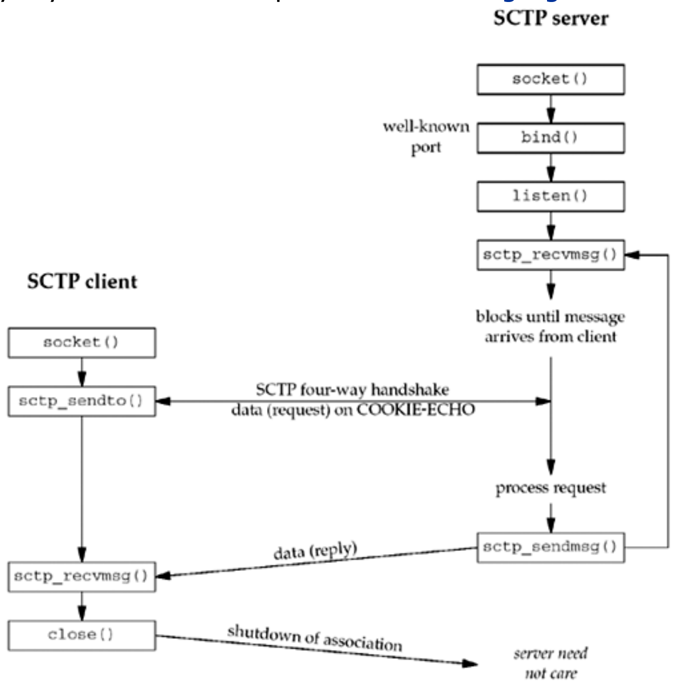


- When the server is started, it opens a socket, binds to an address, and waits for aclient connection with the accept system call.
- Sometime later, the client is started, it opens a socket, and initiates an association with the server.
- We assume the client sends a request to the server, the server processes the request, and the server sends back a reply to the client.
- This cycle continues until the client initiates a shutdown of the association.
- This action closes the association, whereupon the server either exits or waits for a new association.
- As can be seen by comparison to a typical TCP exchange, an SCTP one-to-one socket exchange proceeds in a fashion similar to TCP.

One-to-Many Style:

- The one-to-many style in SCTP allows for multiple SCTP associations to be active on a single socket simultaneously.
- This style is similar to a UDP socket, where multiple concurrent transport-layer associations can be established.
- It provides access to all of SCTP's features and is recommended for new applications developed for SCTP.

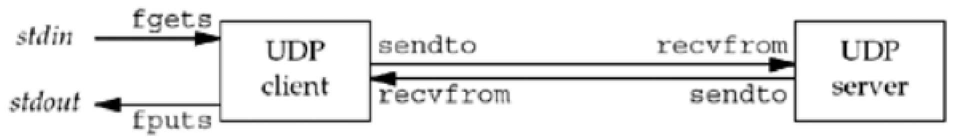
A typical one-to-many style timeline is depicted in following figure.



- First, the server is started, creates a socket, binds to an address, calls listen to enable client associations, and calls sctp_recvmsg, which blocks waiting for the first message to arrive.
- A client opens a socket and calls sctp_sendto, which implicitly sets up the association and piggybacks the data request to the server on the third packet of the four-way handshake.
- The server receives the request, and processes and sends back a reply. The client receives the reply and closes the socket, thus closing the association.
- The server loops back to receive the next message.

4. Develop the “C” program to demonstrate the UDP echo client: main function and dg_cli function

Ans:



UDP Echo Server: '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);

}
```

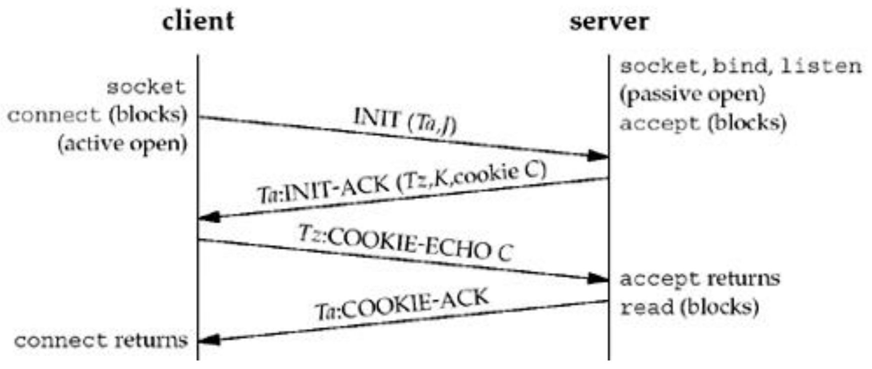
UDP Echo Client: 'dg_cli' Function,

```
#include "unp.h"

void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen){

    int n;

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

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

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

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

        Fputs(recvline, stdout); } }
```

5. Construct a suitable sequence diagram to indicate the functionality of 4-way handshake for the establishment of an association in SCTP protocol.

Ans:

- The server must be prepared to accept an incoming association.
- This preparation is normally done by calling socket, bind, and listen.
- The client issues an active open by calling connect or by sending a message.
- This causes the client SCTP to send an INIT message to tell the server the client's list of IP addresses, the number of outbound streams the client is requesting, and the number of inbound streams the client can support.
- The server acknowledges the client's INIT message with an INIT-ACK message, which contains the server's list of IP addresses, number of outbound streams the server is requesting, number of inbound streams the server can support, and a state cookie.
- The client echos the server's state cookie with a COOKIE-ECHO message.
- The minimum number of packets required for this exchange is four; hence, this process is called SCTP's four-way handshake.



6. Explain the following functions of UDP socket: recvfrom b. sendto

Ans: 'recvfrom' and 'sendto' Functions:

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);
#include<sys/socket.h>
ssize_t **sendto** (int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr *to, socklen_t addrlen);

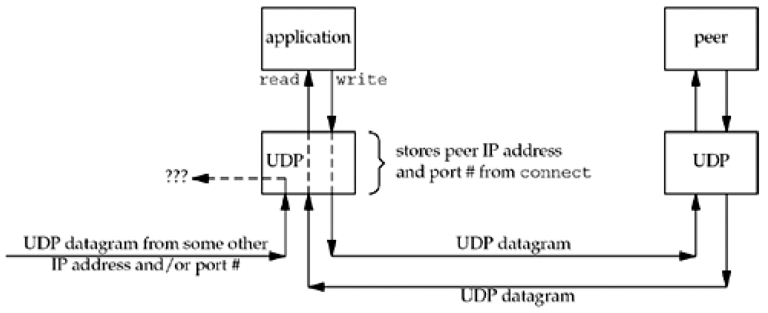
- The first three arguments, **sockfd, buff, and nbytes**, are identical to the first three arguments for read and write: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.
- The recvfrom function fills in the socket address structure pointed to by from with 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.
- The to argument for sendto is a socket address structure containing the protocol address of where the data is to be sent.
- The size of this socket address structure is specified by addrlen.

7. Differentiate between Unconnected and Connected UDP sockets. Identify the resulting changes with a connected UDP socket compared to default connected UDP socket?

Ans:

Unconnected UDP socket	Connected UDP socket
Is the default when we create a UDP socket.	Is the result of calling 'connect' on a UDP socket.
For sending data, you explicitly include the destination address and port in the sendto() function. For receiving data, you use the recvfrom() function	Once a UDP socket is "connected," you can use send() and recv() functions without specifying the destination address and port each time.
Unconnected UDP sockets are suitable when you need to send data to different endpoints or when the communication pattern doesn't require maintaining a connection state. This is common in applications where low latency is crucial, and the overhead of connection management in TCP is not acceptable.	Connected UDP sockets are useful in scenarios where you are repeatedly sending or receiving data to/from the same remote endpoint. It can provide a slight performance benefit because you don't need to specify the destination for every packet.

- With a connected UDP socket, three things change, compared to the default unconnected UDP socket:
  - We can no longer specify the destination IP address and port for an output operation. That is, we do not use sendto, but write or send instead. Anything written to a connected UDP socket is automatically sent to the protocol address.
  - We do not need to use recvfrom to learn the sender of a datagram, but read, recv, or recvmsg instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in connect.
  - Asynchronous errors are returned to the process for connected UDP sockets. The corollary, as we previously described, is that unconnected UDP sockets do not receive asynchronous errors.


application
read
write
peer
UDP
???
stores peer IP address and port # from connect
UDP
UDP datagram from some other IP address and/or port #
UDP datagram
UDP datagram

Connected UDP socket.