NP U3

1. **UDP Echo Client/Server- functions - dg_cli isnt that imp**
    a. main
    b. dg_echo
    c. dg_cli Functions

Answer - Main

```
 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));
 }
```

Create UDP socket, bind server's well-known port
We create a UDP socket by specifying the second argument to socket as SOCK_DGRAM . 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.

-Dg_Cli

```
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);
}
```

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
print the echoed line to standard output using fputs .

c Dg_Echo

```
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);
```

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 .

Despite the simplicity of this function, there are numerous details to consider.

this function never terminates.
this function provides an iterative server, not a concurrent server
There is no call to fork , so a single server process handles any and all clients.

2, Explain the lack of flow control with UDP.

There is no flow control, and hence no window mechanism. The receiver may overflow with incoming messages. The lack of flow control means that the process using UDP should provide for this service if needed.

To examine the effect of UDP not having any flow control, we modify our dg_cli function to send a fixed number of datagrams.

Next, we modify the server (dg_echo) to receive datagrams and count the number received.
When we terminate the server with our terminal interrupt key, it prints the number of received datagrams and terminates.

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);
}
}
```

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);
}
```

3. **Steps involved in building an echo Client-Server application using UDP.**

Client
In the UDP Echo Client,

-a socket a created
-Then we bind the socket
-After the binding is successful, we send messages input from the user
-Wait until a response from the server is received
-Process the reply and display the data received from the server using sendto() and recvfrom() functions.

Server
-In the UDP Echo Server, we create a socket and bind to an advertised port number
-Then an infinite loop is started to process the client requests for connections
-The process receives data from the client using the recvfrom() function and echoes the same data using the sendto() function.
-This type of server is capable of handling multiple clients automatically as UDP is a datagram-based protocol. Hence no exclusive connection is required to a client in this case.

4.Identify the resulting changes with a connected UDP socket compared to the

default connected UDP socket?

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 specified by connect.

*Similar to TCP, we can call sendto for a connected UDP socket, but we cannot specify a destination address. The fifth argument to sendto() must be a null pointer, and the sixth argument should be 0. The POSIX specification states that when the fifth argument is a null pointer, the sixth argument is ignored.

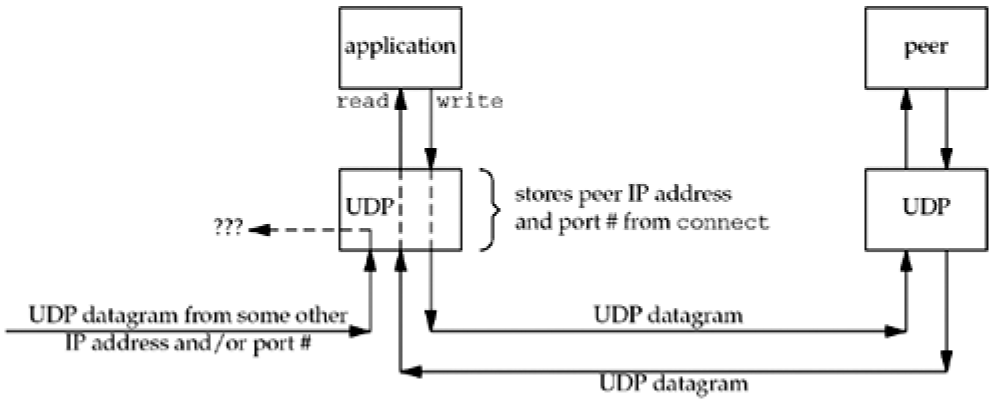
*We do not need to use recvfrom() to learn the sender of a datagram, but read, recv, or recvmsg instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in connect.

*This limits a connected UDP socket to
exchanging datagrams with one and only one peer. Technically, a connected UDP socket exchanges datagrams with only one IP address, because it is possible to connect to a multicast or broadcast address.

*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.

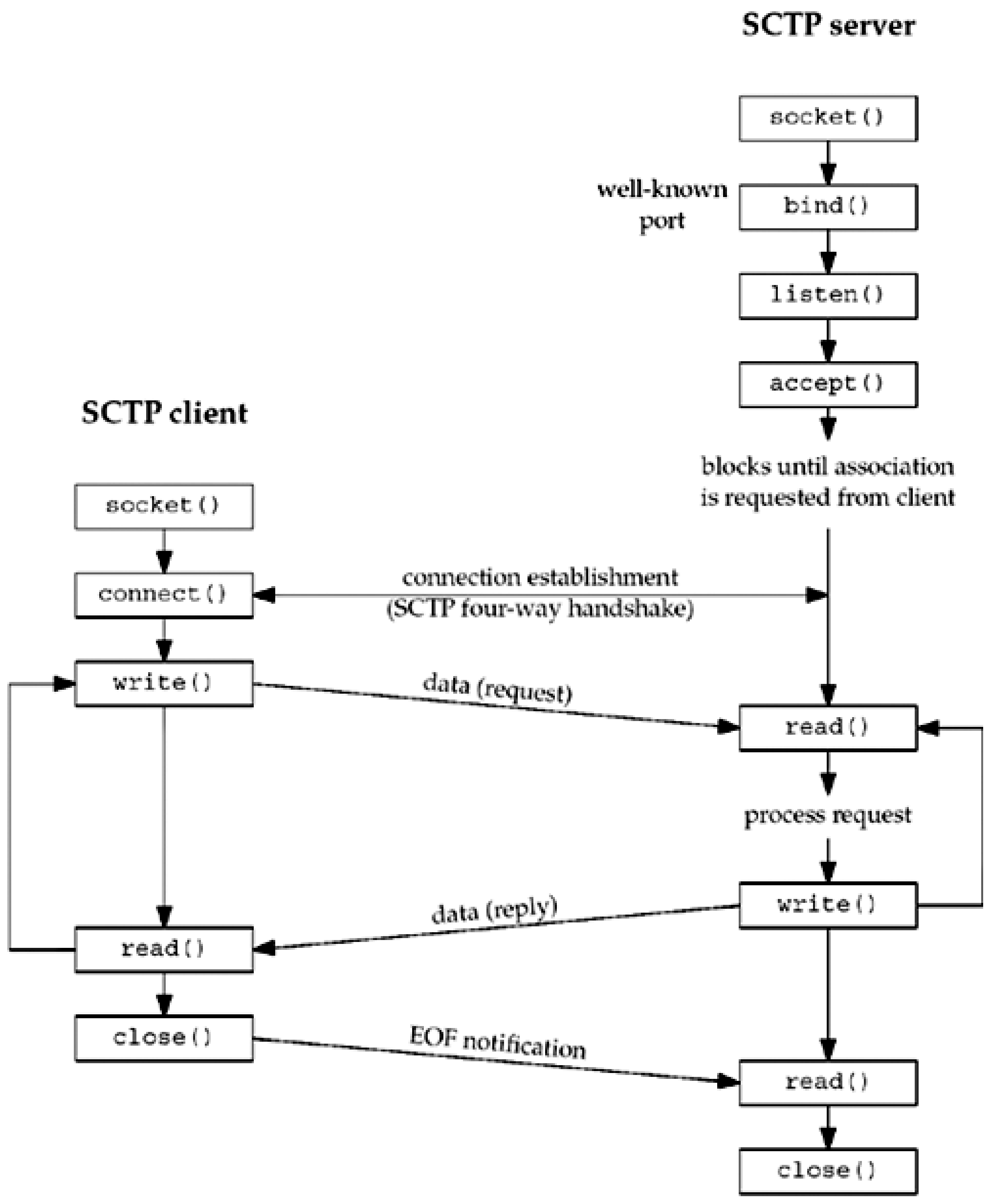
5. Connected vs unconnected sockets

Please write points from answer 4 along with this diagram


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

6. **Interface Models**

The One-to-One Style
The one-to-one style was developed to ease the porting of existing TCP applications to SCTP.



-Any socket options must be converted to the SCTP equivalent.

-SCTP preserves message boundaries; thus, application-layer message boundaries are not required. For example, an application protocol based on TCP might do a write() system call to write a two-byte message length field, x, followed by a write() system call that writes x bytes of data.

-However, if this is one with SCTP, the receiving SCTP will receive two separate messages.

-Some TCP applications use a half-close to signal the end of input to the other side. To port such applications to SCTP, the application-layer protocol will need to be rewritten so that the application signals the end of input in the application data stream.

-The send function can be used in a normal fashion.

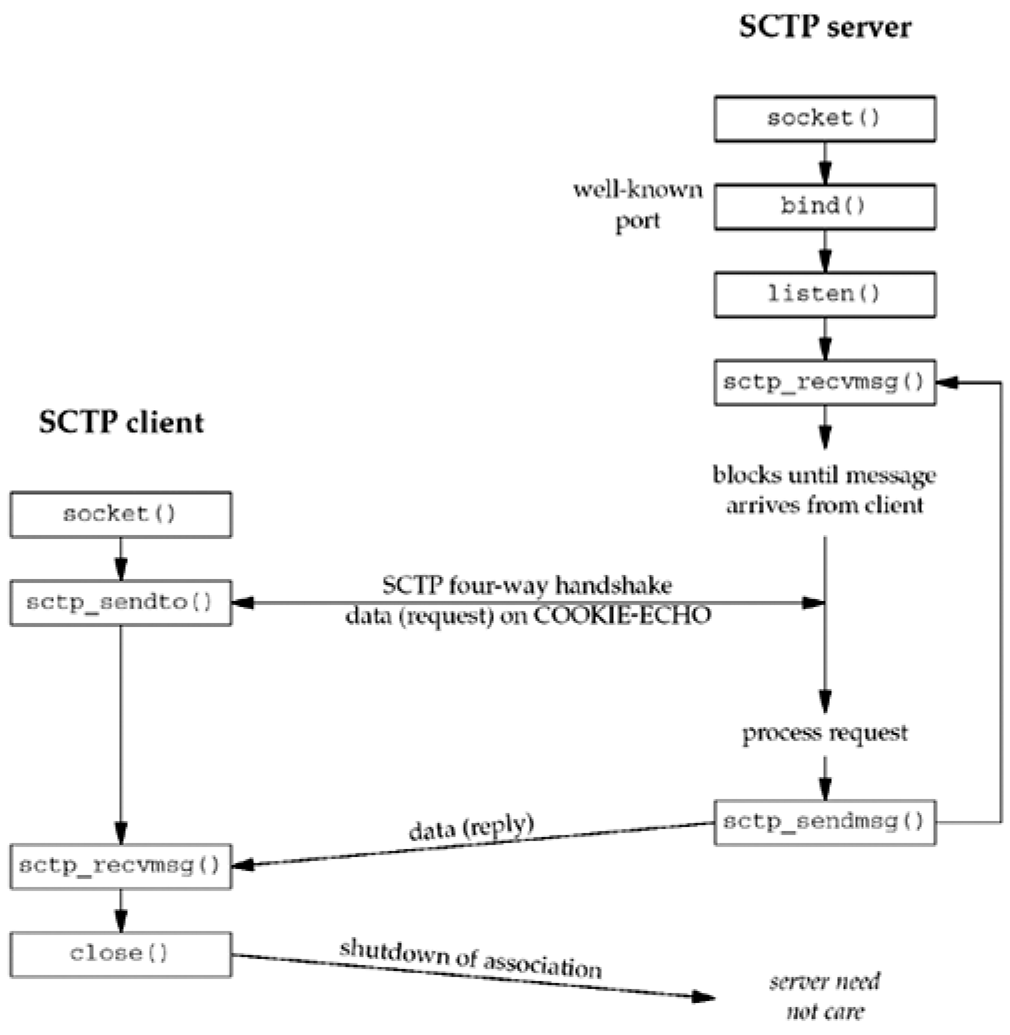
One to many interface model

-The one-to-many style provides an application writer the ability to write a server without managing a large number of socket descriptors.

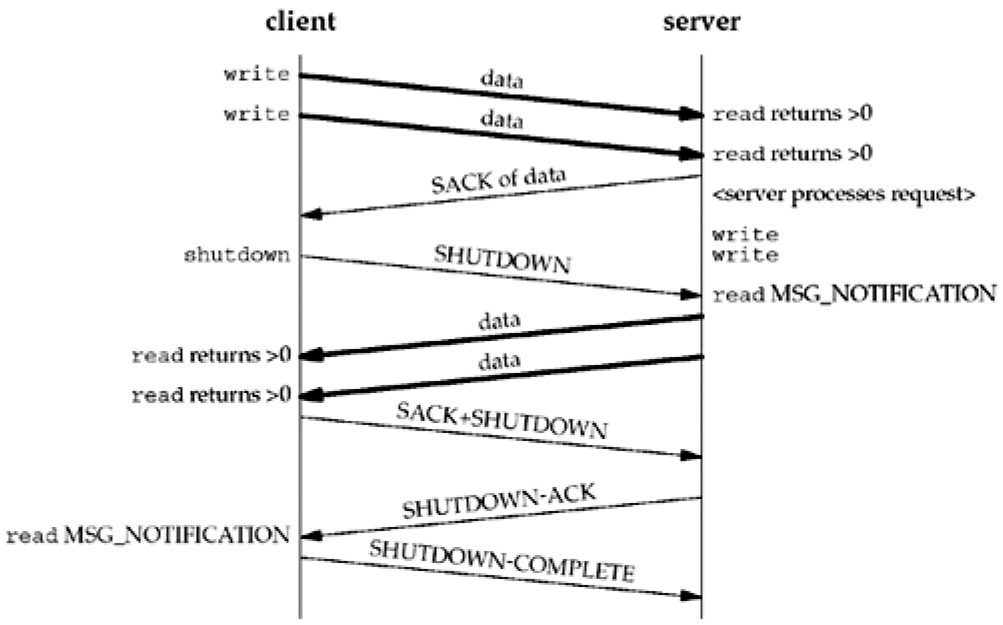
-When the client closes the association, the server-side will automatically close as well, thus removing any state for the association inside the kernel.

-Using the one-to-many style is the only method that can be used to cause data to be piggybacked on the third or fourth packet of the four-way handshake.

-Any sendto, sendmsg, or sctp_sendmsg to an address for which an association does not yet exist will cause an active open to be attempted, thus creating a new association with that address.
-The user must use the sendto, sendmsg, or sctp_sendmsg functions, and may not use the send or write function.



6. **Shutdown Function in SCTP**



-The shutdown function can be used with an SCTP endpoint using the one-to-one-style interface. -

-Because SCTP's design does not provide a half-closed state, an SCTP endpoint reacts to a shutdown call differently than a TCP endpoint.

When an SCTP endpoint initiates a shutdown sequence, both endpoints must complete transmission of any data currently in the queue and close the association. The endpoint that initiated the active open may wish to invoke shutdown instead of close so that the endpoint can be used to connect to a new peer.

7. **Sequence diagram to indicate the functioning of a 4-way handshake for the establishment of an association in SCTP protocol.**

-The server must be prepared to accept an incoming association.

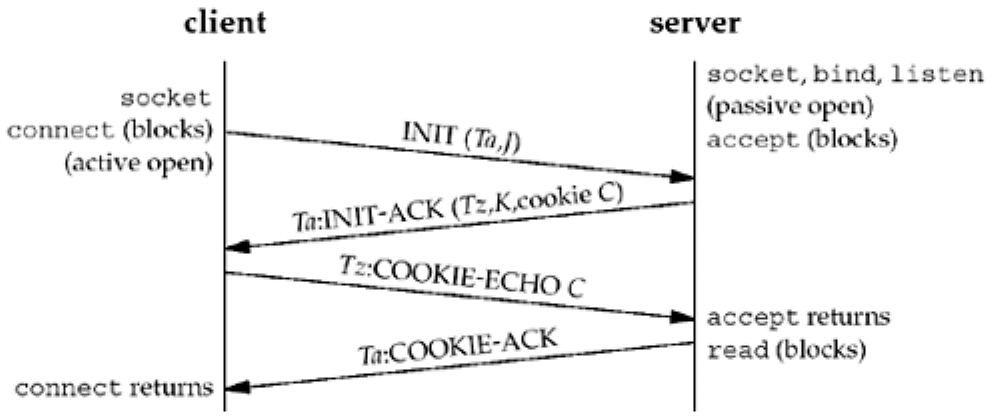
-This preparation is normally done by calling socket, bind, and listen.

-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.



8. Sendto and recfrom 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);

---

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr *to, socklen_t 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*.

-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 final two arguments to recvfrom are similar to the final two arguments to accept: The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP).

The final two arguments to sendto are similar to the final two arguments to connect: We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).

9. **Explain the server not running thingy - not important , im skipping**

Ans - starting the client without starting the server. If we do so and type in a single line to the client, nothing happens. The client blocks forever in its call to recvfrom, waiting for a server reply that will never appear.

```
int
main(int argc, char **argv)
{
    int     sockfd;
    socklen_t len;
    struct sockaddr_in cliaddr, servaddr;

    if (argc != 2)
        err_quit("usage: udpcli <IPaddress>");

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

    len = sizeof(cliaddr);
    Getsockname(sockfd, (SA *) &cliaddr, &len);
    printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

    exit(0);
}
```

10. Notification in SCTP - again i am not doing this
Ans - SCTP makes a variety of notifications available to the application programmer. The SCTP user can track the state of its association(s) via these notifications. Notifications communicate transport-level events, including network status change, association startups, remote operational errors, and undeliverable messages.

Eight events can be subscribed to using the SCTP_EVENTS socket option.

-Seven of these events generate additional data—termed a notification—that a user will receive via the normal socket descriptor.

Notifications have the following form:

```
struct sctp_tlv {
  u_int16_t sn_type;
  u_int16_t sn_flags;
  u_int32_t sn_length;
};

/* notification event */
union sctp_notification {
  struct sctp_tlv sn_header;
  struct sctp_assoc_change sn_assoc_change;
  struct sctp_paddr_change sn_paddr_change;
  struct sctp_remote_error sn_remote_error;
  struct sctp_send_failed sn_send_failed;
  struct sctp_shutdown_event sn_shutdown_event;
  struct sctp_adaption_event sn_adaption_event;
  struct sctp_pdapi_event sn_pdapi_event;
};



11. SCTP One-to-Many-Style Streaming Echo Server: main Function

i-Set stream increment option
By default, our server responds using the next higher stream than the one on which the mes
sage was received.

ii-Create an SCTP socket
An SCTP one-to-many-style socket is created.

iii-Bind an address
An Internet socket address structure is filled in with the wildcard address (INADDR_ANY) a
nd the server's well-known port, SERV_PORT. Binding the wildcard address tells the system
 that this SCTP endpoint will use all available local addresses in any association that is
set up.

iv- Set up for notifications of interest
The server changes its notification subscription for the one-to-many SCTP socket. The serv
er subscribes to just the sctp_data_io_event, which will allow the server to see the sctp_
sndrcvinfo structure.

v- Enable incoming associations
The server enables incoming associations with the listen call. Then, control enters the ma
in processing loop.

vi- Wait for message
The server initializes the size of the client socket address structure, then blocks while
 waiting for a message from any remote peer.

vii - Increment stream number if desired
When a message arrives, the server checks the stream_increment flag to see if it should in
crement the stream number. If the flag is set (no arguments were passed on the command lin
e), the server increments the stream number of the message.

viii-Send back response
35–38 The server sends back the message using the payload protocol ID, flags, and the poss
ibly modified stream number from the sri structure.
This program runs forever until the user shuts it down with an external signal.

11. SCTP One-to-Many-Style Streaming Echo Client: main Function

Validate arguments and create a socket
i-The client validates the arguments passed to it. First, the client verifies that the cal
ler provided a host to send messages to. It then checks if the "echo to all" option is bei
ng enabled

ii- Set up server address
The client translates the server address, passed on the command line, using the inet_pton
```

```
 function. It combines that with the server's well-known port number and uses the resultin
g address as the destination for the requests.

iii-Set up for notifications of interest
The client explicitly sets the notification subscription provided by our one-to-many SCTP
 socket. Again, it wants no MSG_NOTIFICATION events. Therefore, the client turns these off
(as was done in the server) and only enables the receipt of the sctp_sndrcvinfo structure.

vi -Call echo processing function
If the echo_to_all flag is not set, the client calls the sctpstr_cli function, discussed i
n Section 10.4. If the echo_to_all flag is set, the client calls the sctpstr_cli_echoall f
unction. We will discuss this function in Section 10.5 as we explore uses for SCTP stream
s.

v-Finish up
On return from processing, the client closes the SCTP socket, which shuts down any SCTP as
sociations using the socket. The client then returns from main with a return code of 0, in
dicating that the program ran successfully.
```

12. Determining Outgoing Interface with UDP

```
int
main(int argc, char **argv)
{
    int     sockfd;
    socklen_t len;
    struct sockaddr_in cliaddr, servaddr;

    if (argc != 2)
        err_quit("usage: udpcli <IPaddress>");

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

    len = sizeof(cliaddr);
    Getsockname(sockfd, (SA *) &cliaddr, &len);
    printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

    exit(0);
}
```

-The first time we run the program, the command-line argument is an IP address that follows the default route. The kernel assigns the local IP address to the primary address of the interface to which the default route points.
-The second time, the argument is the IP address of a system connected to a second Ethernet interface, so the kernel assigns the local IP address to the primary address of this second interface. Calling connect on a UDP socket does not send anything to that host

13. **TCP and UDP Echo Server Using select- nah fam, its too big, fk it**