BHARTIYA INSTITUTE OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

8CS5A UNIX NETWORK PROGRAMMING & SIMULATION LAB

Network programming plays a vital part in development of peer-to-peer transmission of any kind message in any form. Whenever there is a need to communicate between client and the server, socket programming plays an important role.

Socket programming works on both TCP and UDP protocols. There are numerous applications like chat or FTP applications, which use socket programming for networking. Let us take an example and explore what socket programming is:

Consider an example of chat between you and your friend on Facebook:

1. Facebook chat panel is considered as an “Endpoint” for communication.

2. Both are having unique ID like your login “Address” and both are ready to listen any kind of chat in any form.

3. Whenever you want to chat with your friend, you simply click on your friend’s icon in the list and start sending a message

just like a “Caller”.

4. After your message is sent, your friend will start receiving chat as a “Receiver”.

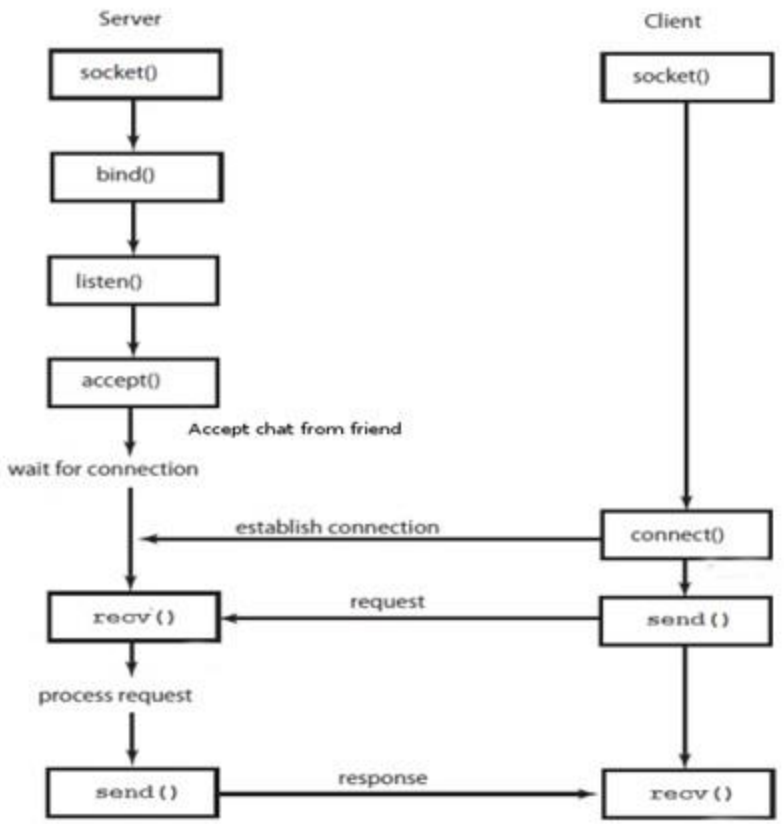
5. As soon as this channel is established, you can start “Communicating” with your friend.

6. Once you are finished with the conversation, you can “Close” the chat window anytime.

So in terms of Socket programming below analogies are used:

- Socket() : Endpoint for communication
- Bind() : Assign a unique ID.
- Listen() : Wait for a chat.
- Connect() : Initialize chat.
- Accept() : Accept chat from friend.
- Send(), Recv() : Start Chatting.
- Close() : Hang up.

TCP connection process:



Data Structure:

1.

Struct sockaddr

```
{
    unsigned short sa_family;
    char sa_data[14];
}
```

2.

Struct sockaddr_in

```
{
    short sin_family;
    unsigned short sin_port; // Port Number
    struct in_addr sin_addr; // IP Address
```

```
    char sin_zero[8];
}
```

3.

```
Struct in_addr

{
    unsigned long s_addr; // 4 bytes long
}
```

Functions used in C for socket programming:

1. Socket Function : Endpoint

This function creates an endpoint for network connections.

Int Socket(int serverDomain, int typeOfConnection, int protocol)

serverDomain = PF_INET (IPv4 communication)

typeOfConnection = SOCK_STREAM (TCP) or SOCK_DGRAM (UDP)

protocol = 0 (For basic connection)

Return : Descriptor on success and -1 on an error.

2. Bind Function : Bind IP and Port

Int Bind(int sockFuncDescriptor, struct sockaddr *server_addr, socklen_t *addressLength)

sockFuncDescriptor = socket descriptor returned by socket()

server_addr = pointer to a valid sockaddr_in structure cast as a sockaddr * pointer

addressLength= length of the sockaddr_in structure

3. Listen Function : Wait for connection

Int Listen(int sock, int MaxConn)

sock = socket returned by socket()

MaxConn = Maximum length of the pending connections queue

4. Accept Function : Accept new connection from client.

Int Accept(int sock, (struct sockaddr *)&client, socklen_t *client_len)

sock = the socket in listen state

client = A new client detail when accept returns

client_len = pointer to size of the client structure

5. Connect Function : Connect to client

Int Connect(int socker, (struct sockaddr *)&server_addr, socklen_t length)

socket : Return from socket()

server_addr : a sockaddr_in struct pointer filled with all the remote server details and cast as a sockaddr struct pointer

length : size of the server_addr struct

6. Send and Receive Function : As name depict

Int send(int socket, void *message, size_t length, int flag)

socket = A connected socket

message = Pointer to a message.

length = Size of the message buffer

flag = 0 (for now)

7. Close Function:

Int close(int socket)
socket = The socket that need to be close.

1. Write two programs in C: hello_client and hello_server

- The server listens for, and accepts, a single TCP connection; it reads all the data it can from that connection, and prints it to the screen; then it closes the connection

PROGRAM:

TCP Server

```
#include <unistd.h>

#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
   int server_fd, new_socket, valread;
   struct sockaddr_in address;
   int opt = 1;
   int addrlen = sizeof(address);
   char buffer[1024] = {0};
   char *hello = "Hello from server";
   // Creating socket file descriptor
   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
   {  perror("socket failed");
      exit(EXIT_FAILURE);
   }

   // Forcefully attaching socket to the port 8080
   if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,&opt, sizeof(opt)))
   {   perror("setsockopt");
      exit(EXIT_FAILURE);
   }
   address.sin_family = AF_INET;
   address.sin_addr.s_addr = INADDR_ANY;
   address.sin_port = htons( PORT );

   // Forcefully attaching socket to the port 8080
   if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))<0)
   {
      perror("bind failed");
      exit(EXIT_FAILURE);
```

```
    }
    if (listen(server_fd, 3) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen))<0)
    { perror("accept");
        exit(EXIT_FAILURE);
    }
    valread = read( new_socket , buffer, 1024);
    printf("%s\n",buffer );
    send(new_socket , hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
    return 0;
}

// Tcp Client


#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    struct sockaddr_in address;
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *hello = "Hello from client";
    char buffer[1024] = {0};
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
```

```
    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }
    send(sock , hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
    valread = read( sock , buffer, 1024);
    printf("%s\n",buffer );
    return 0;
}
```

• The client connects to the server, sends the string “Hello, world!”, then closes the connection

```
#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<errno.h>

main()

{

        int sock, cli;

        struct sockaddr_in server, clientDetail;
```

```
        unsigned int len;
        char mesg[] = “Hello world with Socket programming!”;
        int sentconf;
         if((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
                perror(“There are some issue in creating socket : “);
                exit(-1);
        }
         server.sin_family = AF_INET;
        server.sin_port = htons(10012);
        server.sin_addr.s_addr = INADDR_ANY;
        bzero(&server.sin_zero, 8);
       len = sizeof(struct sockaddr_in);
         if((bind(sock, (struct sockaddr *)&server, len)) == -1)
        {
                perror(“Error in Binding”);
                exit(-1);
        }
         if((listen(sock, 5)) == -1)
        {
          perror(“Error in Listening”);
                exit(-1);
        }
```

```
        while(1)
        {
                if((cli = accept(sock, (struct sockaddr *)&clientDetail, &len)) ==-1)
                {
          perror("Error in Accepting");
                        exit(-1);
                }
                 sentconf =  send(cli, mesg, strlen(mesg), 0);
                printf("Sending %d bytes to clientDetail : %c\n", sentconf,
                                        inet_ntoa(clientDetail.sin_addr));
                close(cli);
        }
}-
```

2. Write an Echo_Client and Echo_server using TCP to estimate the round trip time from client to the server. The server should be such that it can accept multiple connections at any given time.

   PROGRAM:

TCP Echo Server: main Function

```c
#include    "unp.h"

int
main(int argc, char **argv)
{
    int                 listenfd, connfd;
```

```
    pid_t               childpid;
    socklen_t           clilen;
    struct sockaddr_in  cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 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(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

        if ( (childpid = Fork()) == 0) {    /* child process */
            Close(listenfd);    /* close listening socket */
            str_echo(connfd);   /* process the request */
            exit(0);
        }
        Close(connfd);          /* parent closes connected socket */
    }
}
```

TCP Echo Server: str_echo Function

```
#include    "unp.h"

void
str_echo(int sockfd)
{
    ssize_t     n;
    char        buf[MAXLINE];

again:
    while ( (n = read(sockfd, buf, MAXLINE)) > 0)
        Writen(sockfd, buf, n);

    if (n < 0 && errno == EINTR)
        goto again;
    else if (n < 0)
        err_sys("str_echo: read error");
}
```

TCP 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: tcpcli <IPaddress>");

    sockfd = Socket(AF_INET, SOCK_STREAM, 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));

    str_cli(stdin, sockfd);     /* do it all */

    exit(0);
}
```

TCP Echo Client: str_cli Function

```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    char    sendline[MAXLINE], recvline[MAXLINE];

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

        Writen(sockfd, sendline, strlen(sendline));
```

```
        if (Readline(sockfd, recvline, MAXLINE) == 0)
            err_quit("str_cli: server terminated prematurely");

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

3. Repeat Exercises 1 & 2 for UDP.

PROGRAM:

```
// Server program
#include <arpa/inet.h>
#include <errno.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#define PORT 5000
#define MAXLINE 1024
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
int main()
{
int listenfd, connfd, udpfd, nready, maxfdp1;
  char buffer[MAXLINE];
  pid_t childpid;
  fd_set rset;
  ssize_t n;
  socklen_t len;
  const int on = 1;
  struct sockaddr_in cliaddr, servaddr;
```

```
char* message = "Hello Client";
void sig_chld(int);

/* create listening TCP socket */
listenfd = socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

// binding server addr structure to listenfd
bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
listen(listenfd, 10);

/* create UDP socket */
udpfd = socket(AF_INET, SOCK_DGRAM, 0);
// binding server addr structure to udp sockfd
bind(udpfd, (struct sockaddr*)&servaddr, sizeof(servaddr));

// clear the descriptor set
FD_ZERO(&rset);

// get maxfd
maxfdp1 = max(listenfd, udpfd) + 1;
for (;;) {

   // set listenfd and udpfd in readset
   FD_SET(listenfd, &rset);
   FD_SET(udpfd, &rset);

   // select the ready descriptor
   nready = select(maxfdp1, &rset, NULL, NULL, NULL);

   // if tcp socket is readable then handle
   // it by accepting the connection
   if (FD_ISSET(listenfd, &rset)) {
      len = sizeof(cliaddr);
      connfd = accept(listenfd, (struct sockaddr*)&cliaddr, &len);
      if ((childpid = fork()) == 0) {
         close(listenfd);
         bzero(buffer, sizeof(buffer));
         printf("Message From TCP client: ");
         read(connfd, buffer, sizeof(buffer));
         puts(buffer);
```

```
            write(connfd, (const char*)message, sizeof(buffer));
            close(connfd);
            exit(0);

                    }
         close(connfd);
      }
      // if udp socket is readable receive the message.
      if (FD_ISSET(udpfd, &rset)) {
         len = sizeof(cliaddr);
         bzero(buffer, sizeof(buffer));
         printf("\nMessage from UDP client: ");
         n = recvfrom(udpfd, buffer, sizeof(buffer), 0,
                  (struct sockaddr*)&cliaddr, &len);
         puts(buffer);
         sendto(udpfd, (const char*)message, sizeof(buffer), 0,
               (struct sockaddr*)&cliaddr, sizeof(cliaddr));
      }
   }
}


/ UDP client program
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/socket.h>
#include <sys/types.h>
#define PORT 5000
#define MAXLINE 1024
int main()
{
   int sockfd;
   char buffer[MAXLINE];
   char* message = "Hello Server";
   struct sockaddr_in servaddr;

   int n, len;
   // Creating socket file descriptor
   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      printf("socket creation failed");
      exit(0);
```

```
    }

    memset(&servaddr, 0, sizeof(servaddr));
// Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    // send hello message to server
    sendto(sockfd, (const char*)message, strlen(message),
        0, (const struct sockaddr*)&servaddr,
        sizeof(servaddr));

    // receive server's response
    printf("Message from server: ");
    n = recvfrom(sockfd, (char*)buffer, MAXLINE,
            0, (struct sockaddr*)&servaddr,
            &len);
    puts(buffer);
    close(sockfd);
    return 0;
}
```

.

4. Repeat Exercise 2 with multiplexed I/O operations

TCP/UDP Echo Server using I/O Multiplexing

We have used the concept of I/O multiplexing for managing both TCP and UDP port in the same server.

```
    fd_set fdvar;
    FD_ZERO(&fdvar);
    FD_SET(tcp_sfd,&fdvar);
    FD_SET(udp_sfd,&fdvar);
    int maxpl = max(tcp_sfd,udp_sfd);
    cout << "Waiting for a client...\n";

    if(select(maxpl+2 ,&fdvar,NULL,NULL,NULL)==-1)
    {
        perror("error in select");
```

```
}
if(FD_ISSET(udp_sfd,&fdvar))
{
   // UDP
}
else
{
   //TCP
}
```

5. Simulate Bellman-Ford Routing algorithm in NS2

Bellman Ford's Algorithm

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

Bellman Ford Pseudocode

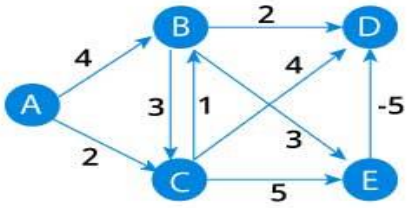
We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path

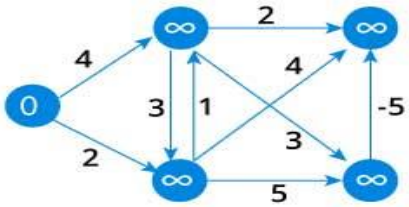
1

Start with a weighted graph



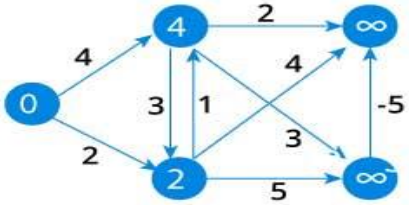
2

Choose a starting vertex and assign infinity path values to all other vertices



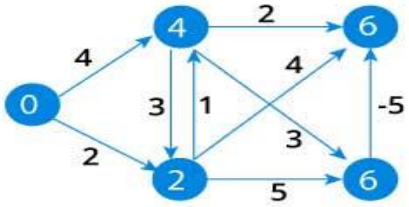
3

Visit each edge and relax the path distances if they are inaccurate



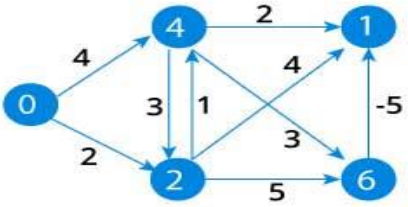
4

We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



5

Notice how the vertex at the top right corner had its path length adjusted



6

After all the vertices have their path lengths, we check if a negative cycle is present.

A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

```
function bellmanFord(G, S)
        for each vertex V in G
                        distance[V] <- infinite
                        previous[V] <- NULL
        distance[S] <- 0
        for each vertex V in G
                for each edge (U,V) in G
                        tempDistance <- distance[U] + edge_weight(U, V)
                        if tempDistance < distance[V]
                           distance[V] <- tempDistance
                           previous[V] <- U

        for each edge (U,V) in G
                If distance[U] + edge_weight(U, V) < distance[V}
                        Error: Negative Cycle Exists

        return distance[], previous[]
```

Bellman Ford's Algorithm Code

```
#include <stdio.h>
#include <stdlib.h>

#define INFINITY 99999

//struct for the edges of the graph
struct Edge {
        int u;      //start vertex of the edge
        int v;      //end vertex of the edge
        int w;      //weight of the edge (u,v)
};

//Graph - it consists of edges
struct Graph {
        int V;      //total number of vertices in the graph
        int E;      //total number of edges in the graph
        struct Edge *edge;//array of edges
};

void bellmanford(struct Graph *g, int source);
void display(int arr[], int size);
```

```
int main(void) {
	//create graph
struct Graph *g = (struct Graph*)malloc(sizeof(struct Graph));
	g->V = 4; //total vertices
	g->E = 5;  //total edges
	//array of edges for graph
	g->edge = (struct Edge*)malloc(g->E * sizeof(struct Edge));

	//------- adding the edges of the graph
	/*
		edge(u, v)
		where	u = start vertex of the edge (u,v)
				v = end vertex of the edge (u,v)

		w is the weight of the edge (u,v)
	*/

	//edge 0 --> 1
	g->edge[0].u = 0;
	g->edge[0].v = 1;
	g->edge[0].w = 5;

	//edge 0 --> 2
	g->edge[1].u = 0;
	g->edge[1].v = 2;
	g->edge[1].w = 4;
//edge 1 --> 3
	g->edge[2].u = 1;
	g->edge[2].v = 3;
	g->edge[2].w = 3;

	//edge 2 --> 1
	g->edge[3].u = 2;
	g->edge[3].v = 1;
	g->edge[3].w = -6;

	//edge 3 --> 2
	g->edge[4].u = 3;
	g->edge[4].v = 2;
	g->edge[4].w = 2;
```

```
	bellmanford(g, 0);		//0 is the source vertex

	return 0;
}
void bellmanford(struct Graph *g, int source) {
	//variables
	int i, j, u, v, w;

	//total vertex in the graph g
	int tV = g->V;

	//total edge in the graph g
	int tE = g->E;
//distance array
	//size equal to the number of vertices of the graph g
	int d[tV];

	//predecessor array
	//size equal to the number of vertices of the graph g
	int p[tV];

	//step 1: fill the distance array and predecessor array
	for (i = 0; i < tV; i++) {
		d[i] = INFINITY;
		p[i] = 0;
}

	//mark the source vertex
	d[source] = 0;

	//step 2: relax edges |V| - 1 times
	for(i = 1; i <= tV-1; i++) {
		for(j = 0; j < tE; j++) {
			//get the edge data
			u = g->edge[j].u;
			v = g->edge[j].v;
			w = g->edge[j].w;

			if(d[u] != INFINITY && d[v] > d[u] + w) {
				d[v] = d[u] + w;
				p[v] = u;
			}
```

```
                }
        }

        //step 3: detect negative cycle
        //if value changes then we have a negative cycle in the graph
        //and we cannot find the shortest distances
        for(i = 0; i < tE; i++) {
                u = g->edge[i].u;
                v = g->edge[i].v;
                w = g->edge[i].w;
                if(d[u] != INFINITY && d[v] > d[u] + w) {
                        printf("Negative weight cycle detected!\n");
                        return;
                }
        }

        //No negative weight cycle found!
        //print the distance and predecessor array
        printf("Distance array: ");
        display(d, tV);
        printf("Predecessor array: ");
        display(p, tV);
}

void display(int arr[], int size) {
        int i;
        for(i = 0; i < size; i ++) {
                printf("%d ", arr[i]);
        }
        printf("\n");
}
```