Experiment 7

Implementation of Client-Server communication using Socket Programming and TCP as transport layer protocol

Aim: Client sends a string to the server using tcp protocol. The server reverses the string and returns it to the client, which then displays the reversed string.

Description:

Steps for creating a TCP connection by a client are:

1. **Creation of client socket**

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

This function call creates a socket and returns a socket descriptor. The domain parameter specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <sys/socket.h>. In this program, the domain **AF_INET** is used. The socket has the indicated type, which specifies the communication semantics. **SOCK_STREAM** type provides sequenced, reliable, two-way, connection based byte streams. The **protocol** field specifies the protocol used. We always use 0. If the system call is a failure, a -1 is returned. The header files used are **sys/types.h** and **sys/socket.h**.

2. **Filling the fields of the server address structure.**

The socket address structure is of type **struct sockaddr_in**.

```
struct sockaddr_in {
                u_short sin_family;
                u_short sin_port;
                struct in_addr sin_addr;
                char sin_zero[8];        /*unused, always zero*/
              };

struct in_addr {
              u_long s_addr;
};
```

The fields of the socket address structure are
sin_family which in our case is AF_INET
sin_port which is the port number where socket binds
sin_addr which is the IP address of the server machine

The header file that is to be used is **netinet/in.h**

Example

struct sockaddr_in servaddr;
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(port_number);

Why htons is used ?. Numbers on different machines may be represented differently (big-endian machines and little-endian machines). In a little-endian machine the low order byte of an integer appears at the lower address; in a big-endian machine instead the low order byte appears at the higher address. Network order, the order in which numbers are sent on the internet is big-endian. It is necessary to ensure that the right representation is used on each machine. Functions are used to convert from host to network form before transmission- htons for short integers and htonl for long integers.

The value for servaddr.sin_addr is assigned using the following function

inet_pton(AF_INET, "IP_Address", &servaddr.sin_addr);

The binary value of the dotted decimal IP address is stored in the field when the function returns.

3. **Binding of the client socket to a local port**

This is optional in the case of client and we usually do not use the *bind* function on the client side.

4. Connection of client to the server

A server is identified by an IP address and a port number. The connection operation is used on the client side to identify and start the connection to the server.

int connect(int sd, struct sockaddr * addr, int addrlen);

sd – file descriptor of local socket
addr – pointer to protocol address of other socket
addrlen – length in bytes of address structure

The header files to be used are sys/types.h and sys/socket.h

It returns 0 on sucess and -1 in case of failure.

5. **Reading from socket**
In the case of TCP connection reading from a socket can be done using the *read* system call

int read(int sd, char * buf, int length);

6. writing to a socket

In the case of TCP connection writing to a socket can be done using the *write* system call

int write(int sd, char * buf, int length);

7. **closing the connection**

The connection can be closed using the close system call

int close(int sd);

Steps for TCP Connection for server

1. **Creating a listening socket**

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

This system call creates a socket and returns a socket descriptor. The *domain* field used is **AF_INET**. The socket type is **SOCK_STREAM**. The **protocol** field is 0. If the system call is a failure, a -1 is returned. Header files used are **sys/types.h** and **sys/socket.h**.

2. **Binding to a local port**

int bind(int sd, struct sockaddr * addr, int addrlen);

This call is used to specify for a socket the protocol port number where it will wait for messages. A call to *bind* is optional on the client side, but required on the server side. The first field is the *socket* descriptor of local socket. Second is a pointer to protocol address structure of this socket. The third is the length in bytes of the structure referenced by *addr.* This system call returns an integer. It is 0 for success and -1 for failure. The header files are **sys/types.h** and **sys/socket.h**.

3. **Listening on the port**

The listen function is used on the server in the connection oriented communication to prepare a socket to accept messages from clients.

int listen(int fd, int qlen);

fd – file descriptor of a socket that has already been bound
qlen – specifies the maximum number of messages that can wait to be processed by the server while the server is busy servicing another request. Usually it is taken as 5. The header files used are *sys/types.h* and *sys/socket.h.* This function returns 0 on success and -1 on failure.

4. **Accepting a connection from the client**

The accept function is used on the server in the case of connection oriented communication to accept a connection request from a client.

int accept(int fd, struct sockaddr * addressp, int * addrlen);

The first field is the descriptor of server socket that is listening. The second parameter *addressp* points to a socket address structure that will be filled by the address of calling client when the function returns. The third parameter *addrlen* is an integer that will contain the actual length of address structure of client. It returns an integer that is a descriptor of a new socket called the connection socket. Server sockets send data and read data from this socket. The header files used are *sys/types.h* and *sys/socket.h.*

Algorithm

Client

1. Create socket
2. Connect the socket to the server

3. Read the string to be reversed from the standard input and send it to the server
Read the matrices from the standard input and send it to server using socket

4. Read the reversed string from the socket and display it on the standard output
Read product matrix from the socket and display it on the standard output
5. Close the socket

Server

1. Create listening socket

2. bind IP address and port number to the socket

3. listen for incoming requests on the listening socket

4. accept the incoming request

5. connection socket is created when *accept* returns

6. Read the string using the connection socket from the client

7. Reverse the string

8. Send the string to the client using the connection socket

9. close the connection socket

10. close the listening socket

Client Program

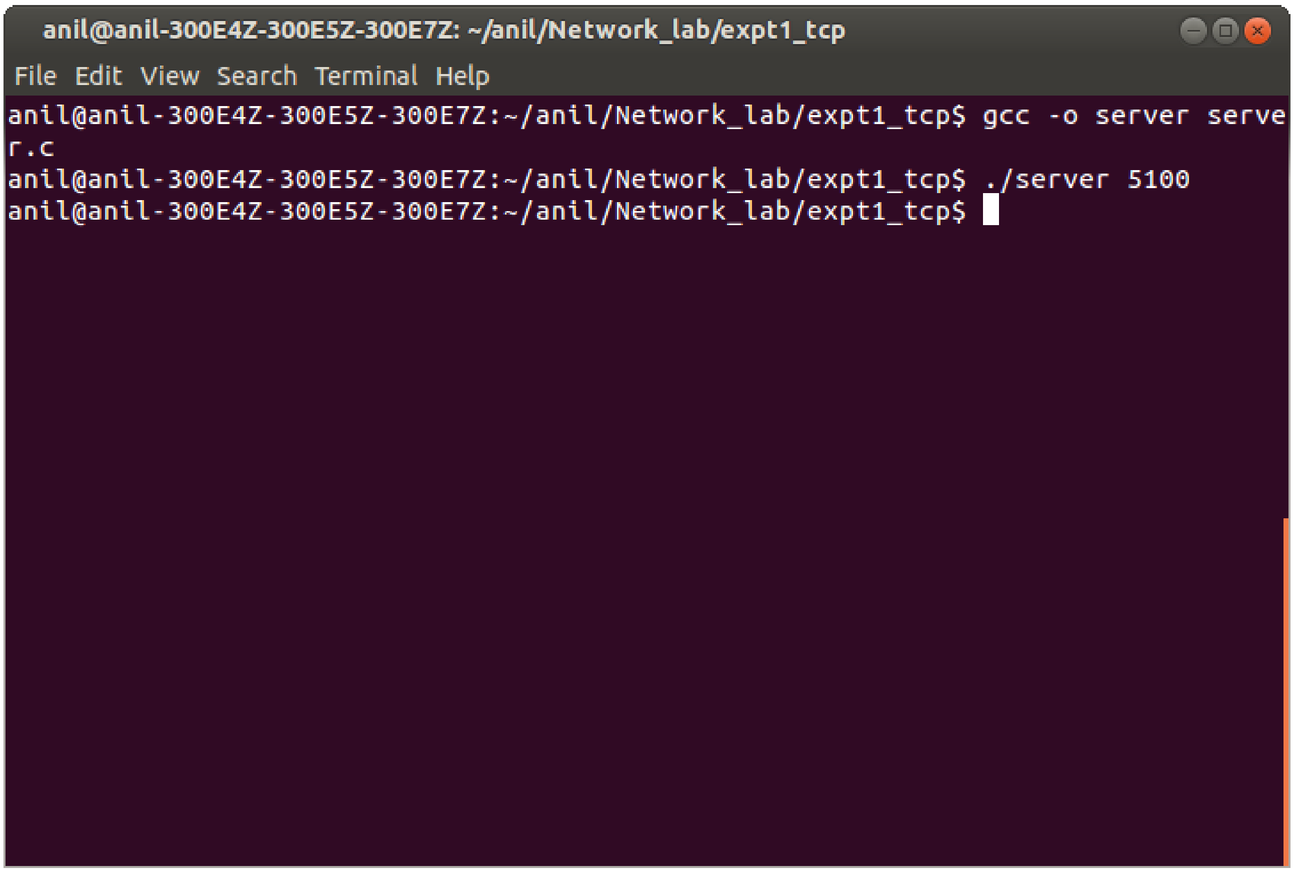
```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>


int main( int argc, char *argv[])
{

struct sockaddr_in server;
int sd ;
char buffer[200];

if((sd   = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
   perror("Socket failed:");
   exit(1);
}

// server socket address structure initialisation

bzero(&server, sizeof(server) );
server.sin_family = AF_INET;
server.sin_port = htons(atoi(argv[2]));
inet_pton(AF_INET, argv[1], &server.sin_addr);

if(connect(sd, (struct sockaddr *)&server, sizeof(server))< 0)
{
   perror("Connection failed:");
   exit(1);
}

fgets(buffer, sizeof(buffer), stdin);
buffer[strlen(buffer) - 1] = '\0';

write (sd,buffer, sizeof(buffer));
read(sd,buffer, sizeof(buffer));

printf("%s\n", buffer);
close(fd);

}
```

Server Program

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>


int main( int argc, char *argv[])
{

struct sockaddr_in server, cli;
int cli_len;
int sd, n, i, len;
int data, temp;

char buffer[100];

if((sd   = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
   perror("Socket failed:");
   exit(1);
}


// server socket address structure initialisation

bzero(&server, sizeof(server) );
server.sin_family = AF_INET;
server.sin_port = htons(atoi(argv[1]));
server.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(sd, (struct sockaddr*)&server, sizeof(server)) < 0)
{
   perror("bind failed:");
   exit(1);
}
listen(sd,5);

if((data =  accept(sd , (struct sockaddr *) &cli, &cli_len)) < 0)
{
    perror("accept failed:");
    exit(1);
}

read(data,buffer, sizeof(buffer));

len = strlen(buffer);
for( i =0; i<= len/2; i++)
 {
   temp =  buffer[i];
   buffer[i] = buffer[len - 1-i];
   buffer[len-1-i] = temp;
}

write (data,buffer, sizeof(buffer));

close(data);
close(sd);
}
```

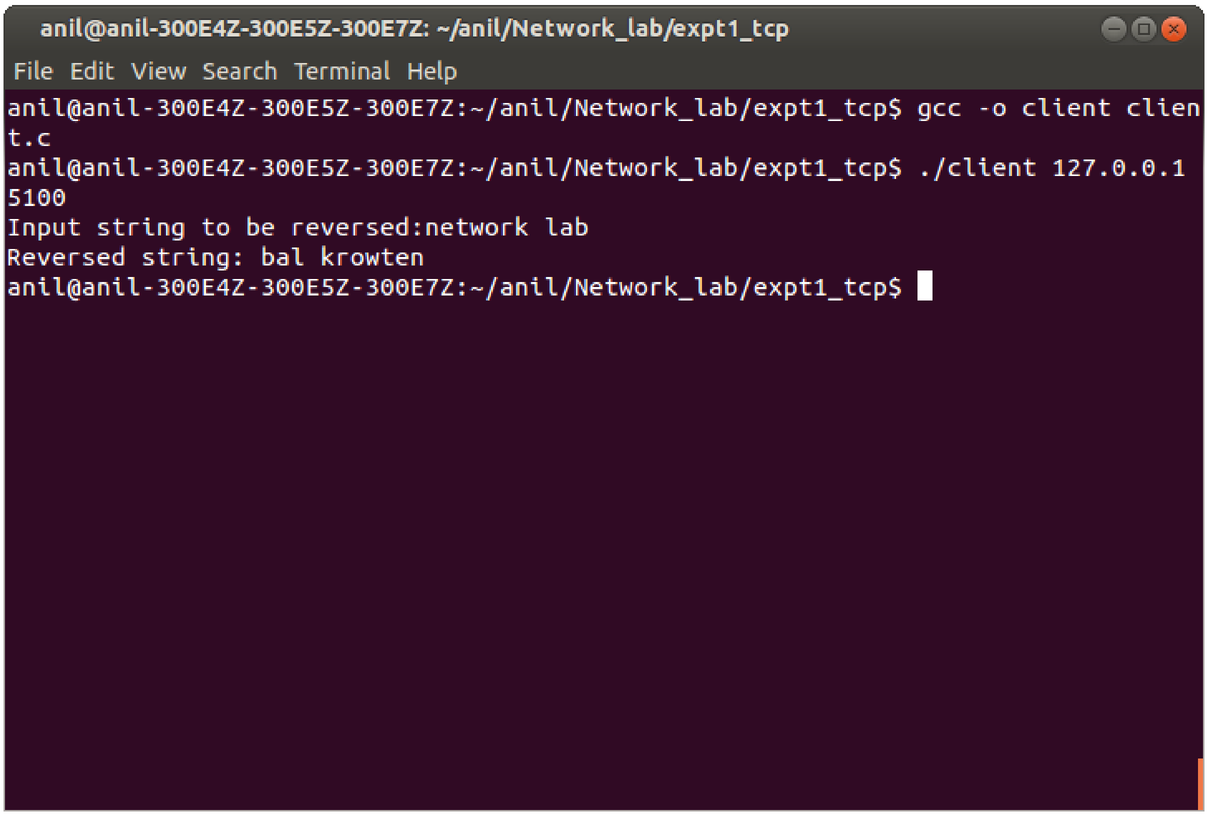
Output

Server


anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt1_tcp
File Edit View Search Terminal Help

```
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt1_tcp$ gcc -o server serve
r.c
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt1_tcp$ ./server 5100
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt1_tcp$ 
```

Client


anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt1_tcp
File Edit View Search Terminal Help

```
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt1_tcp$ gcc -o client clien
t.c
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt1_tcp$ ./client 127.0.0.1
5100
Input string to be reversed:network lab
Reversed string: bal krowten
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt1_tcp$ 
```

Experiment 8

Implementation of Client-Server communication using Socket Programming and UDP as transport layer protocol

Aim: Client sends two matrices to the server using udp protocol. The server multiplies the matrices and sends the product to the client, which then displays the product matrix.

Description:

Steps for transfer of data using UDP

1. **Creation of UDP socket**

The function call for creating a UDP socket is

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

The *domain* parameter specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <sys/socket.h>. In this program, the domain **AF_INET** is used. The next field *type* has the value **SOCK_DGRAM.** It supports datagrams (connectionless, unreliable messages of a fixed maximum length). The *protocol* field specifies the protocol used. We always use 0. If the socket function call is successful, a socket descriptor is returned. Otherwise -1 is returned. The header files necessary for this function call are **sys/types.h** and **sys/socket.h**.

2. **Filling the fields of the server address structure.**

The socket address structure is of type **struct sockaddr_in**.

```
struct sockaddr_in {
                u_short sin_family;
                u_short sin_port;
                struct in_addr sin_addr;
                char sin_zero[8];          /*unused, always zero*/
              };

struct in_addr {
              u_long s_addr;
};
```

The fields of the socket address structure are
sin_family which in our case is AF_INET
sin_port which is the port number where socket binds
sin_addr is used to store the IP address of the server machine and is of type **struct in_addr**

The header file that is to be used is **netinet/in.h**

The value for **servaddr.sin_addr** is assigned using the following function

inet_pton(AF_INET, “IP_Address”, &servaddr.sin_addr);

The binary value of the dotted decimal IP address is stored in the field when the function returns.

3. **Binding of a port to the socket in the case of server**

This call is used to specify for a socket the protocol port number where it will wait for messages. A call to bind is optional in the case of client and compulsory on the server side.

int bind(int sd, struct sockaddr* addr, int addrlen);

The first field is the socket descriptor. The second is a pointer to the address structure of this socket. The third field is the length in bytes of the size of the structure referenced by *addr*. The header files are **sys/types.h** and **sys/socket.h**. This function call returns an integer, which is 0 for success and -1 for failure.

4. **Receiving data**

ssize_t recvfrom(int s, void * buf, size_t len, int flags, struct sockaddr * from, socklen_t * fromlen);

The *recvfrom* calls are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection oriented. The first parameter *s* is the socket descriptor to read from. The second parameter *buf* is the buffer to read information into. The third parameter *len* is the maximum length of the buffer. The fourth parameter is *flag*. It is set to zero. The fifth parameter *from* is a pointer to **struct sockaddr** variable that will be filled with the IP address and port of the orginating machine. The sixth parameter *fromlen* is a pointer to a local int variable that should be initialized to **sizeof(struct sockaddr)**. When the function returns, the integer variable that *fromlen* points to will contain the actual number of bytes that is contained in the socket address structure. The header files required are **sys/types.h** and **sys/socket.h**. When the function returns, the number of bytes received is returned or -1 if there is an error.

5. **Sending data**

sendto- sends a message from a socket

ssize_t sendto(int s, const void * buf, size_t len, int flags, const struct sockaddr * to, socklen_t tolen);

The first parameter *s* is the socket descriptor of the sending socket. The second parameter *buf* is the array which stores data that is to be sent. The third parameter *len* is the length of that data in bytes. The fourth parameter is the *flag* parameter. It is set to zero. The fifth parameter *to* points to a variable that contains the destination IP address and port. The sixth parameter *tolen* is set to **sizeof(struct sockaddr)**. This function returns the number of bytes actually sent or -1 on error. The header files used are **sys/types.h** and **sys/socket.h**.

Algorithm

Client

1. Create socket

2. Read the matrices from the standard input and send it to server using socket

3. Read product matrix from the socket and display it on the standard output

4. Close the socket

Server

1. Create socket

2. bind IP address and port number to the socket

3. Read the matrices socket from the client using socket

4. Find product of matrices

5. Send the product matrix to the client using socket

6. close the socket

Client program

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<stdlib.h>
main(int argc, char * argv[])
{
  int i,j,n;
  int sock_fd;

  struct sockaddr_in servaddr;
  int matrix_1[10][10], matrix_2[10][10], matrix_product[10][10];
  int size[2][2];

  int num_rows_1, num_cols_1, num_rows_2, num_cols_2;

  if(argc != 3)
  {
    fprintf(stderr, "Usage: ./client IPaddress_of_server port\n");
    exit(1);
  }

  printf("Enter the number of rows of first matrix\n");
  scanf("%d", &num_rows_1);

  printf("Enter the number of columns of first matrix\n");
  scanf("%d", &num_cols_1);

  printf("Enter the values row by row one on each line\n" );

  for ( i = 0; i < num_rows_1; i++)
  for( j=0; j<num_cols_1; j++)
  {
     scanf("%d", &matrix_1[i][j]);
  }

  size[0][0] = num_rows_1;
  size[0][1] = num_cols_1;

  printf("Enter the number of rows of second matrix\n");
  scanf("%d", &num_rows_2);

  printf("Enter the number of columns of second matrix\n");
  scanf("%d", &num_cols_2);

 if( num_cols_1 != num_rows_2)
  {
     printf("MATRICES CANNOT BE MULTIPLIED\n");
     exit(1);
  }

  printf("Enter the values row by row one on each line\n");

  for (i = 0; i < num_rows_2; i++)
  for(j=0; j<num_cols_2; j++)
  {
     scanf("%d", &matrix_2[i][j]);
  }

   size[1][0] = num_rows_2;
   size[1][1] = num_cols_2;

  if((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   {
        printf("Cannot create socket\n");
        exit(1);
   }

  bzero((char*)&servaddr, sizeof(servaddr));

  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(atoi(argv[2]));
  inet_pton(AF_INET, argv[1], &servaddr.sin_addr);


 // SENDING MATRIX  WITH SIZES OF MATRICES 1 AND 2

 n = sendto(sock_fd, size, sizeof(size),0, (struct sockaddr*)&servaddr, sizeof(servaddr));

 if( n < 0)
 {
    perror("error in matrix 1  sending");
    exit(1);
 }

 // SENDING MATRIX 1
 n = sendto(sock_fd, matrix_1, sizeof(matrix_1),0, (struct sockaddr*)&servaddr, sizeof(servaddr));

 if( n < 0)
 {
    perror("error in matrix 1  sending");
    exit(1);
 }

 // SENDING MATRIX 2

 n = sendto(sock_fd, matrix_2, sizeof(matrix_2),0, (struct sockaddr*)&servaddr, sizeof(servaddr));

 if( n < 0)
 {
    perror("error in matrix 2  sending");
    exit(1);
  }

  if((n=recvfrom(sock_fd, matrix_product, sizeof(matrix_product),0, NULL, NULL)) == -1)
  {
    perror("read error from server:");
    exit(1);
 }

printf("\n\nTHE PRODUCT OF MATRICES IS \n\n\n");

for( i=0; i < num_rows_1; i++)
{
for( j=0; j<num_cols_2; j++)
{
   printf("%d ",matrix_product[i][j]);
}
printf("\n");
}

close(sock_fd);

}
```

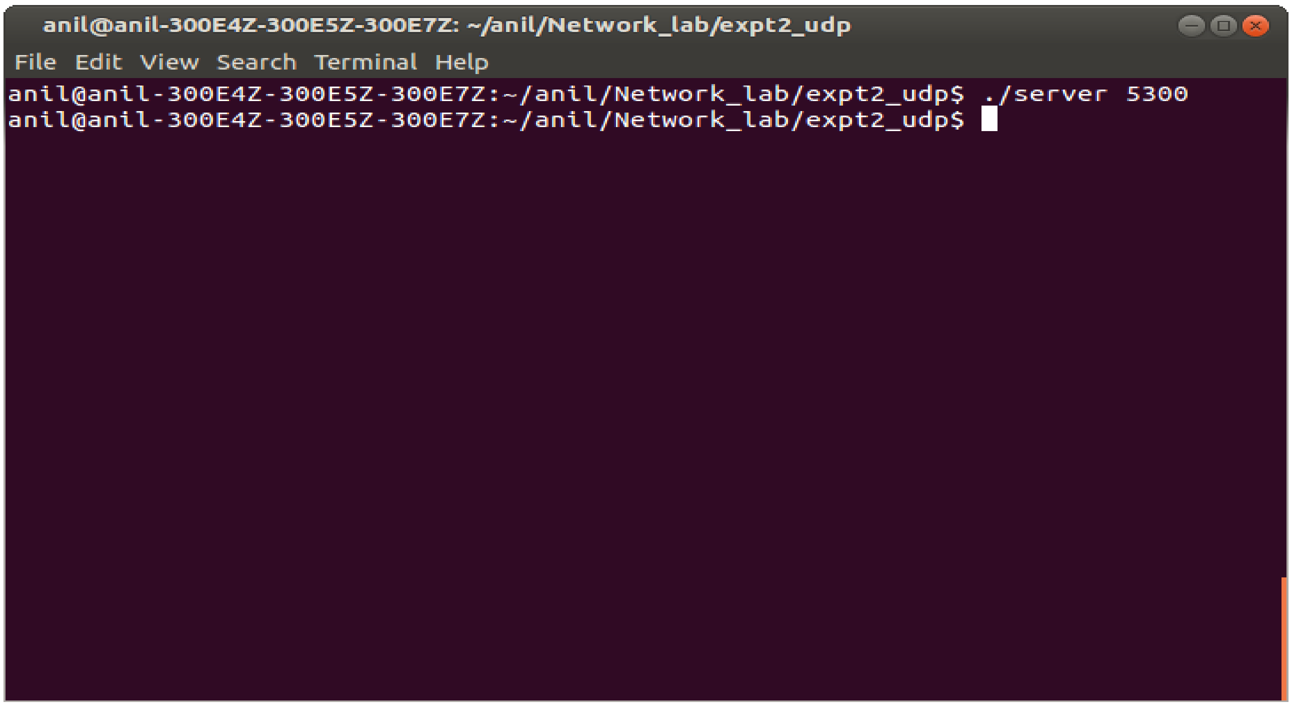
Server Program

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<stdlib.h>


main(int argc, char * argv[])
{

   int n;
   int sock_fd;
   int i,j,k;

   int row_1, row_2, col_1, col_2;

   struct sockaddr_in servaddr, cliaddr;
   int len = sizeof(cliaddr);

   int matrix_1[10][10], matrix_2[10][10], matrix_product[10][10];
   int size[2][2];

   if(argc != 2)
   {
     fprintf(stderr, "Usage: ./server  port\n");
     exit(1);
   }



   if((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
          printf("Cannot create socket\n");
          exit(1);
    }

   bzero((char*)&servaddr, sizeof(servaddr));

   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(atoi(argv[1]));
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

  if(bind(sock_fd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0)
  {
    perror("bind failed:");
    exit(1);
  }

// MATRICES RECEIVE

if((n = recvfrom(sock_fd, size, sizeof(size), 0, (struct sockaddr *)&cliaddr, &len)) == -1)
{
    perror("size  not received:");
    exit(1);
}

// RECEIVE MATRIX 1

if((n = recvfrom(sock_fd, matrix_1, sizeof(matrix_1), 0, (struct sockaddr *)&cliaddr, &len)) == -1)
{
    perror("matrix 1 not received:");
    exit(1);
}
```

```
// RECEIVE MATRIX 2

if((n = recvfrom(sock_fd, matrix_2, sizeof(matrix_2), 0, (struct sockaddr *)&cliaddr, &len)) == -1)
{
  perror("matrix 2 not received:");
  exit(1);
}

row_1 = size[0][0];
col_1 = size[0][1];
row_2 = size[1][0];
col_2 = size[1][1];

for (i =0; i < row_1 ; i++)
for (j =0; j <col_2;  j++)
{
  matrix_product[i][j] = 0;
}

for(i =0; i< row_1 ; i++)
for(j=0; j< col_2 ; j++)
for (k=0; k < col_1; k++)
{
  matrix_product[i][j] += matrix_1[i][k]*matrix_2[k][j];
 }

n = sendto(sock_fd, matrix_product, sizeof(matrix_product),0, (struct sockaddr*)&cliaddr,
sizeof(cliaddr));

if( n < 0)
  {
    perror("error in matrix product  sending");
    exit(1);
  }
close(sock_fd);
}
```

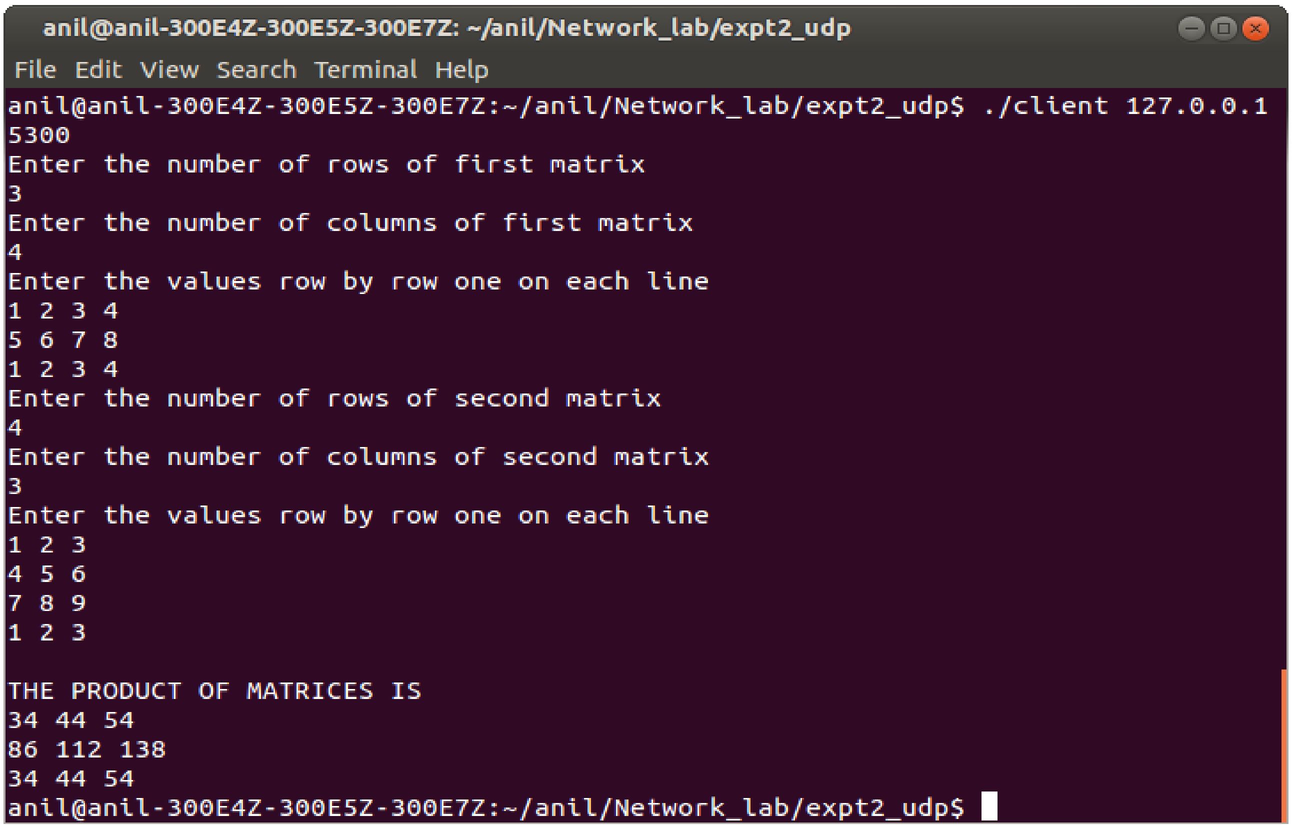
Output

Server


anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt2_udp
File Edit View Search Terminal Help

```
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt2_udp$ ./server 5300
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt2_udp$
```

Client


anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt2_udp
File Edit View Search Terminal Help

```
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt2_udp$ ./client 127.0.0.1
5300
Enter the number of rows of first matrix
3
Enter the number of columns of first matrix
4
Enter the values row by row one on each line
1 2 3 4
5 6 7 8
1 2 3 4
Enter the number of rows of second matrix
4
Enter the number of columns of second matrix
3
Enter the values row by row one on each line
1 2 3
4 5 6
7 8 9
1 2 3

THE PRODUCT OF MATRICES IS
34 44 54
86 112 138
34 44 54
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt2_udp$
```

Experiment 9

Implementation of a multi user chat server using TCP as transport layer protocol

Aim: To implement a chat server so that multiple users can chat simultaneously

Description

To implement chat server the *select* function is used. It is a function called by a process. When a process calls this function , the process goes to sleep and wakes up only when one or more events occur or when a specified amount of time has passed.

int select (int maxfdp1, fd_set * readset, fd_set * writeset, fd_set * exceptset, const struct timeval * timeout);

The header files required are <sys/select.h> and <sys/time.h>
The function returns -1 on error, count of ready descriptors and 0 on timeout. If we want the kernel to wait for as long as one descriptor becomes ready then we specify the timeout argument as a null pointer. *readset, writeset* and *exceptset* specify the descriptors that we want the kernel to test for reading, writing and exception conditions. *select* uses descriptor sets. Each set is an array of integers. Each bit in an integer corresponds to a descriptor. In a 32 bit integer, the first element of the array represents descriptors from 0 to 31. Second element of the array represents descriptors from 32 to 63 and so on. We have four macros for the *fd_set* data type.

void FD_ZERO(fd_set * fdset); // clears all bits in *fd_set*
void FD_SET(int fd, fd_set * fdset); // turns on the bit for *fd* in *fdset*
void FD_CLR(int fd, fd_set * fdset); // turns off the bit for *fd* in *fdset*
int FD_ISSET(int fd, fd_set * fdset); // is the bit for *fd* on in *fdset*

Usage:

fd_set rset;
FD_ZERO(&rset); // all bits off
FD_SET(1, &rset); // turn on bit for *fd* 1
FD_SET(4, &rset); // turn on bit for *fd* 4

Initially you have to initialize the set. If we are not interested in a condition we can set that argument of *select* to NULL, i.e; for *readset*, *writeset* or *exceptset*.

maxfdp1 argument specifies the number of descriptors to be tested. It is equal to value of max descriptor to be tested plus one. This is because descriptor starts with 0. When we call *select* we specify the values of the descriptors that we are interested in and on return the result indicates which descriptors are ready. We turn on all the bits in the descriptor sets that we are interested in. On return of the call, descriptors that are not ready will have the corresponding bit cleared in the descriptor set.

We can use *select* to create a server which can handle clients without forking process for each client.

rset

fd0	fd1	fd2	fd3	
0	0	0	1	

Maxfd + 1 = 4

$\longleftrightarrow$

Client

-1
-1
-1
-1
. . .
-1

Descriptors 0, 1, 2 are respectively for standard input, output and error. Next available descriptor is 3 which is set for listening socket. *Client* is an array that contains the connected socket descriptor for each client. All elements are initialized to -1. The first non zero for descriptor set is that for the listening socket. When the first client establishes a connection with the server, the listening descriptor becomes readable and server calls *accept*. The new connected descriptor will be 4. The arrays are updated.

rset

fd0	fd1	fd2	fd3	fd4	
0	0	0	1	1	

Maxfd + 1 = 5

Client

4
-1
-1
-1
. . .
-1

Now if a second client connects

rset

fd0	fd1	fd2	fd3	fd4	fd5	
0	0	0	1	1	1	

Maxfd + 1 = 6

Client

4
5
-1
-1
. . .
-1

If the first client terminates connection by sending a FIN segment, descriptor 4 becomes readable and *read* returns 0. This socket is closed by the server and data structures are updated.

rset

fd0	fd1	fd2	fd3	fd4	fd5	
0	0	0	1	0	1	

Maxfd + 1 = 6

Client

-1
5
-1
-1
. . .
-1

The descriptor 4 in *rset* is set to zero. When clients arrive the connected socket descriptor is placed in the first available entry, i.e; first entry with value equal to -1.

Program

client

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<stdlib.h>


main(int argc, char * argv[])
```

```
{
  int i,j,n;
  int sock_fd, max_fd, nready, fd[2];

  char buffer[100], line[100];

  struct sockaddr_in servaddr;
  fd_set rset;

  if(argc != 3)
  {
    fprintf(stderr, "Usage: ./client IPaddress_of_server port\n");
    exit(1);
  }
  if((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
        printf("Cannot create socket\n");
        exit(1);
   }

  bzero((char*)&servaddr, sizeof(servaddr));
  bzero(line, sizeof(line));

  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(atoi(argv[2]));
  inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

  if(connect(sock_fd, (struct sockaddr *)&servaddr, sizeof(servaddr))< 0)
  {
    perror("Connection failed:");
    exit(1);
  }

  fd[0] = 0;
  fd[1] = sock_fd;
  for(; ; )
  {
    FD_ZERO(&rset);
    FD_SET(0, &rset);
    FD_SET(sock_fd, &rset);
    bzero(line, sizeof(line));

    max_fd = sock_fd;

    nready = select(max_fd + 1, &rset, NULL, NULL, NULL);
    for ( j = 0; j <2 ; j++)
    {

      if(FD_ISSET(fd[j], &rset))
      {

        n =   read(fd[j], line, sizeof(line));

        if(j == 0)
         {
            n = write(fd[j+1],  line, strlen(line));

         }
        else
         {
            printf("%s \n", line);
         }

         if(--nready == 0)
         break;
      }
    }
  }
}
```

server

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<stdlib.h>

main(int argc, char * argv[])

{
int n, i, maxi, max_fd, k;
int sock_fd, listen_fd, connfd, client_no;
int nready, num_q,  client[100], chat[100], conn[1000];

char line[1000], buffer[1000];

fd_set rset, allset;
```

```
struct sockaddr_in servaddr, cliaddr;

int len = sizeof(cliaddr);
bzero(line, sizeof(line));

client_no = 0;

if(argc != 2)
{
    fprintf(stderr, "Usage: ./server  port\n");
    exit(1);
}

if((listen_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    printf("Cannot create socket\n");
    exit(1);
}

bzero((char*)&servaddr, sizeof(servaddr));

servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(atoi(argv[1]));
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(listen_fd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0)
{
    perror("bind failed:");
    exit(1);
}

listen(listen_fd, num_q);
max_fd = listen_fd;

maxi = -1;

for(i=0; i < 100; i++)
{
   client[i] = -1;
   chat[i] = -1;
}
FD_ZERO(&allset);

FD_SET(listen_fd, &allset);
```

```
for(; ;)
{
    rset = allset;
    nready = select(max_fd + 1, &rset, NULL, NULL, NULL);

    if(FD_ISSET(listen_fd, &rset))
    {
        if((connfd = accept(listen_fd, (struct sockaddr *) &cliaddr,  &len))<0)
         {
             perror("accept failed");
             exit(1);
         }

         chat[++client_no] = connfd;
         conn[connfd] = client_no;

         k = client_no;
         sprintf(buffer, "Client %d has joined chat\n", k);

         while(k > 0 )
           {
              if(chat[k] >0)
              n =   write(chat[k--], buffer, strlen(buffer));
           }
    }

    for( i = 0; i < 100; i++)
     {
         if( client[i] < 0)
          {
             client[i] = connfd;
             break;
           }
     }

      FD_SET(connfd, &allset);
      if(connfd > max_fd)
       max_fd = connfd;

       if( i > maxi)
        maxi = i;

       if(--nready <= 0)
         continue;
}
```

```
for( i =0; i <=maxi; i++)
{
    bzero(line, sizeof(line));

    if((sock_fd = client[i]) <0)
     continue;

    if( FD_ISSET(sock_fd, &rset))
     {
         n= read(sock_fd, line, sizeof(line));
         if( n == 0)
          {
              close(sock_fd);
              FD_CLR(sock_fd, &allset);
              client[i] = -1;

              bzero(buffer, sizeof(buffer));
              sprintf(buffer, "Client %d has left chat\n", conn[sock_fd]);

              k = client_no;
              while(k > 0 )
               {
                   if(chat[k] > 0)
                   n =   write(chat[k--], buffer, strlen(buffer));
                }
             }
          else
           {
               if(chat[line[0] - 48] > 0)
               write(chat[line[0] - 48],  line, strlen(line));
             }

           if(--nready <= 0)
           break;
       }
}
}

}
```

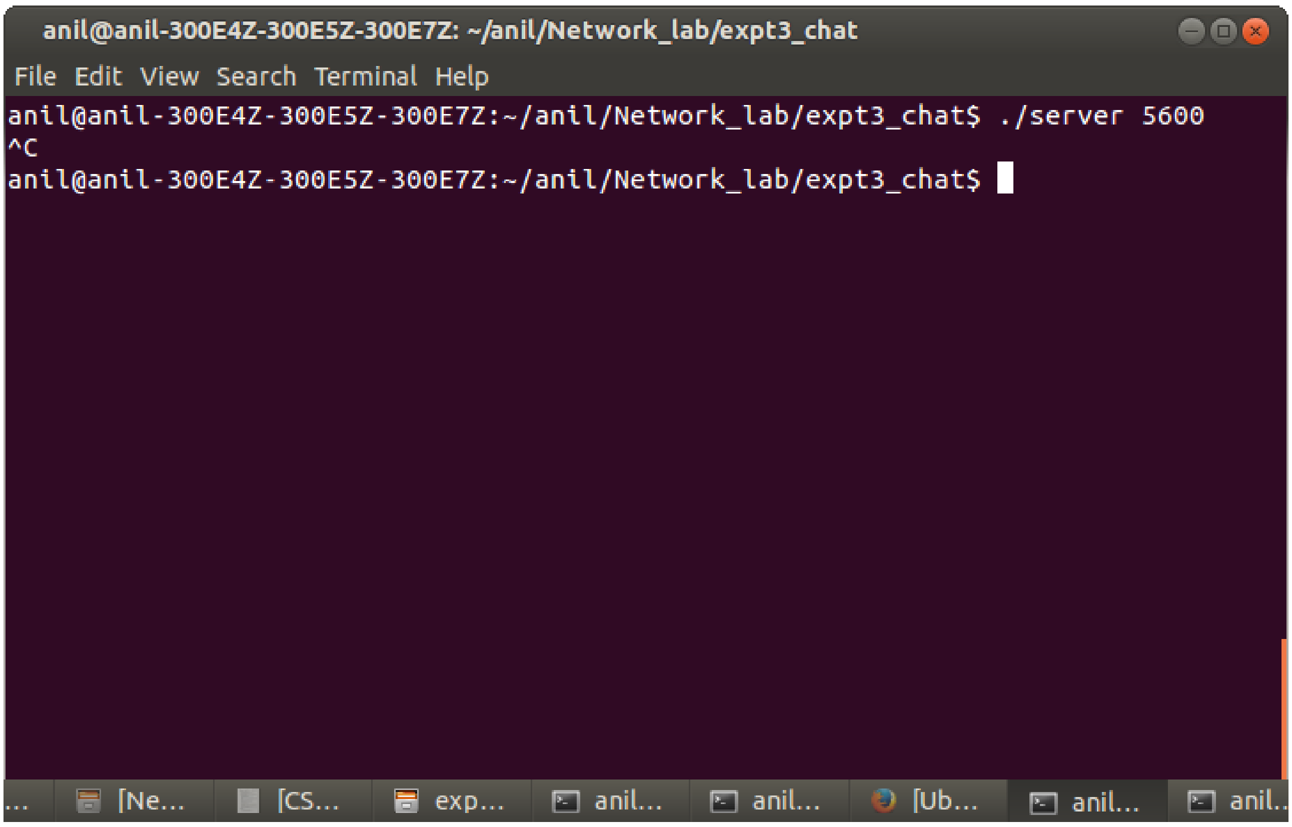
Output

The server is started first. Each client then starts. The clients are numbered consecutively. If client x wants to send message to client y, client x writes

y From x: “contents of message”

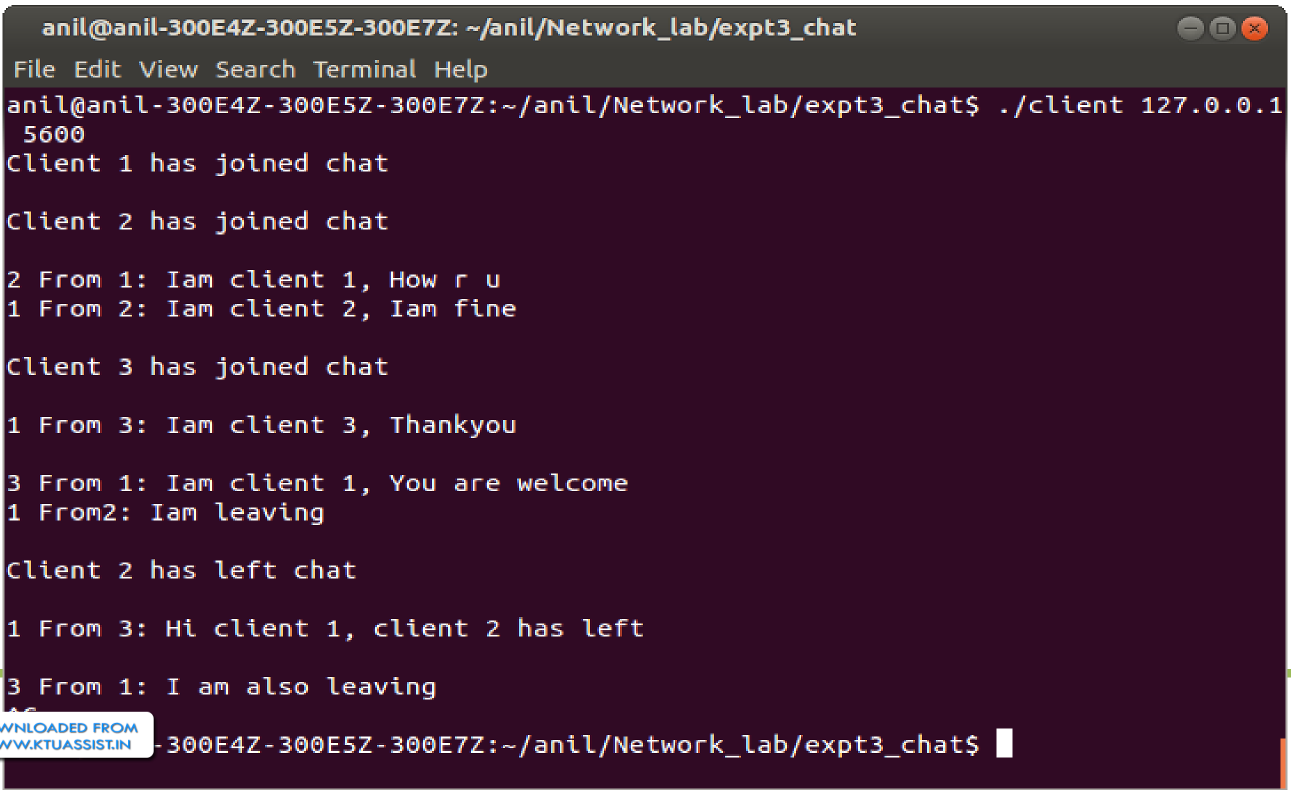
The screenshots show 3 clients chatting with each other through the server running on port 5500.

Server


anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt3_chat
File Edit View Search Terminal Help
[Ne... [CS... exp... anil... anil... [Ub... anil... anil..

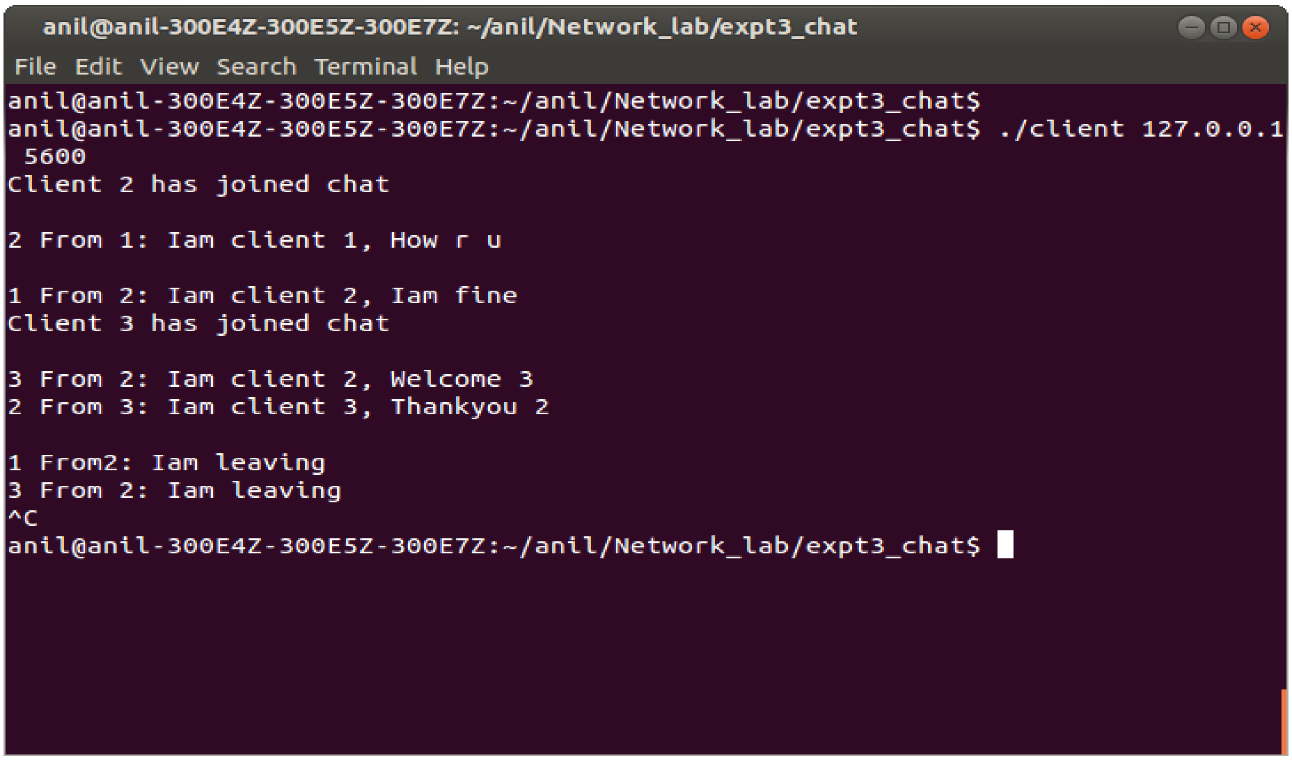
```
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$ ./server 5600
^C
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$
```

Client 1


anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt3_chat
File Edit View Search Terminal Help

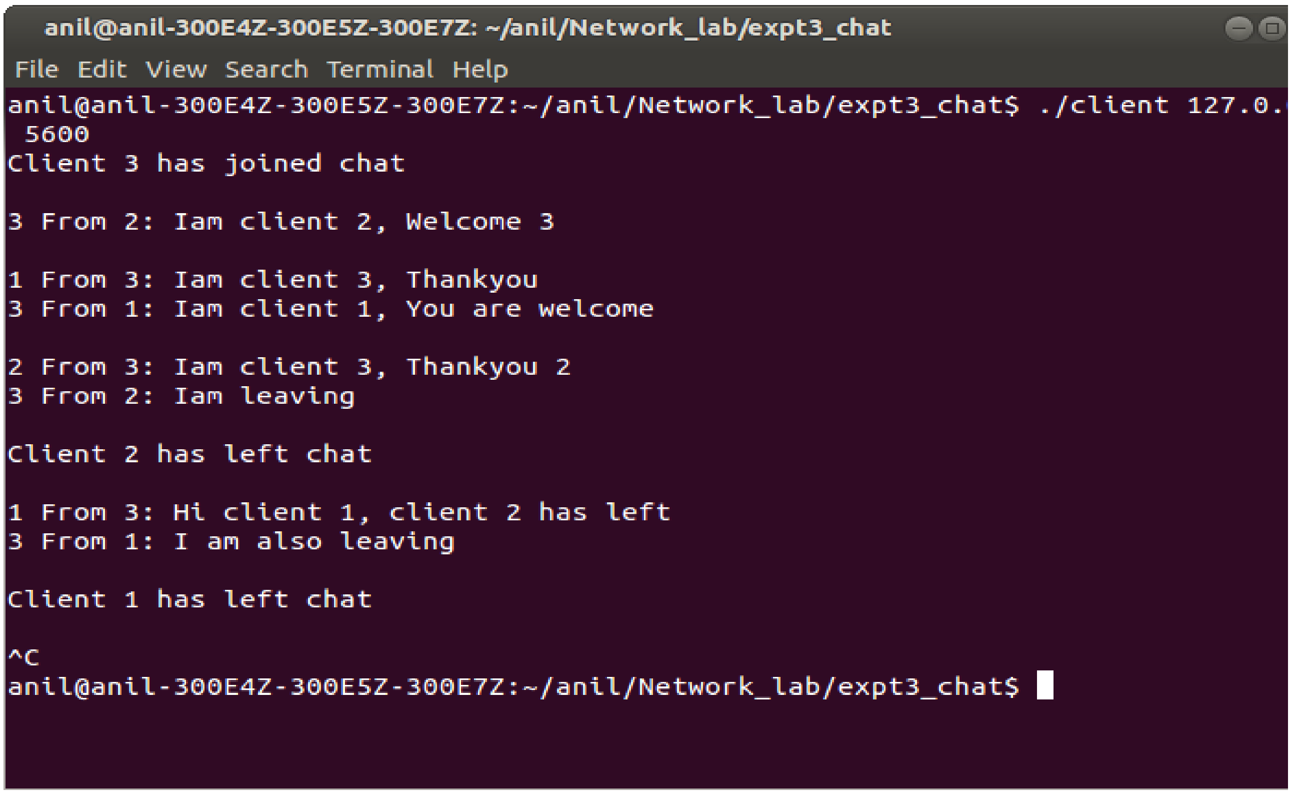
```
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$ ./client 127.0.0.1
 5600
Client 1 has joined chat

Client 2 has joined chat

2 From 1: Iam client 1, How r u
1 From 2: Iam client 2, Iam fine

Client 3 has joined chat

1 From 3: Iam client 3, Thankyou

3 From 1: Iam client 1, You are welcome
1 From2: Iam leaving

Client 2 has left chat

1 From 3: Hi client 1, client 2 has left

3 From 1: I am also leaving
^C
-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$
```

Client 2


anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt3_chat
File Edit View Search Terminal Help

```
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$ ./client 127.0.0.1
 5600
Client 2 has joined chat

2 From 1: Iam client 1, How r u

1 From 2: Iam client 2, Iam fine
Client 3 has joined chat

3 From 2: Iam client 2, Welcome 3
2 From 3: Iam client 3, Thankyou 2

1 From2: Iam leaving
3 From 2: Iam leaving
^C
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$
```

Client 3


anil@anil-300E4Z-300E5Z-300E7Z: ~/anil/Network_lab/expt3_chat
File Edit View Search Terminal Help

```
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$ ./client 127.0.
 5600
Client 3 has joined chat

3 From 2: Iam client 2, Welcome 3

1 From 3: Iam client 3, Thankyou
3 From 1: Iam client 1, You are welcome

2 From 3: Iam client 3, Thankyou 2
3 From 2: Iam leaving

Client 2 has left chat

1 From 3: Hi client 1, client 2 has left
3 From 1: I am also leaving

Client 1 has left chat

^C
anil@anil-300E4Z-300E5Z-300E7Z:~/anil/Network_lab/expt3_chat$
```