**Practical 7 – Computer Networks Lab**

**Name:** Neeraj Belsare

**Roll No.:** 79

**Batch:** A4

**PRN:** 202101040133

**Title:**

Socket Programming

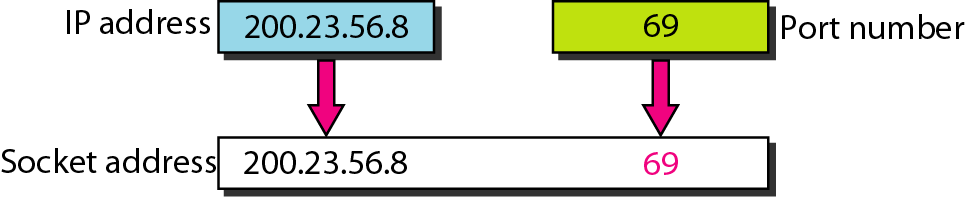
**Aim:**

Write a program to implement simple communication between Client-Server using sockets utility (TCP and UDP) and demonstrate the packets captured traces using Wireshark Packet Analyzer Tool.

**Theory:**

**Socket**

Socket is a software abstraction that represents an endpoint of a two-way communication link between two programs running on a network. Socket provides bi-directional FIFO communication. A socket is created at each end of the communication. Each socket has a specific address called socket address. Socket address is a combination of IP address and port number.



Socket programming is a way of connecting two nodes (client and server) on a network, in order to communicate with each other. The server creates a socket, attaches it to the network port address and then listens/waits for the client to contact it. The client creates a socket and attempts to connect to the server socket. When the connection is established, the transfer of data takes place.

**Types of Sockets:**

1. **Datagram Socket:**

* use User Datagram Protocol. The socket type of datagram socket is SOCK\_DGRAM.
* UDP is connectionless. It does not establish a dedicated connection between the sender and receiver. It does not provide guarantee of delivery. The packets may arrive out of order, be duplicated, or even be lost.
* Datagram sockets are used for applications which require low latency, such as real-time streaming, online gaming, and applications where speed is priority over reliability.

1. **Stream Socket:**

* use Transmission Control Protocol. The socket type of stream socket is SOCK\_STREAM.
* TCP is connection-oriented and reliable. It establishes a connection between the sender and receiver. It guarantees delivery of data. Data is delivered as a continuous stream of bytes. The packets are delivered in order.
* Stream sockets are commonly used for applications that need high reliability such as web browsing, email and file transfer (FTP).

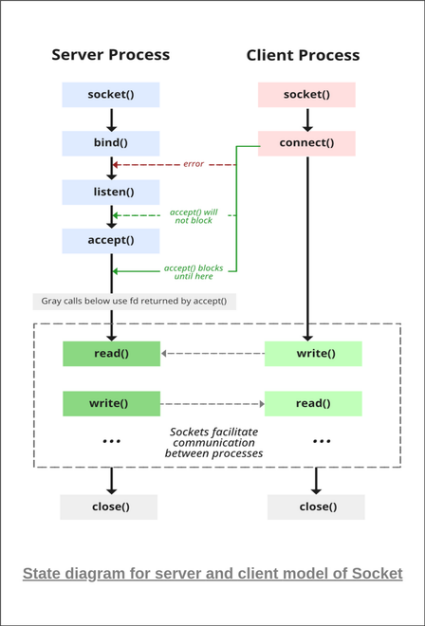
1. **Raw Socket:**

* used in ICMP and IGMP. They provide support in developing new communication protocols or for access to more facilities of an existing protocol. The socket type of Raw Socket is SOCK\_RAW.

1. **Sequenced Packet Socket:**

* The socket type of Sequenced Packet Socket is SOCK\_SEQPACKET.

**State diagram for server and client model of Socket**



**Stages for Server**

**1.** **Socket creation:**

int sockfd = socket (domain, type, protocol)

* **sockfd:** socket descriptor, an integer (like a file-handle)
* **domain:** integer, specifies communication domain. We use AF\_ LOCAL as defined in the POSIX standard for communication between processes on the same host. For communicating between processes on different hosts connected by IPV4, we use AF\_INET and AF\_I NET 6 for processes connected by IPV6.
* **type:** communication type

SOCK\_STREAM: TCP (reliable, connection-oriented)

SOCK\_DGRAM: UDP (unreliable, connectionless)

* **protocol:** Protocol value for Internet Protocol (IP), which is 0. This is the same number which appears on the protocol field in the IP header of a packet.

**2. Setsockopt:**

This helps in manipulating options for the socket referred to by the file descriptor sockfd. This is completely optional, but it helps in the reuse of address and port. Prevents errors such as: “address already in use”.

int setsockopt (int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

**3. Bind:**

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

After the creation of the socket, the bind function binds the socket to the address and port number specified in addr (custom data structure).

**4. Listen:**

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

**5. Accept:**

int new\_socket= accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, the connection is established between client and server, and they are ready to transfer data.

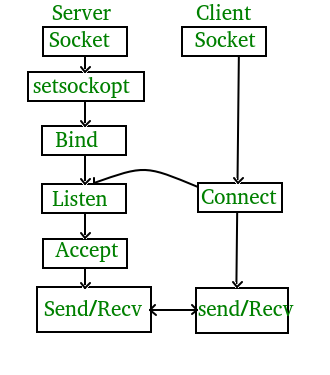
**Stages for Client**

* **Socket connection:** Same as that of the server’s socket creation.
* **Connect:** The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. The server’s address and port are specified in addr.
* int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

**Communication between Client-Server using TCP**

If we are creating a connection between client and server using TCP then it has a few functionalities, TCP is suited for applications that require high reliability, and transmission time is relatively less critical. It is used by other protocols like HTTP, HTTPS, FTP, SMTP, and Telnet. TCP rearranges data packets in the order specified. There is a guarantee that the data transferred remains intact and arrives in the same order in which it was sent. TCP does Flow Control and requires three packets to set up a socket connection before any user data can be sent. TCP handles reliability and congestion control. It also does error checking and error recovery. Erroneous packets are retransmitted from the source to the destination.

The entire process can be broken down into the following steps:



The entire process can be broken down into the following steps:

**TCP Server:**

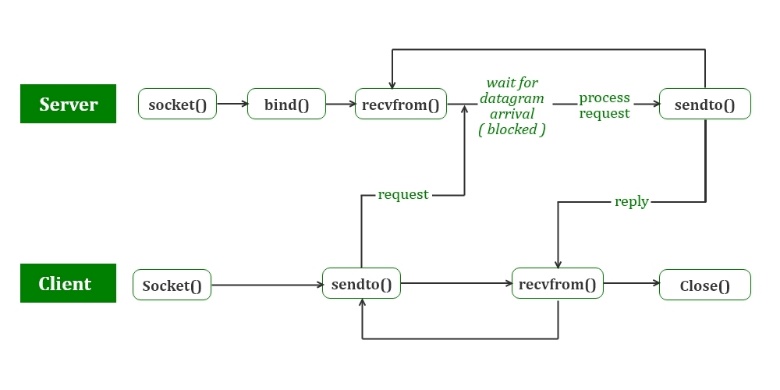
1. Create TCP socket.
2. Bind the socket to server address.
3. Put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
4. At this point, connection is established between client and server, and they are ready to transfer data.
5. Go back to Step 3.

**TCP Client:**

1. Create TCP socket.
2. connect newly created client socket to server.

**Communication between Client-Server using UDP**

In UDP, the client does not form a connection with the server like in TCP and instead sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of the sender which the server uses to send data to the correct client.



The entire process can be broken down into the following steps:

**UDP Server:**

1. Create a UDP socket.
2. Bind the socket to the server address.
3. Wait until the datagram packet arrives from the client.
4. Process the datagram packet and send a reply to the client.
5. Go back to Step 3.

**UDP Client:**

1. Create a UDP socket.
2. Send a message to the server.
3. Wait until a response from the server is received.
4. Process the reply and go back to step 2, if necessary.
5. Close socket descriptor and exit.

**Procedure/Code:**

1. **Communication between Client-Server using TCP**

**server.c**

#include <stdio.h>

#include <netdb.h>

#include <netinet/in.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <unistd.h> // read(), write(), close()

#define MAX 80

#define PORT 8080

#define SA struct sockaddr

// Function designed for chat between client and server.

void func(int connfd)

{

char buff[MAX];

int n;

// infinite loop for chat

for (;;) {

bzero(buff, MAX);

// read the message from client and copy it in buffer

read(connfd, buff, sizeof(buff));

// print buffer which contains the client contents

printf("From client: %s\t To client : ", buff);

bzero(buff, MAX);

n = 0;

// copy server message in the buffer

while ((buff[n++] = getchar()) != '\n')

;

// and send that buffer to client

write(connfd, buff, sizeof(buff));

// if msg contains "Exit" then server exit and chat ended.

if (strncmp("exit", buff, 4) == 0) {

printf("Server Exit...\n");

break;

}

}

}

// Driver function

int main()

{

int sockfd, connfd, len;

struct sockaddr\_in servaddr, cli;

// socket create and verification

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1) {

printf("socket creation failed...\n");

exit(0);

}

else

printf("Socket successfully created..\n");

bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(PORT);

// Binding newly created socket to given IP and verification

if ((bind(sockfd, (SA\*)&servaddr, sizeof(servaddr))) != 0) {

printf("socket bind failed...\n");

exit(0);

}

else

printf("Socket successfully binded..\n");

// Now server is ready to listen and verification

if ((listen(sockfd, 5)) != 0) {

printf("Listen failed...\n");

exit(0);

}

else

printf("Server listening..\n");

len = sizeof(cli);

// Accept the data packet from client and verification

connfd = accept(sockfd, (SA\*)&cli, &len);

if (connfd < 0) {

printf("server accept failed...\n");

exit(0);

}

else

printf("server accept the client...\n");

// Function for chatting between client and server

func(connfd);

// After chatting close the socket

close(sockfd);

}

**client.c**

#include <arpa/inet.h> // inet\_addr()

#include <netdb.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <strings.h> // bzero()

#include <sys/socket.h>

#include <unistd.h> // read(), write(), close()

#define MAX 80

#define PORT 8080

#define SA struct sockaddr

void func(int sockfd)

{

char buff[MAX];

int n;

for (;;) {

bzero(buff, sizeof(buff));

printf("Enter the string : ");

n = 0;

while ((buff[n++] = getchar()) != '\n')

;

write(sockfd, buff, sizeof(buff));

bzero(buff, sizeof(buff));

read(sockfd, buff, sizeof(buff));

printf("From Server : %s", buff);

if ((strncmp(buff, "exit", 4)) == 0) {

printf("Client Exit...\n");

break;

}

}

}

int main()

{

int sockfd, connfd;

struct sockaddr\_in servaddr, cli;

// socket create and verification

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1) {

printf("socket creation failed...\n");

exit(0);

}

else

printf("Socket successfully created..\n");

bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

servaddr.sin\_port = htons(PORT);

// connect the client socket to server socket

if (connect(sockfd, (SA\*)&servaddr, sizeof(servaddr))

!= 0) {

printf("connection with the server failed...\n");

exit(0);

}

else

printf("connected to the server..\n");

// function for chat

func(sockfd);

// close the socket

close(sockfd);

}

**b) Communication between Client-Server using UDP**

**server\_udp.cpp**

// Server side implementation of UDP client-server model

#include <bits/stdc++.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#define PORT 8080

#define MAXLINE 1024

// Driver code

int main() {

int sockfd;

char buffer[MAXLINE];

const char \*hello = "Hello from server";

struct sockaddr\_in servaddr, cliaddr;

// Creating socket file descriptor

if ( (sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0 ) {

perror("socket creation failed");

exit(EXIT\_FAILURE);

}

memset(&servaddr, 0, sizeof(servaddr));

memset(&cliaddr, 0, sizeof(cliaddr));

// Filling server information

servaddr.sin\_family = AF\_INET; // IPv4

servaddr.sin\_addr.s\_addr = INADDR\_ANY;

servaddr.sin\_port = htons(PORT);

// Bind the socket with the server address

if ( bind(sockfd, (const struct sockaddr \*)&servaddr,

sizeof(servaddr)) < 0 )

{

perror("bind failed");

exit(EXIT\_FAILURE);

}

socklen\_t len;

int n;

len = sizeof(cliaddr); //len is value/result

n = recvfrom(sockfd, (char \*)buffer, MAXLINE,

MSG\_WAITALL, ( struct sockaddr \*) &cliaddr,

&len);

buffer[n] = '\0';

printf("Client : %s\n", buffer);

sendto(sockfd, (const char \*)hello, strlen(hello),

MSG\_CONFIRM, (const struct sockaddr \*) &cliaddr,

len);

std::cout<<"Hello message sent."<<std::endl;

return 0;

}

**client\_udp.cpp**

// Client side implementation of UDP client-server model

#include <bits/stdc++.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#define PORT 8080

#define MAXLINE 1024

// Driver code

int main() {

int sockfd;

char buffer[MAXLINE];

const char \*hello = "Hello from client";

struct sockaddr\_in servaddr;

// Creating socket file descriptor

if ( (sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0 ) {

perror("socket creation failed");

exit(EXIT\_FAILURE);

}

memset(&servaddr, 0, sizeof(servaddr));

// Filling server information

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(PORT);

servaddr.sin\_addr.s\_addr = INADDR\_ANY;

int n;

socklen\_t len;

sendto(sockfd, (const char \*)hello, strlen(hello),

MSG\_CONFIRM, (const struct sockaddr \*) &servaddr,

sizeof(servaddr));

std::cout<<"Hello message sent."<<std::endl;

n = recvfrom(sockfd, (char \*)buffer, MAXLINE,

MSG\_WAITALL, (struct sockaddr \*) &servaddr,

&len);

buffer[n] = '\0';

std::cout<<"Server :"<<buffer<<std::endl;

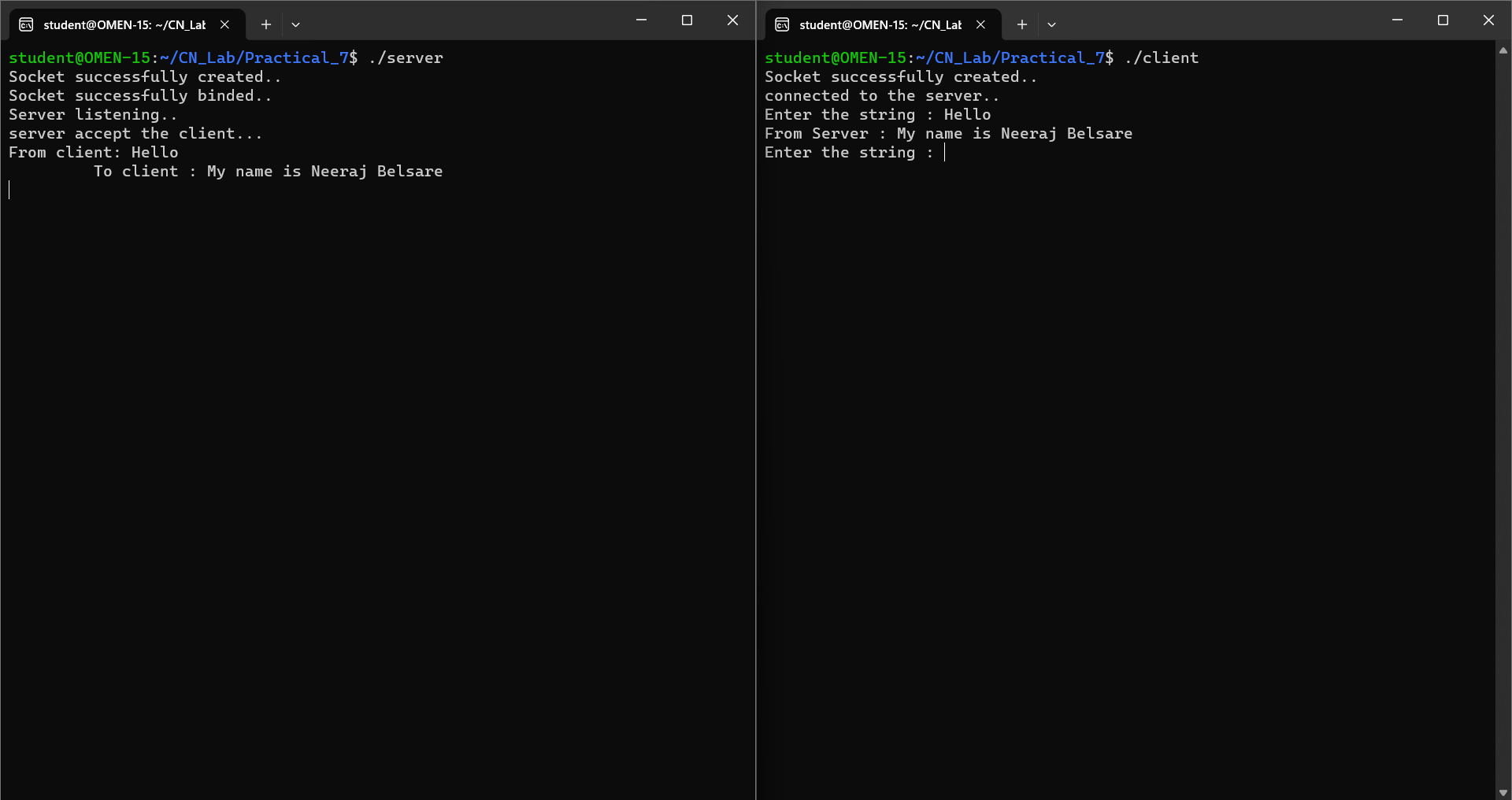
close(sockfd);

return 0;

}

**Output:**

**a) Communication between Client-Server using TCP**



**b) Communication between Client-Server using UDP**

