DATE :- 06-08-24

1.) TCP Server-Client Communication:

Problem Statement: Write a TCP server and client program in C where the server listens for incoming connections and echoes back any message it receives from the client. The client should be able to send a message to the server and display the echoed message.

Requirements:

The server should run indefinitely, waiting for client connections.

The client should take a message as input from the user, send it to the server, and display the response.

Implement proper error handling and cleanup (e.g., closing sockets).

Server code :-

```
rps@rps-virtual-machine:~$ vim serverss.cpp
rps@rps-virtual-machine:~$ g++ -o serverss serverss.cpp
rps@rps-virtual-machine:~$ ./serverss
Server waiting for connections...
Connection accepted from 127.0.0.1:56502
Received: hii
Echoed: hii
Server waiting for connections...
^C
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE] = {0};
    int opt = 1;

    // 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");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Binding the socket to the port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
```

```c
        perror("Bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Listening for incoming connections
    if (listen(server_fd, 3) < 0) {
        perror("Listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    while (1) {
        printf("Server waiting for connections...\n");

        // Accepting a new connection
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0) {
            perror("Accept");
            close(server_fd);
            exit(EXIT_FAILURE);
        }

        printf("Connection accepted from %s:%d\n", inet_ntoa(address.sin_addr), ntohs(address.sin_port));

        // Reading message from the client
        ssize_t valread = read(new_socket, buffer, BUFFER_SIZE);
        if (valread > 0) {
            buffer[valread] = '\0'; // Null-terminate the string
            printf("Received: %s\n", buffer);

            // Echoing the message back to the client
            send(new_socket, buffer, strlen(buffer), 0);
            printf("Echoed: %s\n", buffer);
        }
```


```c
        if (valread > 0) {
            buffer[valread] = '\0'; // Null-terminate the string
            printf("Received: %s\n", buffer);

            // Echoing the message back to the client
            send(new_socket, buffer, strlen(buffer), 0);
            printf("Echoed: %s\n", buffer);
        }
```


```c
        // Closing the socket for the current client
        close(new_socket);
    }

    // Closing the server socket (unreachable in this code as the server runs indefinitely)
    close(server_fd);

    return 0;
```

Client code :-

```
rps@rps-virtual-machine:~$ vim clientss.cpp
rps@rps-virtual-machine:~$ vim clnt.cpp
rps@rps-virtual-machine:~$ g++ -o clnt clnt.cpp
rps@rps-virtual-machine:~$ ./clnt
Enter message: hii
Message sent: hii
Echoed message: hii
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char message[BUFFER_SIZE];
    char buffer[BUFFER_SIZE] = {0};

    // Creating socket file descriptor
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    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;
    }

    // Connecting to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
```

```c
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;
}

// Connecting to the server
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
    printf("\nConnection Failed \n");
    return -1;
}

printf("Enter message: ");
fgets(message, BUFFER_SIZE, stdin);
message[strcspn(message, "\n")] = 0; // Remove newline character

// Sending message to the server
send(sock, message, strlen(message), 0);
printf("Message sent: %s\n", message);

// Reading server's response
ssize_t valread = read(sock, buffer, BUFFER_SIZE);
if (valread > 0) {
    buffer[valread] = '\0'; // Null-terminate the string
    printf("Echoed message: %s\n", buffer);
}

// Closing the socket
close(sock);

return 0;
```

2.) UDP Server-Client Communication:

Problem Statement: Write a UDP server and client program in C where the server listens on a specific port and responds with "Hello, Client!" whenever it receives a message. The client should send a message to the server and print the response.

Requirements:

The server should run indefinitely, waiting for incoming messages.

The client should send a predefined message (e.g., "Hello, Server!") and display the server's response.

Implement proper error handling.

Server code :-

```
rps@rps-virtual-machine:~$ vim udp.cpp
rps@rps-virtual-machine:~$ g++ -o udp udp.cpp
rps@rps-virtual-machine:~$ ./udp
UDP server is running on port 8080
Received message from client: Hello from server!
Response sent to client.
```

```c
#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    char buffer[BUFFER_SIZE];
    struct sockaddr_in serverAddr, clientAddr;
    socklen_t clientAddrLen = sizeof(clientAddr);

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Setting up the server address structure
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(PORT);

    // Bind the socket with the server address
    if (bind(sockfd, (const struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0) {
        perror("Bind failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }
```

```
        perror("Bind failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    std::cout << "UDP server is running on port " << PORT << std::endl;

    while (true) {
        memset(buffer, 0, BUFFER_SIZE);

        // Receive message from client
        ssize_t len = recvfrom(sockfd, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&clientAddr, &clientAddrLen)
        if (len < 0) {
            perror("recvfrom failed");
            continue;
        }

        std::cout << "Received message from client: " << buffer << std::endl;

        // Send response to the client
        const char *response = "Hello, Client!";
        sendto(sockfd, response, strlen(response), 0, (struct sockaddr *)&clientAddr, clientAddrLen);
        std::cout << "Response sent to client." << std::endl;
    }

    // Close the socket (unreachable in this code as the server runs indefinitely)
    close(sockfd);
    return 0;
}
```

Client code :-

```
rps@rps-virtual-machine:~$ vim udpclient.cpp
rps@rps-virtual-machine:~$ ./udpclient
bash: ./udpclient: No such file or directory
rps@rps-virtual-machine:~$ g++ -o udpclient udpclient.cpp
rps@rps-virtual-machine:~$ ./udpclient
Message sent: Hello, Server!
Received from server: Hello, Client!
```

```
#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    char buffer[BUFFER_SIZE];
    struct sockaddr_in serverAddr;
    socklen_t addrLen = sizeof(serverAddr);

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Setting up the server address structure
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    const char *message = "Hello, Server!";

    // Sending message to the server
    sendto(sockfd, message, strlen(message), 0, (const struct sockaddr *)&serverAddr, addrLen);
    std::cout << "Message sent: " << message << std::endl;

    // Receiving response from the server
```

```
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(PORT);
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

const char *message = "Hello, Server!";

// Sending message to the server
sendto(sockfd, message, strlen(message), 0, (const struct sockaddr *)&serverAddr, addrLen);
std::cout << "Message sent: " << message << std::endl;

// Receiving response from the server
ssize_t len = recvfrom(sockfd, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&serverAddr, &addrL
if (len < 0) {
    perror("recvfrom failed");
} else {
    buffer[len] = '\0'; // Null-terminate the string
    std::cout << "Received from server: " << buffer << std::endl;
}

// Close the socket
close(sockfd);
return 0;
```

3.) File Transfer using TCP:

Problem Statement: Write a TCP server and client program in C to transfer a file from the client to the server. The server should save the received file with the same name, and the client should specify the file to be sent.

Requirements:

The server should run indefinitely, waiting for file transfer requests.

The client should prompt the user for a file path, read the file, and send its contents to the server.

Implement proper error handling and file operations.

SERVER CODE :-

```
rps@rps-virtual-machine:~$ ./tcpserver
Server is waiting for file transfer requests...
Connected to client
Receiving file: tcpclient.cpp
File received and saved successfully
^C
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024
#define FILE_NAME_SIZE 256

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE];
    char file_name[FILE_NAME_SIZE];

    // 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
    int opt = 1;
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
        perror("setsockopt");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
```

```c
// Bind the socket to the port
if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
    perror("Bind failed");
    close(server_fd);
    exit(EXIT_FAILURE);
}

// Listen for incoming connections
if (listen(server_fd, 3) < 0) {
    perror("Listen failed");
    close(server_fd);
    exit(EXIT_FAILURE);
}

printf("Server is waiting for file transfer requests...\n");

while (1) {
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0) {
        perror("Accept failed");
        continue;
    }

    printf("Connected to client\n");

    // Receiving the file name
    ssize_t valread = read(new_socket, file_name, FILE_NAME_SIZE);
    if (valread <= 0) {
        perror("Failed to receive file name");
        close(new_socket);
        continue;
    }
```

```
        perror("Failed to receive file name");
        close(new_socket);
        continue;
    }
    file_name[valread] = '\0';
    printf("Receiving file: %s\n", file_name);

    // Open the file to write the received contents
    FILE *file = fopen(file_name, "wb");
    if (file == NULL) {
        perror("File open failed");
        close(new_socket);
        continue;
    }

    // Receive the file contents
    while ((valread = read(new_socket, buffer, BUFFER_SIZE)) > 0) {
        fwrite(buffer, sizeof(char), valread, file);
    }

    if (valread < 0) {
        perror("File receive failed");
    }

    printf("File received and saved successfully\n");

    // Close the file and socket
    fclose(file);
    close(new_socket);
}

// Close the server socket (unreachable in this code as the server runs indefinitely)
close(server_fd);
return 0;
```

CLIENT CODE :-

```
rps@rps-virtual-machine:~$ ./tcpclient
Enter the path of the file to be sent: tcpclient.cpp
File sent successfully
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
    char file_path[BUFFER_SIZE];

    // Creating socket file descriptor
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    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;
    }

    // Connecting to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
```

rps@rps-virtual-machine: ~ rps@rps-virtual-machine: ~ rps@rps-virtual-machine: ~

```
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
    printf("\nConnection Failed \n");
    return -1;
}

// Prompt the user for the file path
printf("Enter the path of the file to be sent: ");
scanf("%s", file_path);

// Open the file
FILE *file = fopen(file_path, "rb");
if (file == NULL) {
    perror("File open error");
    close(sock);
    return -1;
}

// Send the file name to the server
char *file_name = strrchr(file_path, '/');
if (file_name == NULL) {
    file_name = file_path;
} else {
    file_name++;
}
send(sock, file_name, strlen(file_name), 0);

// Read and send the file contents
size_t bytes_read;
while ((bytes_read = fread(buffer, sizeof(char), BUFFER_SIZE, file)) > 0) {
    send(sock, buffer, bytes_read, 0);
}

printf("File sent successfully\n");

// Close the file and socket
```

```
    size_t bytes_read;
    while ((bytes_read = fread(buffer, sizeof(char), BUFFER_SIZE, file)) > 0) {
        send(sock, buffer, bytes_read, 0);
    }

    printf("File sent successfully\n");

    // Close the file and socket
    fclose(file);
    close(sock);

    return 0;
}
```

4) Broadcast Messaging using UDP:

Problem Statement: Write a UDP server and client program in C to implement a simple broadcast messaging system. The server should broadcast a message to all clients in the network, and each client should display any broadcast messages it receives.

Requirements:

The server should send a broadcast message to a specific port.

Each client should listen on the same port and display any messages it receives.

Implement proper error handling and use UDP broadcast mechanisms.

Server code :-

```
rps@rps-virtual-machine:~$ ./broadserver
Enter the message to broadcast: ameesha
Broadcast message sent: ameesha
rps@rps-virtual-machine:~$
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BROADCAST_IP "255.255.255.255"
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    struct sockaddr_in broadcastAddr;
    char message[BUFFER_SIZE];
    int broadcastPermission = 1;

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Set socket options to allow broadcast
    if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcastPermission, sizeof(broadcastPermission)) < 0) {
        perror("setsockopt(SO_BROADCAST) failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Setup the broadcast address structure
    memset(&broadcastAddr, 0, sizeof(broadcastAddr));
    broadcastAddr.sin_family = AF_INET;
    broadcastAddr.sin_addr.s_addr = inet_addr(BROADCAST_IP);
    broadcastAddr.sin_port = htons(PORT);
```

```c
    // Setup the broadcast address structure
    memset(&broadcastAddr, 0, sizeof(broadcastAddr));
    broadcastAddr.sin_family = AF_INET;
    broadcastAddr.sin_addr.s_addr = inet_addr(BROADCAST_IP);
    broadcastAddr.sin_port = htons(PORT);

    printf("Enter the message to broadcast: ");
    fgets(message, BUFFER_SIZE, stdin);
    message[strcspn(message, "\n")] = '\0'; // Remove newline character

    // Send the broadcast message
    if (sendto(sockfd, message, strlen(message), 0, (struct sockaddr *)&broadcastAddr, sizeof(broadcastAddr)) < 0) {
        perror("Broadcast failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("Broadcast message sent: %s\n", message);

    // Close the socket
    close(sockfd);
    return 0;
}
```

CLIENT CODE :-

```
rps@rps-virtual-machine:~$ ./broadclient
Listening for broadcast messages on port 8080...
hii
ameesha
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    struct sockaddr_in serverAddr;
    char buffer[BUFFER_SIZE];
    socklen_t addrLen = sizeof(serverAddr);

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Setup the server address structure
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(PORT);

    // Bind the socket to the port
    if (bind(sockfd, (const struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0) {
        perror("Bind failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("Listening for broadcast messages on port %d...\n", PORT);
```

```c
// Infinite loop to listen for messages
while (1) {
    memset(buffer, 0, BUFFER_SIZE);
    ssize_t len = recvfrom(sockfd, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&serverAddr, &addrLen);
    if (len < 0) {
        perror("recvfrom failed");
    } else {
        buffer[len] = '\0'; // Null-terminate the string
        printf("Received message: %s\n", buffer);
    }
}

// Close the socket (unreachable in this code as the client listens indefinitely)
close(sockfd);
return 0;
```