

EXAM 2-4:

1. Objective:

Create a C++ application that combines signal handling and socket programming to manage network communication while gracefully handling interruptions (e.g., SIGINT for program termination). The application should be capable of sending and receiving messages over a network while responding appropriately to system signals.

Requirements:

Socket Programming:

Implement a TCP server that listens for incoming connections on a specified port.

Implement a TCP client that connects to the server and exchanges messages.

Signal Handling:

Implement signal handlers for SIGINT (Ctrl+C) and SIGTERM to gracefully shut down the server and client.

Ensure that the program can handle interruptions without crashing or leaving resources unfreed.

Data Exchange:

The client should be able to send a message to the server.

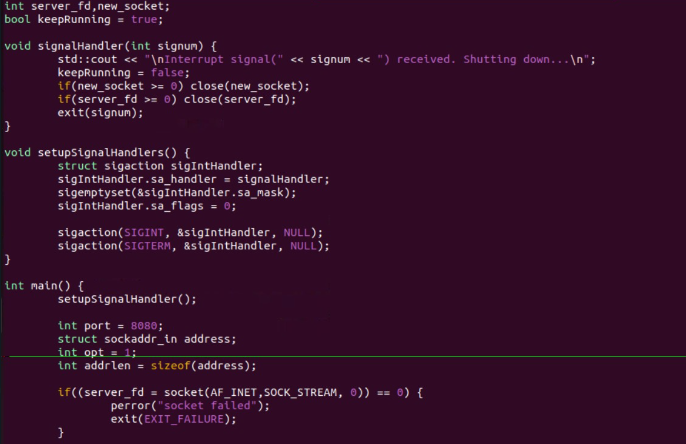
The server should echo the received message back to the client.

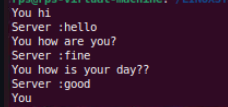
Graceful Shutdown:

When the server receives a SIGINT or SIGTERM signal, it should close all active connections and free resources before terminating.

When the client receives a SIGINT or SIGTERM signal, it should inform the server before terminating.







Tcp echo:

1.

CP Echo Server:

Implement a TCP server that:

Binds to port 9090.

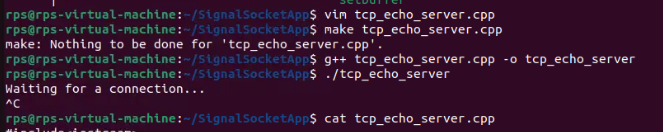
Listens for incoming connections.

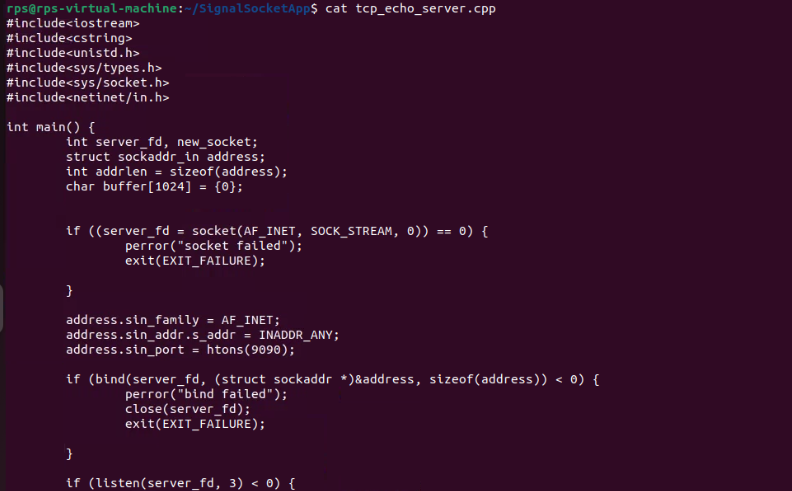
Accepts a connection from a client.

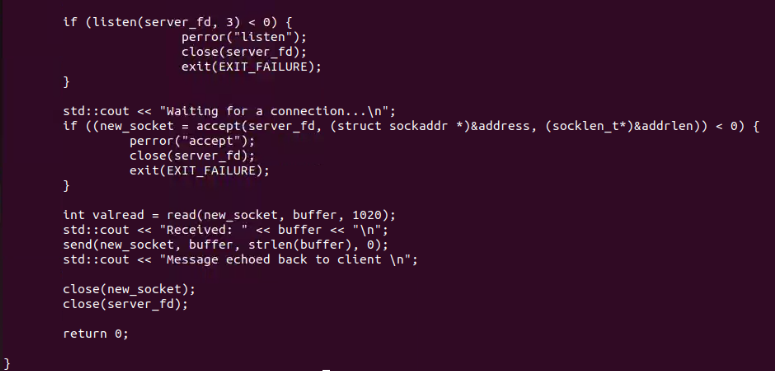
Receives a message from the client and echoes the same message back to the client.

Closes the connection and terminates.

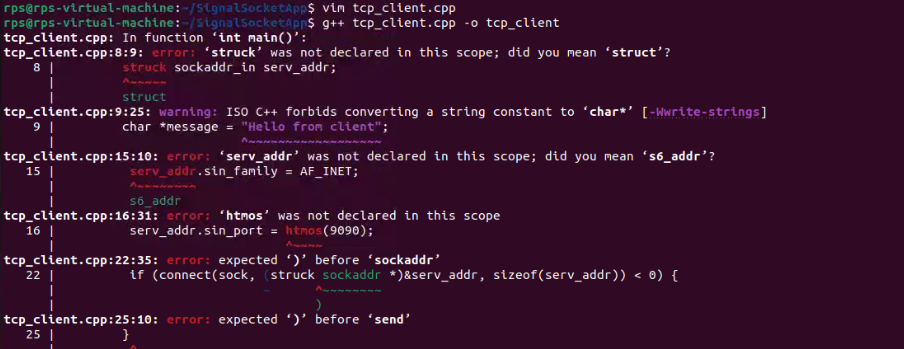
UDP Client-Server Communication:

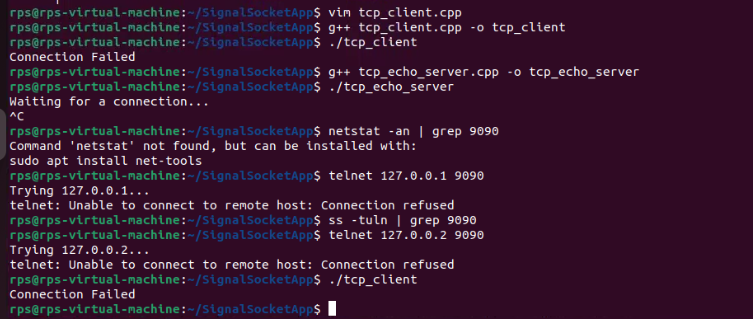


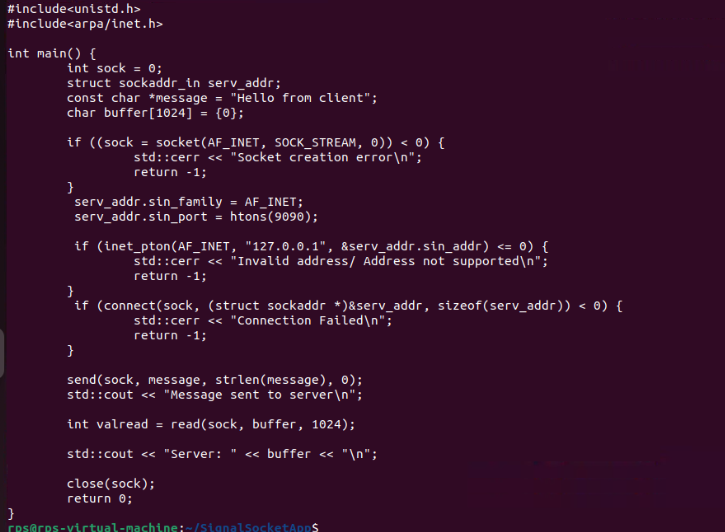




Cliemt\:

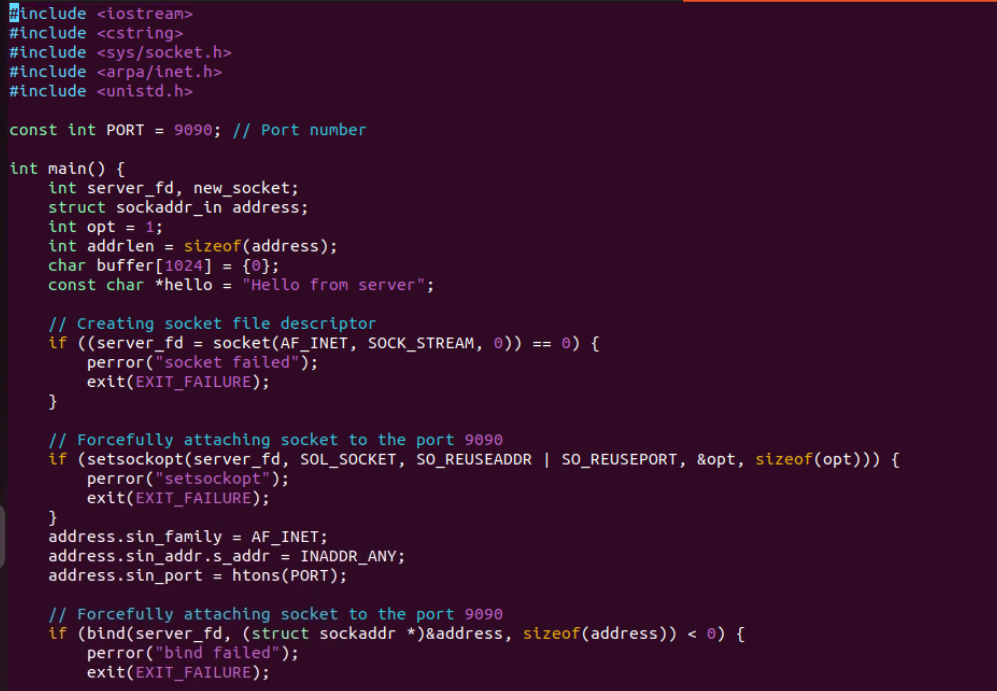


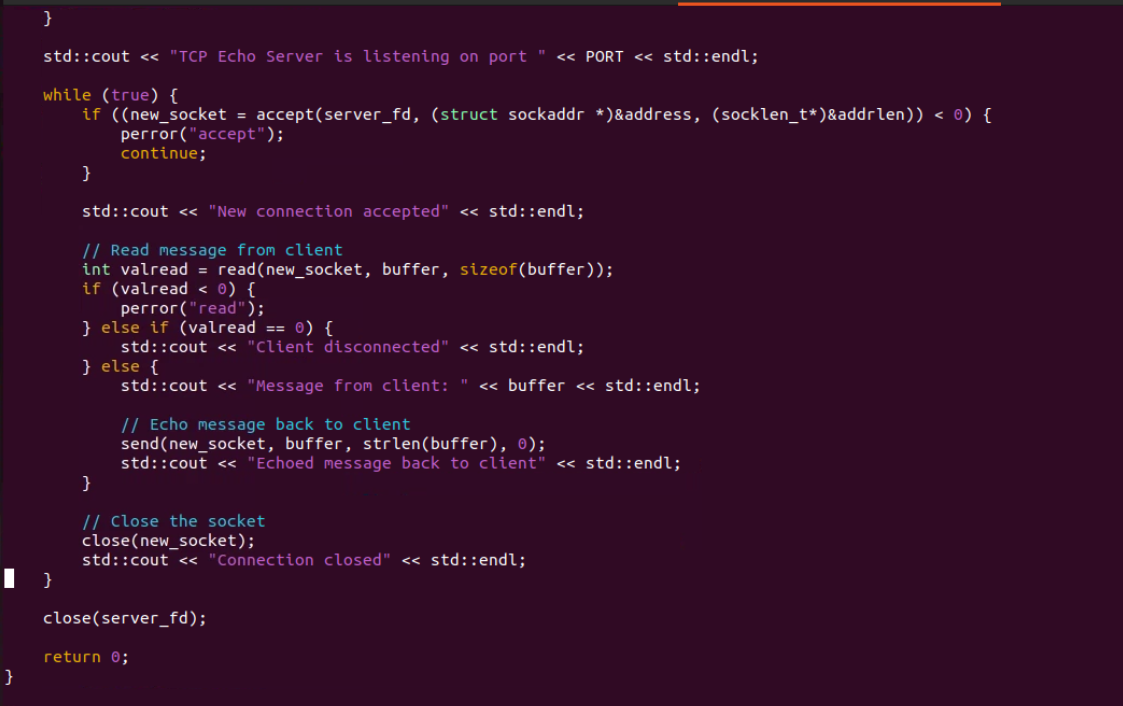




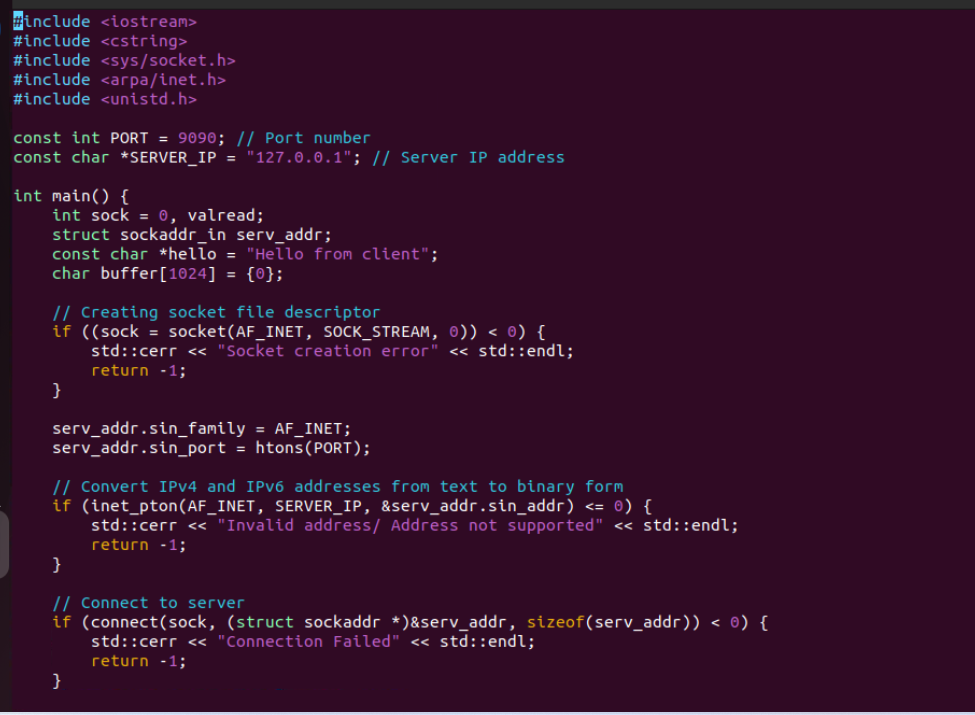
**2.** **CP Echo Server:Implement a TCP server that:Binds to port 9090.Listens for incoming connections.Accepts a connection from a client.Receives a message from the client and echoes the same message back to the client.Closes the connection and terminates.UDP Client-Server Communication:**

**Server:**

****

****

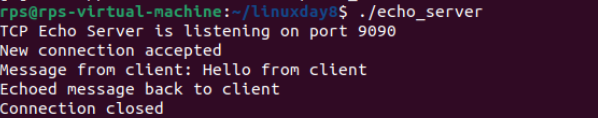
**Client:**

****

****

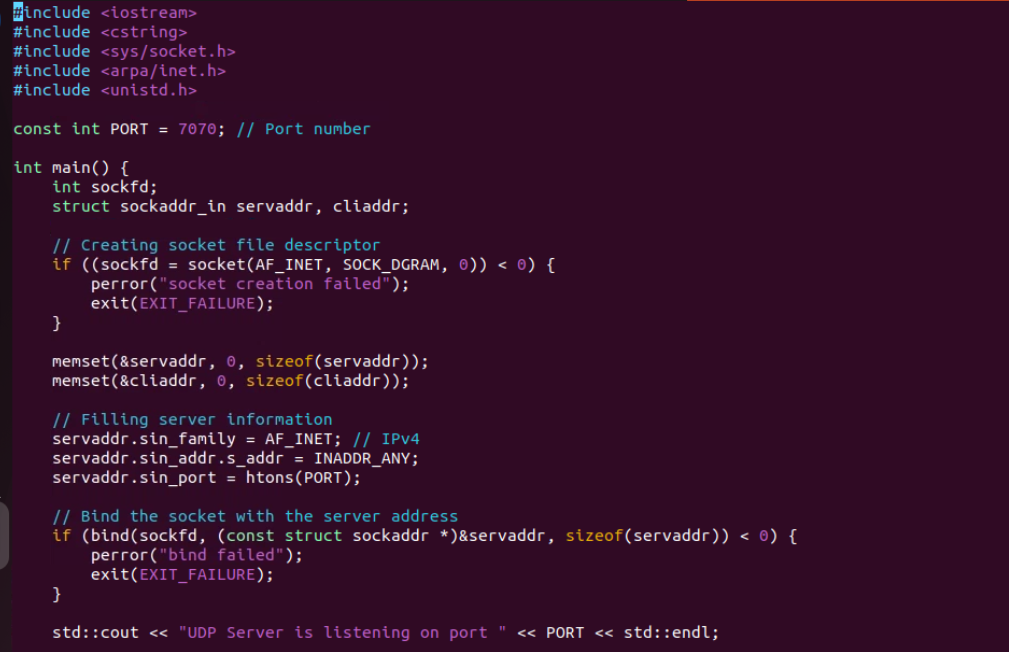
**Output:**

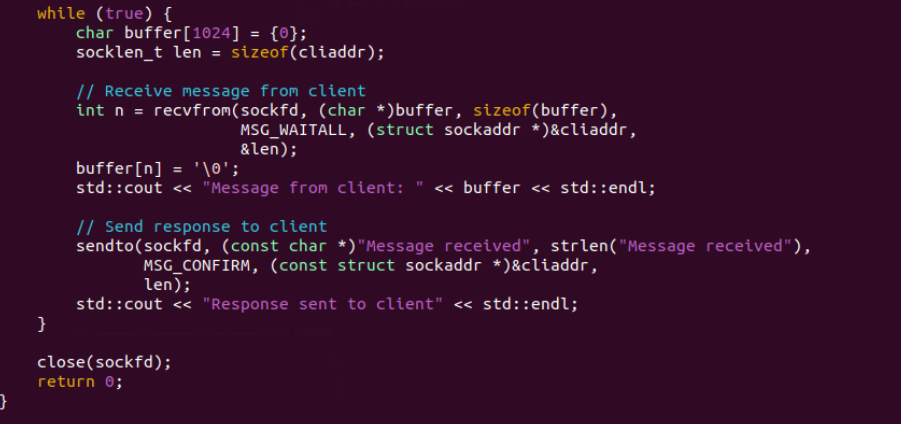
****

****

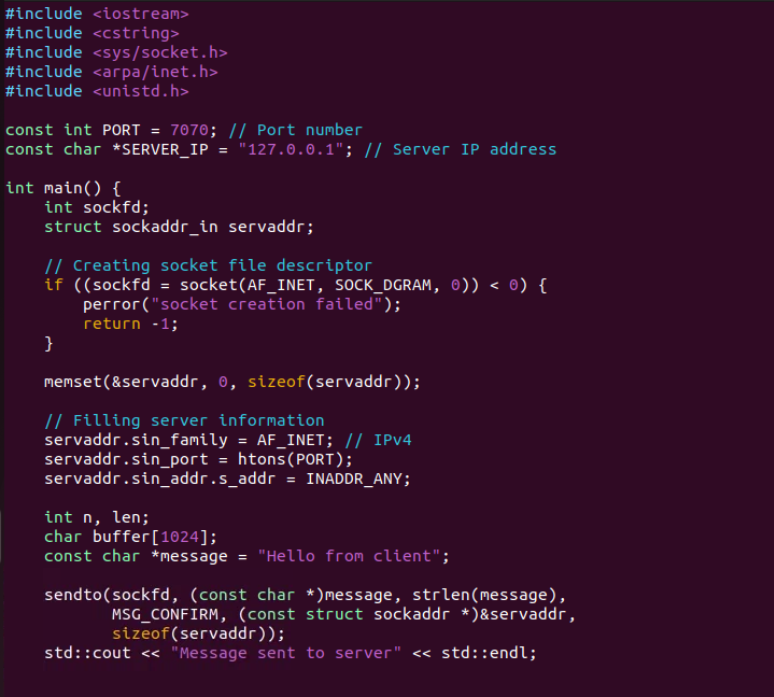
**3.** **Create a UDP server that:Binds to port 7070.Receives a message from a client.Sends a response message back to the client.Closes the socket and terminates.Create a UDP client that:Sends a message to the server on port 7070.Receives and prints the response message from the server.Closes the socket and terminates.Multi-Client TCP Server:**

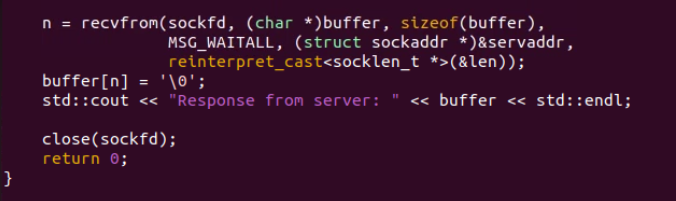
**Server:**

****

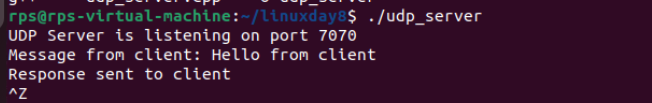
****

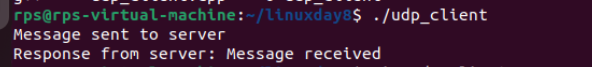
**Client:**

****

****

**Output:**

****

****

**4.** **Implement a TCP server that:**

**Binds to port 6060.**

**Listens for incoming connections.**

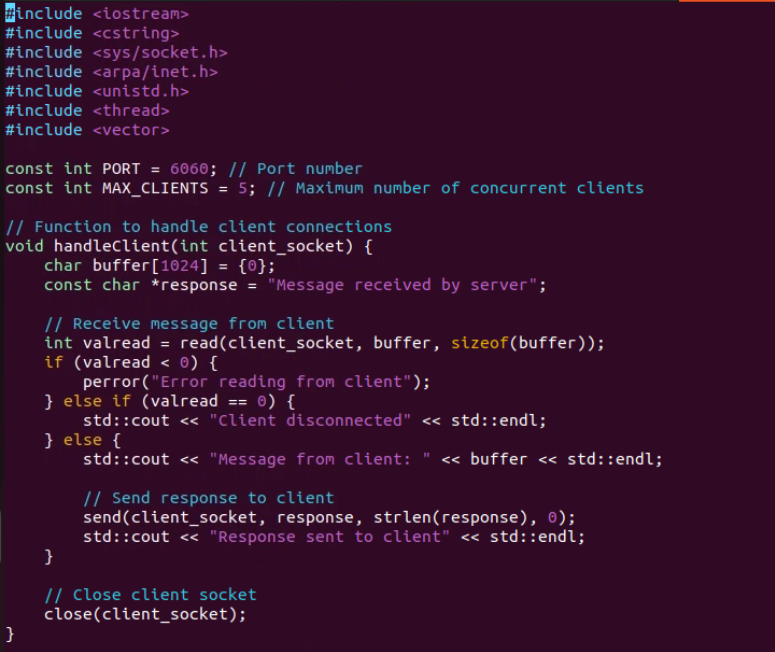
**Accepts multiple client connections concurrently.**

**Receives a message from each client and sends a unique response back.**

**Closes the connections and terminates after handling all clients.**

**TCP Client with Error Handling:**

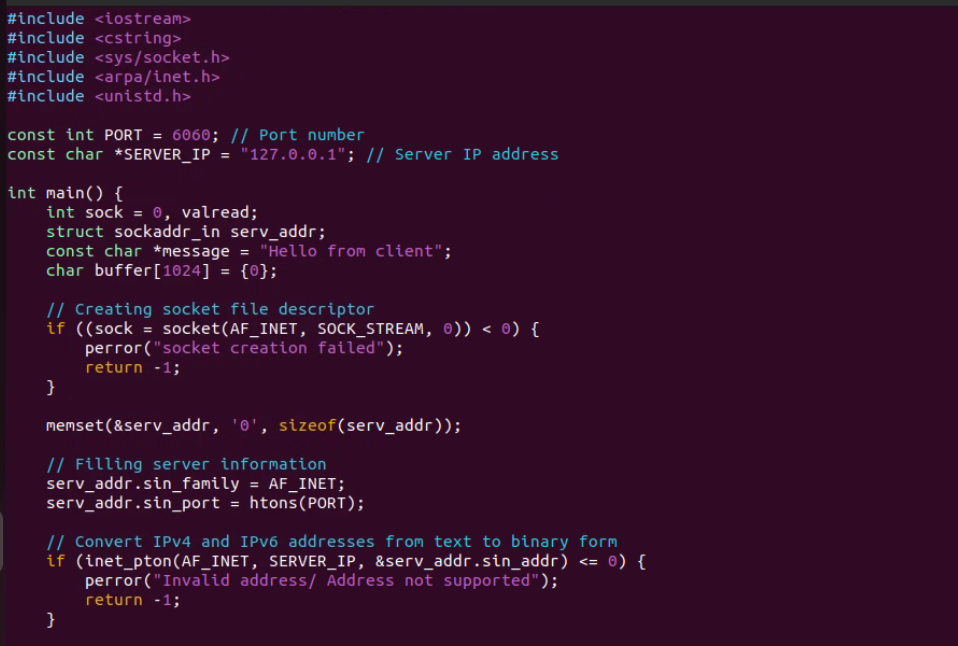
**Tcp\_server:**

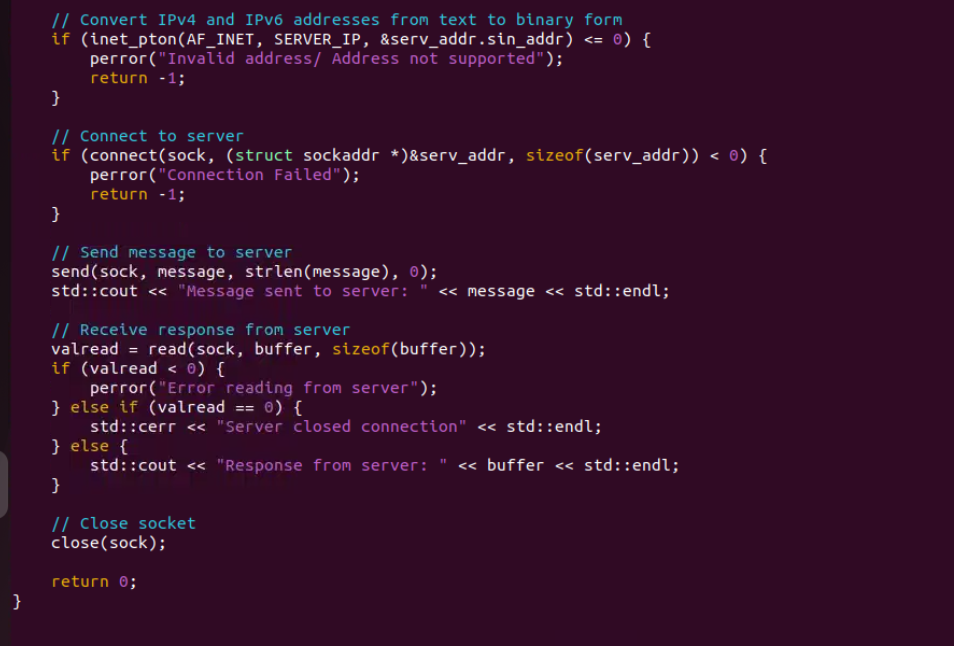
****

****

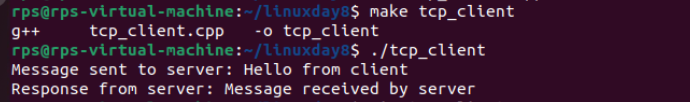
****

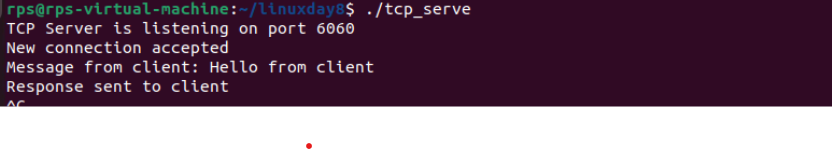
**Tcp\_client:**

****

****

**Output:**

****

****

**5. UDP Client-Server Communication:**

**Create a UDP server that: Binds to port 7070.**

**Receives a message from a client.**

**Sends a response message back to the client.**

**Closes the socket and terminates.**

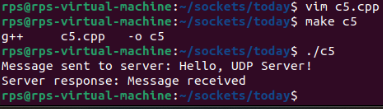
**Create a UDP client that:**

**Sends a message to the server on port 7070.**

**Receives and prints the response message from the server.**

**Closes the socket and terminates.**

**CLIENT:**

****

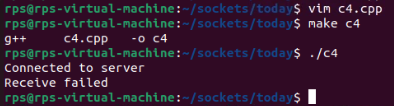
**Server:**

****

**6. Implement a TCP server that:**

**Binds to port 4040. Listens for incoming connections. Uses select() to handle multiple client connections. Receives a message from each client and sends a response back. Closes the connections and terminates.**

**Client:**

****

**#include <iostream>**

**#include <cstring> // For memset**

**#include <unistd.h> // For close**

**#include <sys/types.h>**

**#include <sys/socket.h>**

**#include <netinet/in.h>**

**#include <arpa/inet.h>**

**#include <sys/select.h>**

**#include <fcntl.h> // For fcntl**

**#define SERVER\_PORT 4040**

**#define SERVER\_IP "127.0.0.1" // Localhost for testing**

**#define BUFFER\_SIZE 1024**

**#define TIMEOUT\_SEC 5**

**int main() {**

**int sock = socket(AF\_INET, SOCK\_STREAM, 0);**

**if (sock < 0) {**

**std::cerr << "Socket creation failed\n";**

**return 1; }**

**int flags = fcntl(sock, F\_GETFL, 0);**

**fcntl(sock, F\_SETFL, flags | O\_NONBLOCK);**

**sockaddr\_in server\_address;**

**memset(&server\_address, 0, sizeof(server\_address));**

**server\_address.sin\_family = AF\_INET;**

**server\_address.sin\_port = htons(SERVER\_PORT);**

**inet\_pton(AF\_INET, SERVER\_IP, &server\_address.sin\_addr);**

**if (connect(sock, (struct sockaddr\*)&server\_address, sizeof(server\_address)) < 0) {**

**if (errno != EINPROGRESS) {**

**std::cerr << "Connection failed\n";**

**close(sock);**

**return 1; }**

**}**

**fd\_set write\_fds;**

**struct timeval timeout;**

**int select\_result;**

**FD\_ZERO(&write\_fds);**

**FD\_SET(sock, &write\_fds);**

**timeout.tv\_sec = TIMEOUT\_SEC;**

**timeout.tv\_usec = 0;**

**select\_result = select(sock + 1, nullptr, &write\_fds, nullptr, &timeout);**

**if (select\_result == -1) {**

**std::cerr << "Select error\n";**

**close(sock);**

**return 1;**

**} else if (select\_result == 0) {**

**std::cerr << "Connection timed out\n";**

**close(sock);**

**return 1;**

**} else if (FD\_ISSET(sock, &write\_fds)) {**

**std::cout << "Connected to server\n"; }**

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

**ssize\_t bytes\_sent = send(sock, message, strlen(message), 0);**

**if (bytes\_sent < 0) {**

**std::cerr << "Send failed\n";**

**close(sock);**

**return 1; }**

**char buffer[BUFFER\_SIZE];**

**memset(buffer, 0, BUFFER\_SIZE);**

**ssize\_t bytes\_received = recv(sock, buffer, BUFFER\_SIZE, 0);**

**if (bytes\_received < 0) {**

**std::cerr << "Receive failed\n";**

**close(sock);**

**return 1;**

**} else if (bytes\_received == 0) {**

**std::cout << "Server closed the connection\n";**

**} else {**

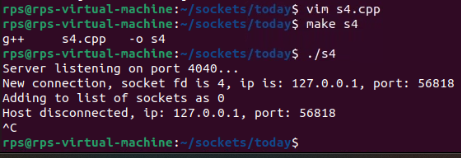
**std::cout << "Received message from server: " << buffer << "\n"; }**

**close(sock);**

**return 0;**

**}**

**SERVER:**

****

**#include <iostream>**

**#include <cstring> // For memset**

**#include <unistd.h> // For close**

**#include <sys/types.h>**

**#include <sys/socket.h>**

**#include <netinet/in.h>**

**#include <arpa/inet.h>**

**#include <sys/select.h>**

**#define PORT 4040**

**#define BUFFER\_SIZE 1024**

**#define MAX\_CLIENTS 10**

**int main() {**

**int server\_fd, new\_socket, client\_socket[MAX\_CLIENTS], max\_clients = MAX\_CLIENTS, activity, i, valread, sd;**

**int max\_sd;**

**struct sockaddr\_in address;**

**char buffer[BUFFER\_SIZE];**

**int addrlen = sizeof(address);**

**for (i = 0; i < max\_clients; i++) {**

**client\_socket[i] = 0;**

**} server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);**

**if (server\_fd < 0) {**

**std::cerr << "Socket creation failed\n";**

**return 1;**

**} address.sin\_family = AF\_INET;**

**address.sin\_addr.s\_addr = INADDR\_ANY;**

**address.sin\_port = htons(PORT);**

**if (bind(server\_fd, (struct sockaddr\*)&address, sizeof(address)) < 0) {**

**std::cerr << "Bind failed\n";**

**close(server\_fd);**

**return 1; }**

**if (listen(server\_fd, 3) < 0) {**

**std::cerr << "Listen failed\n";**

**close(server\_fd);**

**return 1; }**

**std::cout << "Server listening on port " << PORT << "...\n";**

**fd\_set readfds;**

**while (true) {**

**FD\_ZERO(&readfds);**

**FD\_SET(server\_fd, &readfds);**

**max\_sd = server\_fd;**

**for (i = 0; i < max\_clients; i++) {**

**sd = client\_socket[i];**

**if (sd > 0) {**

**FD\_SET(sd, &readfds);**

**}**

**if (sd > max\_sd) {**

**max\_sd = sd; }**

**}**

**activity = select(max\_sd + 1, &readfds, NULL, NULL, NULL);**

**if ((activity < 0) && (errno != EINTR)) {**

**std::cerr << "Select error\n"; }**

**if (FD\_ISSET(server\_fd, &readfds)) {**

**int addrlen = sizeof(address);**

**if ((new\_socket = accept(server\_fd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen)) < 0) {**

**std::cerr << "Accept failed\n";**

**close(server\_fd);**

**return 1;**

**} std::cout << "New connection, socket fd is " << new\_socket << ", ip is: "**

**<< inet\_ntoa(address.sin\_addr) << ", port: " << ntohs(address.sin\_port) << "\n";**

**for (i = 0; i < max\_clients; i++) {**

**if (client\_socket[i] == 0) {**

**client\_socket[i] = new\_socket;**

**std::cout << "Adding to list of sockets as " << i << "\n";**

**break; } }**

**}**

**for (i = 0; i < max\_clients; i++) {**

**sd = client\_socket[i];**

**if (FD\_ISSET(sd, &readfds)) {**

**if ((valread = read(sd, buffer, BUFFER\_SIZE)) == 0) {**

**getpeername(sd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen);**

**std::cout << "Host disconnected, ip: " << inet\_ntoa(address.sin\_addr) << ", port: " << ntohs(address.sin\_port) << "\n";**

**close(sd);**

**client\_socket[i] = 0;**

**} else {**

**buffer[valread] = '\0';**

**send(sd, buffer, strlen(buffer), 0); }**

**} }**

**} close(server\_fd);**

**return 0;**

**}**

**7. TCP Client with Error Handling:**

**Create a TCP client that:**

**Connects to a server at port 5050.**

**Sends a message to the server.**

**Handles and displays error messages for common issues such as connection failure or data transmission errors.**

**Receives and prints the response message from the server.**

**Closes the socket and terminates.**

**Client:**

**#include <iostream>**

**#include <cstring>**

**#include <sys/socket.h>**

**#include <netinet/in.h>**

**#include <arpa/inet.h>**

**#include <unistd.h>**

**#include <cerrno>**

**using namespace std;**

**void tcp\_client() {**

**int client\_socket = socket(AF\_INET, SOCK\_STREAM, 0);**

**if (client\_socket == -1) {**

**perror("Socket creation failed");**

**return;**

**} struct sockaddr\_in server\_address;**

**memset(&server\_address, 0, sizeof(server\_address));**

**server\_address.sin\_family = AF\_INET;**

**server\_address.sin\_port = htons(5050);**

**if (inet\_pton(AF\_INET, "127.0.0.1", &server\_address.sin\_addr) <= 0) {**

**perror("Invalid address or Address not supported");**

**close(client\_socket);**

**return;**

**}**

**if (connect(client\_socket, (struct sockaddr\*)&server\_address, sizeof(server\_address)) < 0) {**

**perror("Connection failed");**

**close(client\_socket);**

**return;**

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

**if (send(client\_socket, message, strlen(message), 0) < 0) {**

**perror("Send failed");**

**close(client\_socket);**

**return; }**

**cout << "Message sent to server: " << message << endl;**

**char buffer[1024];**

**ssize\_t valread = read(client\_socket, buffer, sizeof(buffer) - 1);**

**if (valread < 0) {**

**perror("Read failed");**

**close(client\_socket);**

**return;**

**}**

**buffer[valread] = '\0'; // Null-terminate the received data**

**cout << "Server response: " << buffer << endl;**

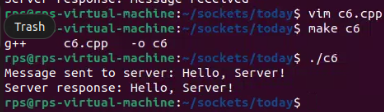
**close(client\_socket); }**

**int main() {**

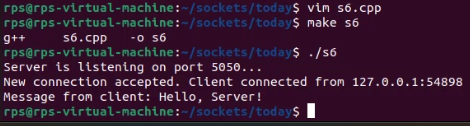
**tcp\_client();**

**return 0;**

**}**

****

**Server:**

****

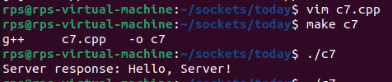
****

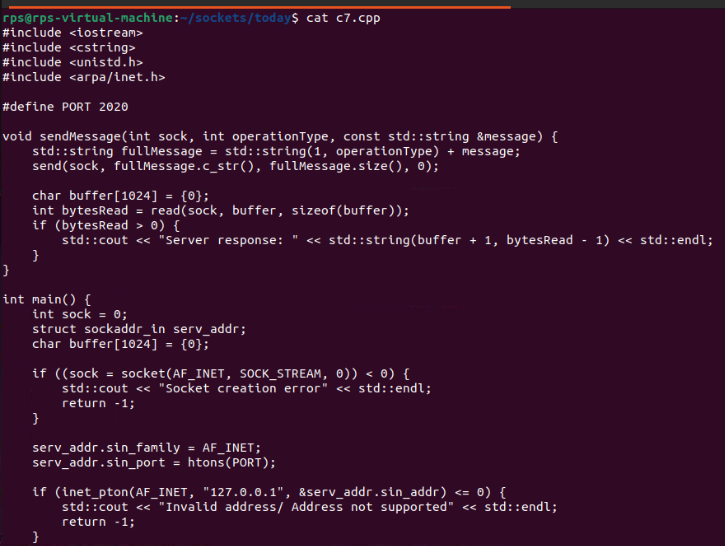
****

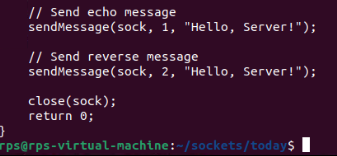
**8. TCP Server with Custom Protocol:**

**Implement a TCP server that: Binds to port 2020. Listens for incoming connections. Implements a simple custom protocol where: The first byte of the message indicates the type of operation (e.g., 1 for echo, 2 for reverse). For operation type 1, the server echoes the message back. For operation type 2, the server sends back the reversed message. Closes the connection and terminates.**

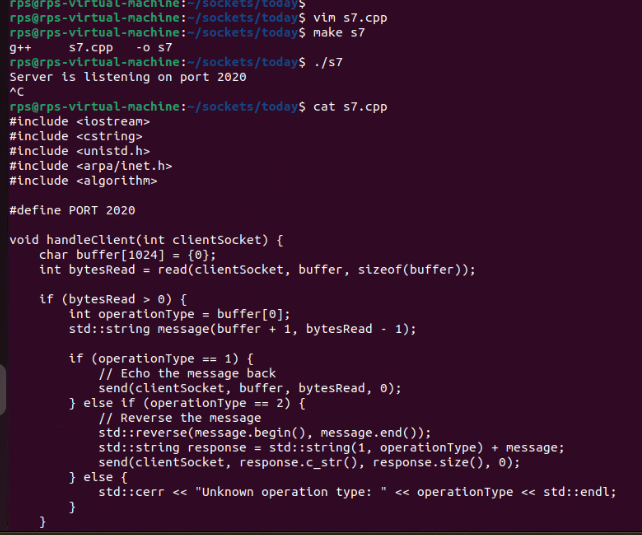
**Client**

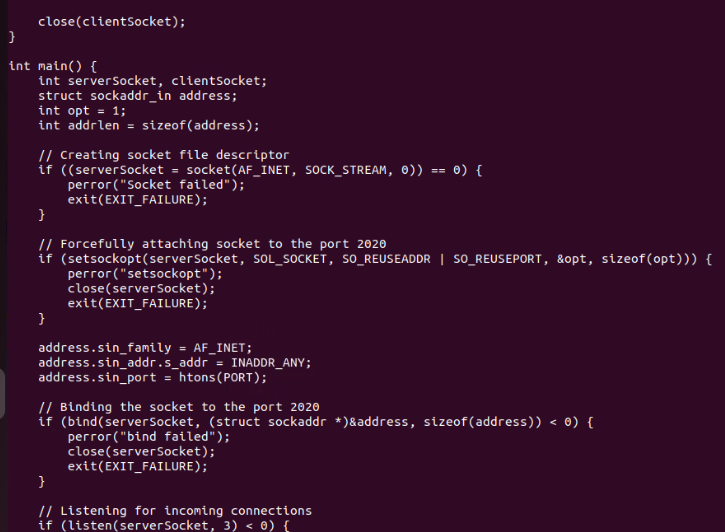
****

****

****

**Server:**

****

****

