Ping - pong app

server:

```
#include <iostream>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>

using namespace std;
int main() {
    // Create a UDP socket
    int serverSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (serverSocket == -1) {
        std::cerr << "Error creating socket\n";
        return -1;
    }

    // Bind the socket to an address and port
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY; // Listen on any available interface
    serverAddress.sin_port = htons(8080); // Use port 8080
    if (bind(serverSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) == -1) {
        std::cerr << "Error binding socket\n";
        close(serverSocket);
        return -1;
    }

    cout<<"Server is running on port 8080\n";
    char buffer[1024];
    sockaddr_in clientAddress;
    socklen_t clientAddrLen = sizeof(clientAddress);

    int server_timeout = 10;
    // while (true) { // Run forever
    while(server_timeout--){ // run for 10 packets
        // Receive data from the client
        ssize_t bytesRead = recvfrom(serverSocket, buffer, sizeof(buffer), 0, (struct
sockaddr*)&clientAddress, &clientAddrLen);
        if (bytesRead == -1) {
            std::cerr << "Error receiving data\n";
            close(serverSocket);
            return -1;
        }
```

```
        // Print received message
        std::cout << "Received ping message: " << buffer << "\n";

        // Send a pong message back to the client
        const char* pongMessage = "pong";
        ssize_t bytesSent = sendto(serverSocket, pongMessage, strlen(pongMessage), 0,
                                   (struct sockaddr*)&clientAddress, clientAddrLen);
        if (bytesSent == -1) {
            std::cerr << "Error sending pong message\n";
            close(serverSocket);
            return -1;
        }
        cout<<"Pong message sent\n";
    }

    // Close the socket
    close(serverSocket);
    return 0;
}


Client:
#include <iostream>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <ctime>

#define PING_COUNT 10
#define TIMEOUT 1  // Timeout in seconds
using namespace std;
int main() {
    // Create a UDP socket
    int clientSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (clientSocket == -1) {
        std::cerr << "Error creating socket\n";
        return -1;
    }

    cout<<"Client is running\n";
    // Set up the server address
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
```

```
    serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server's IP address
    serverAddress.sin_port = htons(8080); // Server's port

    // Set the timeout for receiving data
    struct timeval tv;
    tv.tv_sec = TIMEOUT;
    tv.tv_usec = 0;
    if (setsockopt(clientSocket, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) < 0) {
        std::cerr << "Error setting socket timeout\n";
        close(clientSocket);
        return -1;
    }

    char message[] = "ping";
    char buffer[1024];
    sockaddr_in serverReply;
    socklen_t serverAddrLen = sizeof(serverReply);

    double totalRTT = 0;

    for (int i = 0; i < PING_COUNT; ++i) {
        cout<<"Sending ping message "<<i+1<<"\n";
        // Record the current time before sending the message
        clock_t startTime = clock();

        // Send the ping message to the server
        ssize_t bytesSent = sendto(clientSocket, message, strlen(message), 0,
                                   (struct sockaddr*)&serverAddress, sizeof(serverAddress));
        if (bytesSent == -1) {
            std::cerr << "Error sending ping message\n";
            close(clientSocket);
            return -1;
        }

        // Wait for the pong message from the server
        ssize_t bytesReceived = recvfrom(clientSocket, buffer, sizeof(buffer), 0, (struct
sockaddr*)&serverReply, &serverAddrLen);
        if (bytesReceived == -1) {
            std::cerr << "Request timed out, packet lost\n";
        } else {
            // Record the time after receiving the pong message
            clock_t endTime = clock();

            // Calculate and print the Round Trip Time (RTT)
```

```
            double rtt = double(endTime - startTime) / CLOCKS_PER_SEC * 1000.0; // RTT in
milliseconds
            totalRTT += rtt;
            std::cout << "Ping " << i + 1 << " RTT: " << rtt << " ms\n";
        }
    }
    cout<<"Average RTT: "<<totalRTT/PING_COUNT<<" ms\n";
    cout<<"Closing client\n";

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

File Transfer using Stream Socket

Server:

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

#define PORT 8080
#define BUFFER_SIZE 100

int main() {
    int server_fd, client_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);

    // Create socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Configure server address
```

```
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind socket
    if (bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Listen for connections
    if (listen(server_fd, 5) < 0) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d\n", PORT);

    while (1) {
        // Accept client connection
        client_fd = accept(server_fd, (struct sockaddr*)&client_addr, &addr_len);
        if (client_fd < 0) {
            perror("accept");
            continue;
        }

        // Receive filename from client
        char filename[BUFFER_SIZE] = {0};
        ssize_t bytes_received;
        size_t total_received = 0;
        int filename_received = 0;

        while (total_received < sizeof(filename) - 1) {
            bytes_received = recv(client_fd, filename + total_received, sizeof(filename) -
total_received - 1, 0);
            if (bytes_received <= 0) break;
            total_received += bytes_received;
            if (strchr(filename, '\0')) {
                filename_received = 1;
                break;
            }
```

```
        }

        if (!filename_received) {
            close(client_fd);
            continue;
        }

        // Open file
        int file_fd = open(filename, O_RDONLY);
        if (file_fd == -1) {
            close(client_fd);
            continue;
        }

        // Read and send file in chunks
        char buffer[BUFFER_SIZE];
        ssize_t bytes_read;

        while ((bytes_read = read(file_fd, buffer, BUFFER_SIZE)) > 0) {
            ssize_t bytes_sent = 0;
            while (bytes_sent < bytes_read) {
                ssize_t sent = send(client_fd, buffer + bytes_sent, bytes_read - bytes_sent, 0);
                if (sent == -1) {
                    perror("send");
                    break;
                }
                bytes_sent += sent;
            }
        }

        close(file_fd);
        close(client_fd);
    }

    close(server_fd);
    return 0;
}
```

Client:

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

```c
#include <arpa/inet.h>
#include <fcntl.h>
#include <errno.h>

#define SERVER_IP "127.0.0.1"
#define PORT 8080
#define BUFFER_SIZE 100

int is_delimiter(char c) {
    return (c == ' ' || c == ',' || c == ';' || c == ':' || c == '.' || c == '\t' || c == '\n' || c == '\r');
}

int main() {
    int sock_fd;
    struct sockaddr_in server_addr;

    // Create socket
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Configure server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

    // Connect to server
    if (connect(sock_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("connect");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    // Get filename from user
    char filename[BUFFER_SIZE];
    printf("Enter filename: ");
    fgets(filename, BUFFER_SIZE, stdin);
    filename[strcspn(filename, "\n")] = '\0';

    // Send filename to server
    if (send(sock_fd, filename, strlen(filename) + 1, 0) == -1) {
```

```
        perror("send");
        close(sock_fd);
        exit(EXIT_FAILURE);
    }

    // Receive file and count words/bytes
    char buffer[BUFFER_SIZE];
    ssize_t bytes_received;
    int output_fd = open("received_file.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    size_t total_bytes = 0;
    int in_word = 0, word_count = 0;

    while ((bytes_received = recv(sock_fd, buffer, BUFFER_SIZE, 0)) > 0) {
        total_bytes += bytes_received;
        write(output_fd, buffer, bytes_received);

        for (int i = 0; i < bytes_received; i++) {
            if (is_delimiter(buffer[i])) {
                in_word = 0;
            } else {
                if (!in_word) {
                    word_count++;
                    in_word = 1;
                }
            }
        }
    }

    close(output_fd);
    close(sock_fd);

    if (total_bytes == 0) {
        printf("ERR 01: File Not Found\n");
        remove("received_file.txt");
    } else {
        printf("File transfer successful. Size: %zu bytes, Words: %d\n", total_bytes, word_count);
    }

    return 0;
}
```

Socket-based Remote Command Execution
Server:

```
// server.cpp
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cstdlib>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <ctime>
#include <pwd.h>

#define PORT 8080
#define BUFF_SIZE 1024

using namespace std;

// Function to get server information (hostname, user, date/time)
string getServerInfo() {
    char hostBuffer[256];
    gethostname(hostBuffer, sizeof(hostBuffer));

    struct passwd *pw = getpwuid(getuid());
    string user = (pw) ? pw->pw_name : "unknown";

    time_t now = time(0);
    char* dt = ctime(&now);

    string info = "Server Info:\nHostname: " + string(hostBuffer) +
                  "\nUser: " + user +
                  "\nDate & Time: " + string(dt) + "\n";
    return info;
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);

    // Create the TCP socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
```

```
}

// Allow the socket to reuse address and port immediately after close
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
               &opt, sizeof(opt))) {
    perror("setsockopt failure");
    exit(EXIT_FAILURE);
}

// Configure server address settings
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(PORT);

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

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

cout << "Server is listening on port " << PORT << endl;

while (true) {
    // Accept an incoming connection
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                             (socklen_t*)&addrlen)) < 0) {
        perror("Accept failed");
        exit(EXIT_FAILURE);
    }

    // Send initial server info to the client
    string serverInfo = getServerInfo();
    send(new_socket, serverInfo.c_str(), serverInfo.length(), 0);
    cout << "Client connected. Sent server information." << endl;

    char command[BUFF_SIZE];
    while (true) {
        memset(command, 0, BUFF_SIZE);
```

```
int bytesRead = recv(new_socket, command, BUFF_SIZE - 1, 0);
if (bytesRead <= 0) {
    cout << "Client disconnected or an error occurred." << endl;
    break;
}
command[bytesRead] = '\0';

// Terminate connection if the client sends "exit"
if (strncmp(command, "exit", 4) == 0) {
    cout << "Exit command received. Closing connection." << endl;
    break;
}
cout << "Received command: " << command <<"\n";

string full_command = string(command) + " 2>&1";  // Redirect stderr to stdout
FILE *fp = popen(full_command.c_str(), "r");
if (fp == NULL) {
    string errorMsg = "Failed to execute command.\n";
    send(new_socket, errorMsg.c_str(), errorMsg.length(), 0);
    continue;
}

// Read command output
char output[BUFF_SIZE];
string result = "";
while (fgets(output, sizeof(output), fp) != NULL) {
    result += output;
}

// Check command exit status
int status = pclose(fp);
if (WIFEXITED(status)) {
    int exit_code = WEXITSTATUS(status);
    if (exit_code == 127) {  // Standard error code for "command not found"
        result = "Error: Invalid command. The command was not recognized.\n";
    }
} else {
    result = "Error: Command execution failed abnormally.\n";
}

// Handle empty output
if (result.empty()) {
    result = "Command executed successfully, but no output returned.\n";
}
```

```
            // Send final response
            send(new_socket, result.c_str(), result.length(), 0);
        }
        close(new_socket);
        cout << "Closed connection with client." << endl;
    }

    close(server_fd);
    return 0;
}

// commands :
// g++ server.cpp -o server
// ./server
```

Client:

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

#define PORT 8080
#define BUFF_SIZE 1024

using namespace std;

void write_to_log(const string& output , const string& cmd) {
    ofstream logfile("client_log.txt", ios::app);
    if (logfile.is_open()) {
        logfile << "=== Command ===\n";
        logfile << "store " << cmd << "\n\n";
        logfile << "=== Server Response ===\n";
        logfile << output << "\n\n";
        logfile.close();
    } else {
        cerr << "Error opening log file!" << endl;
    }
}
```

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

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        cout << "Socket creation error" << endl;
        return -1;
    }

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

    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        cout << "Invalid address / Address not supported" << endl;
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        cout << "Connection Failed" << endl;
        return -1;
    }

    int valread = recv(sock, buffer, BUFF_SIZE - 1, 0);
    if (valread > 0) {
        buffer[valread] = '\0';
        cout << buffer << endl;
    }

    while (true) {
        cout << "Enter command (or 'exit' to disconnect): ";
        string cmd;
        getline(cin, cmd);

        if (cmd.empty()) continue;

        bool store_output = false;
        if (cmd.rfind("store ", 0) == 0) { // Check if command starts with "store "
            store_output = true;
            cmd = cmd.substr(6); // Remove "store " from the command
        }

        send(sock, cmd.c_str(), cmd.length(), 0);
```

```
        if (cmd == "exit") {
            cout << "Exiting..." << endl;
            break;
        }

        memset(buffer, 0, BUFF_SIZE);
        valread = recv(sock, buffer, BUFF_SIZE - 1, 0);
        if (valread > 0) {
            buffer[valread] = '\0';
            string output(buffer);

            if (store_output) {
                write_to_log(output , cmd);
                cout << "Output stored in log file." << endl;
            } else {
                cout << "Output from server:\n" << output << endl;
            }
        }
    }

    close(sock);
    return 0;
}
```

Simple File Server with Directory Listing and File Download

Server:

```
#include <iostream>
#include <string>
#include <vector>
#include <thread>
#include <mutex>
#include <fstream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>
#include <cstring>

const int BUFFER_SIZE = 1024;
const int PORT = 12345;
std::mutex log_mutex;
```

```
void log_message(const std::string& message) {
    std::lock_guard<std::mutex> lock(log_mutex);
    std::ofstream logfile("server.log", std::ios::app);
    if (logfile.is_open()) {
        logfile << message << std::endl;
    }
}

std::vector<std::string> list_files() {
    std::vector<std::string> files;
    DIR *dir;
    struct dirent *ent;

    if ((dir = opendir(".")) != nullptr) {
        while ((ent = readdir(dir)) != nullptr) {
            if (ent->d_type == DT_REG) {
                files.push_back(ent->d_name);
            }
        }
        closedir(dir);
    }
    return files;
}

void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE];
    std::string welcome = "Welcome to Simple File Server\n";
    send(client_socket, welcome.c_str(), welcome.size(), 0);

    while (true) {
        memset(buffer, 0, BUFFER_SIZE);
        int bytes_received = recv(client_socket, buffer, BUFFER_SIZE, 0);

        if (bytes_received <= 0) break;

        std::string command(buffer);
        command = command.substr(0, command.find('\n'));
        log_message("Command received: " + command);

        if (command == "LIST") {
            auto files = list_files();
            std::string response;
            for (const auto& file : files) {
```

```
            response += file + "\n";
        }
        response += "END_OF_LIST\n";
        send(client_socket, response.c_str(), response.size(), 0);
    }
    else if (command.substr(0, 3) == "GET") {
        std::string filename = command.substr(4);
        struct stat file_stat;

        if (stat(filename.c_str(), &file_stat) == -1) {
            std::string error = "ERROR: File Not Found\n";
            send(client_socket, error.c_str(), error.size(), 0);
            continue;
        }

        std::ifstream file(filename, std::ios::binary);
        if (!file.is_open()) {
            std::string error = "ERROR: File Access Denied\n";
            send(client_socket, error.c_str(), error.size(), 0);
            continue;
        }

        std::string header = "FILESIZE " + std::to_string(file_stat.st_size) + "\n";
        send(client_socket, header.c_str(), header.size(), 0);

        char file_buffer[BUFFER_SIZE];
        while (!file.eof()) {
            file.read(file_buffer, BUFFER_SIZE);
            send(client_socket, file_buffer, file.gcount(), 0);
        }
        file.close();
    }
    else if (command == "QUIT") {
        std::string goodbye = "Goodbye!\n";
        send(client_socket, goodbye.c_str(), goodbye.size(), 0);
        break;
    }
    else {
        std::string error = "ERROR: Invalid Command\n";
        send(client_socket, error.c_str(), error.size(), 0);
    }
}
close(client_socket);
log_message("Client disconnected");
```

```
}

int main() {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("Socket creation failed");
        return 1;
    }

    sockaddr_in address;
    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) {
        perror("Bind failed");
        return 1;
    }

    if (listen(server_fd, 5) < 0) {
        perror("Listen failed");
        return 1;
    }

    std::cout << "Server listening on port " << PORT << std::endl;

    while (true) {
        sockaddr_in client_addr;
        socklen_t addr_len = sizeof(client_addr);
        int client_socket = accept(server_fd, (struct sockaddr*)&client_addr, &addr_len);

        if (client_socket < 0) {
            perror("Accept failed");
            continue;
        }

        std::thread(handle_client, client_socket).detach();
        log_message("New client connected");
    }

    close(server_fd);
    return 0;
}
```

Client:

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

const int BUFFER_SIZE = 1024;
const int PORT = 12345;

void receive_list(int sock) {
    char buffer[BUFFER_SIZE];
    std::string full_response;

    while (true) {
        int bytes_received = recv(sock, buffer, BUFFER_SIZE, 0);
        if (bytes_received <= 0) break;

        full_response.append(buffer, bytes_received);
        if (full_response.find("END_OF_LIST") != std::string::npos) {
            break;
        }
    }

    size_t end_pos = full_response.find("END_OF_LIST");
    std::cout << "\nAvailable files:\n"
              << full_response.substr(0, end_pos) << std::endl;
}

void receive_file(int sock, const std::string& filename) {
    char buffer[BUFFER_SIZE];
    int bytes_received = recv(sock, buffer, BUFFER_SIZE, 0);

    if (bytes_received <= 0) {
        std::cout << "Error receiving file header" << std::endl;
        return;
    }

    std::string header(buffer, bytes_received);
    if (header.substr(0, 8) != "FILESIZE") {
        std::cout << header;
        return;
```

```
    }

    // Generate client copy filename
    size_t dot_pos = filename.find_last_of('.');
    std::string output_name;
    if (dot_pos != std::string::npos) {
        output_name = filename.substr(0, dot_pos) +
                      "_clientcopy" +
                      filename.substr(dot_pos);
    } else {
        output_name = filename + "_clientcopy";
    }

    size_t space_pos = header.find(' ');
    size_t newline_pos = header.find('\n');
    long file_size = std::stol(header.substr(space_pos + 1, newline_pos - space_pos - 1));

    std::ofstream file(output_name, std::ios::binary);
    if (!file.is_open()) {
        std::cout << "Error creating local file" << std::endl;
        return;
    }

    long total_received = 0;
    while (total_received < file_size) {
        bytes_received = recv(sock, buffer, BUFFER_SIZE, 0);
        if (bytes_received <= 0) break;

        // Write received chunk to file
        file.write(buffer, bytes_received);
        total_received += bytes_received;

        // Optional: Display chunk info
        std::cout << "Received chunk: " << bytes_received << " bytes" << std::endl;
    }

    file.close();
    std::cout << "Total received: " << total_received << " bytes" << std::endl;
    std::cout << "File saved as: " << output_name << std::endl;
}


int main() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
if (sock == -1) {
    perror("Socket creation failed");
    return 1;
}

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

if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
    perror("Invalid address");
    return 1;
}

if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
    perror("Connection failed");
    return 1;
}

char welcome[BUFFER_SIZE];
recv(sock, welcome, BUFFER_SIZE, 0);
std::cout << welcome;

while (true) {
    std::cout << "\nEnter command (LIST/GET/QUIT): ";
    std::string command;
    std::getline(std::cin, command);

    if (command == "LIST") {
        send(sock, "LIST\n", 5, 0);
        receive_list(sock);
    }
    else if (command.substr(0, 3) == "GET") {
        send(sock, (command + "\n").c_str(), command.size() + 1, 0);
        receive_file(sock, command.substr(4));
    }
    else if (command == "QUIT") {
        send(sock, "QUIT\n", 5, 0);
        break;
    }
    else {
        std::cout << "Invalid command" << std::endl;
    }
}
```

```
    close(sock);
    std::cout << "Connection closed" << std::endl;
    return 0;
}
```