1. Server code

   (a) Screenshot or well-formatted copy of code

```cpp
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h> // for perror()
#include <unistd.h> // for close()

// load environment variables from .env file
#include <fstream>
#include <cstdlib>

void load_env(const std::string& path) {
    std::ifstream f(path);
    std::string line;
    while (std::getline(f, line)) {
        if (line.empty() || line[0] == '#') continue;
        auto pos = line.find('=');
        if (pos == std::string::npos) continue;

        std::string key = line.substr(0, pos);
        std::string val = line.substr(pos + 1);

        #ifdef _WIN32
                _putenv_s(key.c_str(), val.c_str());
        #else
                setenv(key.c_str(), val.c_str(), 1);
        #endif
    }
}

int main(void){
    printf("Server starting...\n");
    load_env(".env");

    // Declare variables
    const char *server_ip = std::getenv("SERVER_IP");
    const int server_port = std::atoi(std::getenv("SERVER_PORT"
       ));
    char client_message[2048];
    char server_message[2048];
    const char * custom_message="Server: Hello from server, ";

    // debug
```

```
printf("Server starting at IP: %s, Port: %d\n", server_ip,
    server_port);

// Create socket
const int server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket == -1) {
    perror("Failed to create socket");
    return 1;
}

// Bind to the set port and IP
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(server_port);
inet_pton(AF_INET, server_ip, &server_addr.sin_addr);

if (bind(server_socket, (struct sockaddr*)&server_addr,
    sizeof(server_addr)) == -1) {
    perror("Failed to bind socket");
    close(server_socket);
    return 1;
}
printf("Done with binding with IP: %s, Port: %d\n",
    server_ip, server_port);

// Listen for clients:
const char *client_ip = std::getenv("CLIENT_IP");
if (listen(server_socket, 1) == -1) {
    perror("Failed to listen on socket");
    close(server_socket);
    return 1;
}
printf("Listening for incoming connections...\n");

// Accept an incoming connection
struct sockaddr_in client_addr;
socklen_t client_addr_len = sizeof(client_addr);
int client_socket = accept(server_socket, (struct sockaddr
    *)&client_addr, &client_addr_len);
if (client_socket == -1) {
    perror("Failed to accept connection");
    close(server_socket);
    return 1;
}
printf("Client connected at IP: %s \n", client_ip);
```

```
    // Receive client's message

    // clean exising buffer
    memset(client_message, 0, sizeof(client_message));

    recv(client_socket, client_message, sizeof(client_message),
        0);
    printf("Msg from client: %s", client_message);

    // Respond to client
    // prepare server message
    memset(server_message, 0, sizeof(server_message));
    ;

    // add my name in the back as response.
    std::snprintf(server_message, sizeof(server_message), "%s%s
        ", custom_message, strtok(client_message, ":"));
    if (send(client_socket, server_message, sizeof(
        server_message), 0) == -1) {
        perror("Failed to send message");
        close(client_socket);
        close(server_socket);
        return 1;
    }
    printf("Response sent to client: %s\n", server_message);

    // Close the socket
    close(client_socket);
    close(server_socket);

    return 0;
}
```

(b) Quick overview of what the code does in your own words

First, I created an environment file to ensure that the ip address from both cpp file gets the same ip address. I defined the ip as follows:

NET_SUBNET=172.30.0.0/24
SERVER_IP=172.30.0.10
CLIENT_IP=172.30.0.11
SERVER_PORT=3030

Secondly, I made `load_env` function by modifying the scripts from the internet.

After that, in main function, we set the variables server ip, server port, client message

buffer and server message buffer.

Then, we created a socket, bind it to the set port and IP. If then everything goes well, we listen for incoming connections from client side and retrieve client ip. After that, we accept an incoming connection, receive client's message, and respond to client using the client name defined in client code and concat with our custom response defined before.

The server will quit automatically when we response 1 message from client.

2. Client code

   (a) Screenshot or well-formatted copy of code

```
    #include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h>
#include <unistd.h> // for close()

// load environment variables from .env file
#include <fstream>
#include <cstdlib>

void load_env(const std::string& path) {
    std::ifstream f(path);
    std::string line;
    while (std::getline(f, line)) {
        if (line.empty() || line[0] == '#') continue;
        auto pos = line.find('=');
        if (pos == std::string::npos) continue;

        std::string key = line.substr(0, pos);
        std::string val = line.substr(pos + 1);

        #ifdef _WIN32
                _putenv_s(key.c_str(), val.c_str());
        #else
                setenv(key.c_str(), val.c_str(), 1);
        #endif
    }
}

int main(void){
    load_env(".env");
    // Declare variables
    const char *server_ip = std::getenv("SERVER_IP");
    const int server_port = std::atoi(std::getenv("SERVER_PORT"
       ));
    printf("Connecting to server %s:%d\n", server_ip,
       server_port);

    char client_message[1024];
    char server_message[1024];
    const char * custom_message="Zheyuan Wu: ";
```

```
// Create socket:
int client_socket = socket(AF_INET, SOCK_STREAM, 0);
if (client_socket == -1) {
    perror("Failed to create socket");
    return 1;
}else{
    printf("Socket created successfully\n");
}

// Send connection request to server, be sure to set por
   tand IP the same as server-side
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(server_port);
inet_pton(AF_INET, server_ip, &server_addr.sin_addr);

if (connect(client_socket, (struct sockaddr*)&server_addr,
   sizeof(server_addr)) == -1) {
    perror("Failed to connect to server");
    close(client_socket);
    return 1;
}else{
    printf("Connected to server successfully\n");
}

// Get input from the user:
printf("Enter message sent to the server (type \\quit to
   exit): ");
// clean the buffer
memset(client_message, 0, sizeof(client_message));
if (fgets(client_message, sizeof(client_message), stdin) ==
    NULL) {
    // EOF or error reading from stdin, exit the loop
    perror("Error reading from stdin");
    return 1;
}

while (strcmp(client_message, "\\quit\n") != 0) {

    // Send the message to server:
    // add my name in the front
    char buffer[2048];
    std::snprintf(buffer, sizeof(buffer), "%s%s",
       custom_message, client_message);
    send(client_socket, buffer, sizeof(buffer), 0);
```

```
        // Receive the server's response:
        recv(client_socket, server_message, sizeof(
            server_message), 0);


        printf("Server's response: %s\n", server_message);

        printf("Enter message sent to the server (type \\quit
            to exit): ");

        // clean the buffer
        memset(client_message, 0, sizeof(client_message));
        memset(server_message, 0, sizeof(server_message));

        if (fgets(client_message, sizeof(client_message), stdin
            ) == NULL) {
            // EOF or error reading from stdin, exit the loop
            perror("Error reading from stdin");
            break;
        }
    }

    // Close the socket
    close(client_socket);

    return 0;
}
```

(b) The message you send from the client to the server should include your name!

We use `custom_message` to ensure that each message we send to the server starts with our name.

(c) Quick overview of what the code does in your own words.

First we use the same function defined in `server.cpp` to load the environment file to ensure that the server and client gets the same ip address so I don't need to check it anymore.

Secondly, we build sockets that connects to server by address and port provided. If everything goes well, we will ask for user to input the message they like and send them to server with my name defined in `custom_message`. Then concat the string to the buffer and send them to the server.

After that we listen for response from the server and print it and wait for user to input the next message.

The server will quit automatically when we type `quit`.

3. Testing

   (a) Provide and overview of the setup you used to test your client/server program. You should not run them on the same host (same machine/simulated machine. Each machine should have a unique IP, use the Sniffing and Snooping SeedLab setup).

   I use docker compose to run the server and client on separate containers with independent IP addresses.

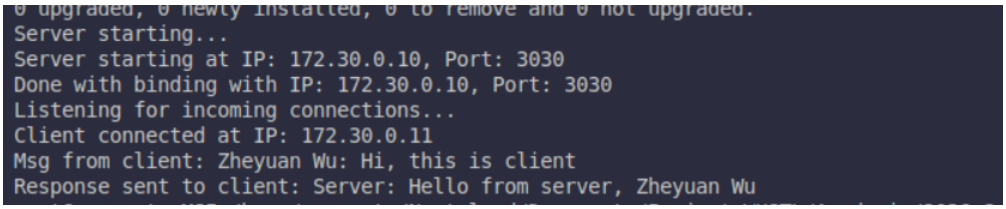
   The docker compose file is provided as follows and env is defined in before.

```
services:
  server:
    image: debian:bookworm-slim
    container_name: hw0-server
    working_dir: /app
    env_file:
      - .env
    networks:
      net-hw0:
        ipv4_address: ${SERVER_IP}
    volumes:
      - ./bin/server:/usr/local/bin/server:ro
    entrypoint: ["bash", "-lc", "apt-get update && apt-get
      install -y --no-install-recommends libstdc++6 iproute2; 
      exec stdbuf -oL -eL /usr/local/bin/server"]

  client:
    image: debian:bookworm-slim
    container_name: hw0-client
    working_dir: /app
    env_file:
      - .env
    networks:
      net-hw0:
        ipv4_address: ${CLIENT_IP}
    depends_on:
      - server
    stdin_open: true
    tty: true
    volumes:
      - ./bin/client:/usr/local/bin/client:ro
    entrypoint: ["bash", "-lc", "apt-get update && apt-get
      install -y --no-install-recommends libstdc++6 iproute2; 
      exec stdbuf -oL -eL /usr/local/bin/client"]

networks:
  net-hw0:
```

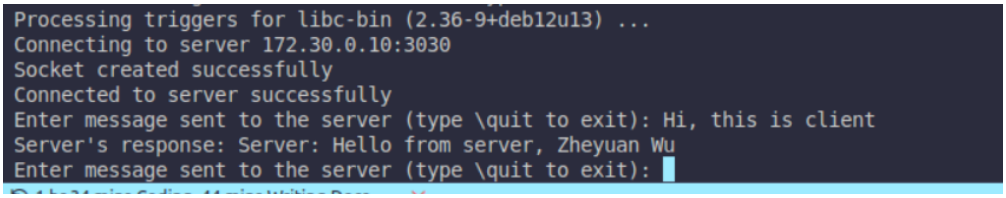
```
name: net-hw0
driver: bridge
ipam:
  config:
    - subnet: ${NET_SUBNET}
```

(b) Provide screenshots showing your test in action

i. Likely at least one of the server receiving the connection and message



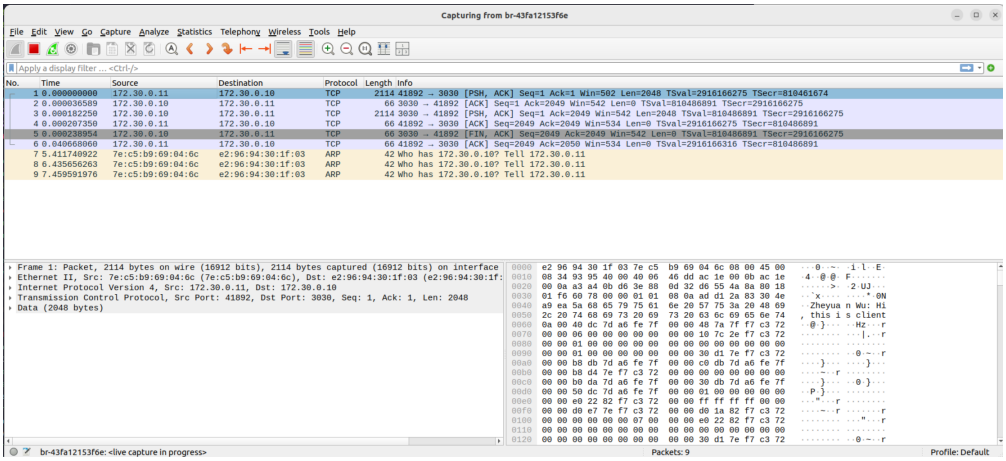
ii. Likely at least one of the client receiving and printing the return message from the server



4. Extra credit

   a. Run your client and server on the HostA and HostB on the SeedVM docker setup for studio 1. Prior to running the client/server, use the SeedAttacker VM to run wireshark and sniff all packets on the network. Use Wireshark to trace the TCP connection from client to server and find the packets containing the message sent from client to server. Is an eavesdropper able to obtain the contents of the message? Include a section in your report for extra credit. Document the work you did and include at least one screenshot showing the Wireshark packet capture. In this screenshot, make sure one of the packets containing the client message is the actively clicked on packet and the message/part of the message is visible in the detailed packet window.

   Here is the screenshot of wireshark packet capture for client message



   Here is the screenshot of wireshark packet capture for server message