Lab 1: Hello Network - Basic TCP Client/Server & Packet Analysis

Objective

In this lab, you will:

1. Set up and use a consistent development environment using Dev Containers.
2. Write a simple TCP server application in C++ that listens for connections.
3. Write a simple TCP client application in C++ that connects to the server.
4. Compile and run the client and server to establish basic network communication.
5. Capture the network traffic generated by your application using `tcpdump`.
6. Analyze the captured traffic using Wireshark to understand the underlying TCP communication flow.

Prerequisites

- Access to a computer with:
  - VS Code installed.
    - The "Dev Containers" extension installed in VS Code.
  - Git installed.
  - Either Orbstack (macOS), Docker Desktop (Windows/macOS/Linux), or access to GitHub Codespaces.
  - Wireshark installed locally (for analyzing captures).
- Basic familiarity with C++ programming concepts.
- The .devcontainer folder that can be found on Teams.

Part 1: Setting up the Development Environment

We will use Dev Containers to ensure everyone has the same tools and libraries, avoiding common "works on my machine" issues.

1. **Open the Project:**
   - Launch VS Code.
   - Go to `File > Open Folder...` (or `File > Open...` on macOS)

and select the directory where you cloned the course repository.
   - Alternatively, if using GitHub Codespaces, open the repository directly on GitHub and launch a new Codespace.
2. **Reopen in Container:**
   - VS Code should detect the `.devcontainer` folder and prompt you in the bottom-right corner: "Folder contains a Dev Container configuration file. Reopen folder to develop in a container."
   - Click the **"Reopen in Container"** button.
   - If you don't see the prompt, open the Command Palette (`Ctrl+Shift+P` or `Cmd+Shift+P`), type `Dev Containers: Reopen in Container`, and press Enter.
3. **Wait for Build:** The first time you do this, VS Code (using Orbstack/Docker/Codespaces) will build the container image based on the `Dockerfile`. This might take a few minutes. Subsequent launches will be much faster.
4. **Verify:** Once the container is running, your VS Code window will be connected to it. You should see the container details in the bottom-left corner (e.g., "Dev Container: C++ Network Programming"). Open a terminal within VS Code (`Terminal > New Terminal`). This terminal is running *inside* the container.

Part 2: Creating the TCP Server

Create the File: In the VS Code explorer, create a new file named `server.cpp` in the root of your project directory.

Add the Code: Type the following C++ code into `server.cpp`. Try to understand what the code is meant to accomplish while typing it in. If you don't understand, try getting Copilot to explain it for you. Or better yet, use Google. Take notes!

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

#define PORT 8080
#define BUFFER_SIZE 1024
```

```
int main() {
    const char* welcome_message = "Hello from IoT Network
Programming Server!";

    int server_fd;
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        std::cerr << "Socket creation failed" << std::endl;
        return 1;
    }

    int opt = 1;
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt))) {
        std::cerr << "setsockopt failed" << std::endl;
        close(server_fd);
        return 1;
    }

    struct 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) {
        std::cerr << "Bind failed" << std::endl;
        close(server_fd);
        return 1;
    }

    if (listen(server_fd, 3) < 0) {  // Allow up to 3 pending
connections
        std::cerr << "Listen failed" << std::endl;
        close(server_fd);
        return 1;
    }

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

    while (true) {
        struct sockaddr_in client_addr;
        socklen_t addrlen = sizeof(client_addr);
```

```
        int client_socket = accept(server_fd, (struct
sockaddr*)&client_addr, &addrlen);
        if (client_socket < 0) {
            std::cerr << "Accept failed" << std::endl;
            continue;
        }

        std::cout << "New client connected" << std::endl;

        send(client_socket, welcome_message,
strlen(welcome_message), 0);
        std::cout << "Welcome message sent" << std::endl;

        close(client_socket);
    }

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

Compile: Open a terminal *inside the dev container* in VS Code (`Terminal > New Terminal`) and run the following command:

```
g++ -o server server.cpp -std=c++11
```

(Note for macOS users compiling locally without the container: You might need `g++ -o server server.cpp -std=c++11 -stdlib=libc++`)

Part 3: Creating the TCP Client

Create the File: In the VS Code explorer, create a new file named `client.cpp`.

Add the Code: Type the following C++ code into `client.cpp`:

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

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

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        std::cerr << "Socket creation error" << std::endl;
        return 1;
    }

    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr) <=
0) {
        std::cerr << "Invalid address or address not supported" <<
std::endl;
        close(sock);
        return 1;
    }

    std::cout << "Connecting to server..." << std::endl;
    if (connect(sock, (struct sockaddr*)&server_addr,
sizeof(server_addr)) < 0) {
        std::cerr << "Connection failed" << std::endl;
        close(sock);
        return 1;
    }

    char buffer[BUFFER_SIZE] = {0};
    int valread = read(sock, buffer, BUFFER_SIZE);

    std::cout << "Message from server: " << buffer << std::endl;

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

Compile: In the same container terminal, run:

```
g++ -o client client.cpp -std=c++11
```

(Note for macOS users compiling locally without the container: You might need `g++ -o client client.cpp -std=c++11 -stdlib=libc++`)

Part 4: Running and Testing

Now, let's run the server and client to see them communicate. You'll need two terminals open *inside the dev container*.

Terminal 1 (Start Server):

```
./server
```

You should see the output: `Server listening on port 8080...` The server will now wait for connections.

Terminal 2 (Start Client):

```
./client
```

You should see output similar to:

```
Connecting to server 127.0.0.1:8080...
Connected successfully!
Message from server: Hello from IoT Network Programming Server!
```

Check Server Output: Look back at Terminal 1 (where the server is running). You should see new lines indicating a client connected and disconnected:

```
New client connected
Welcome message sent (40 bytes)
Client disconnected
```

Part 5: Capturing Network Traffic

Let's repeat the process, but this time capture the network packets exchanged between the client and server.

Stop the Server: Go to Terminal 1 and press `Ctrl+C` to stop the server if it's still running.

Prepare Terminals: You will need three terminals open inside the dev container for this part.

- Terminal 1: For starting the packet capture.
- Terminal 2: For running the server.
- Terminal 3: For running the client.

Terminal 1 (Start Capture): Run the packet capture command. We'll use the helper script provided in the dev container configuration, which uses `tcpdump` underneath. This command tells it to capture traffic on any interface (`-i any`), write it to a file (`-w`), filter for traffic involving port 8080 (`port 8080`), and save the file in the `/workspaces/captures/` directory (which is mapped to the `captures` folder in your project).

```
capture-packets port 8080
```

Alternatively, you can use `tcpdump` directly:

```
sudo tcpdump -i any -w /workspaces/captures/hello_network_$(date
+%Y%m%d-%H%M%S).pcap port 8080
```

You should see output indicating that `tcpdump` is listening.

Terminal 2 (Start Server):

```
./server
```

Wait until you see `Server listening on port 8080...`.

Terminal 3 (Start Client):

```
./client
```

Wait for the client to connect, receive the message, and exit.

Terminal 1 (Stop Capture): Once the client has finished, go back to Terminal 1 (where `capture-packets` or `tcpdump` is running) and press `Ctrl+C` to stop the capture. It will report how many packets were captured and saved.

Part 6: Analyzing the Capture with Wireshark

The captured network traffic is saved in a `.pcap` file inside the `captures` folder of your project directory.

Locate the Capture File: In the VS Code explorer, navigate to the `captures` folder. You should see a `.pcap` file (e.g., `capture-YYYYMMDD-HHMMSS.pcap` or `hello_network_YYYYMMDD-HHMMSS.pcap`).

Download the File: Right-click on the `.pcap` file in the VS Code explorer and select **"Download..."**. Save it to a convenient location on your local machine (like your Desktop or Downloads folder).

Open in Wireshark:

- Launch the Wireshark application on your local computer.
- Go to `File > Open` and select the `.pcap` file you just downloaded.

Analyze the Traffic:

- You will see a list of packets. Since we connected locally (`127.0.0.1`), look for packets with Source and Destination IP addresses of `127.0.0.1`.
- **Filter:** In the filter bar at the top, type `tcp.port == 8080` and press Enter. This will show only the traffic related to our application.

- **Identify Key Packets:** Look for the following sequence:
  - **Three-Way Handshake:**
    - `[SYN]` (Client to Server) - Request to connect.
    - `[SYN, ACK]` (Server to Client) - Acknowledge request, also request to connect back.
    - `[ACK]` (Client to Server) - Acknowledge server's request. Connection established!
  - **Data Transfer:**
    - `[PSH, ACK]` (Server to Client) - This packet likely contains the "Hello from IoT Network..." message. Select this packet and look at the packet details pane (usually at the bottom). Expand the "Transmission Control Protocol" and "Data" sections to see the actual payload.
  - **Connection Teardown (FIN/ACK sequence):**
    - One side sends `[FIN, ACK]` to signal it's done sending.
    - The other side responds with `[ACK]`.
    - The other side sends its own `[FIN, ACK]`.
    - The first side responds with `[ACK]`. Connection closed. (The exact sequence might vary slightly depending on which side closes first).

Troubleshooting

- **`Connection refused` (Client):** The server is likely not running, not listening on the correct port (8080), or not listening on `127.0.0.1` (though `INADDR_ANY` covers this). Double-check the server is running *before* the client.
- **`Address already in use` (Server - Bind failed):** Another process (or a previous instance of your server that didn't shut down cleanly) is already using port 8080. Use `sudo ss -tulnp | grep 8080` to check. Stop the other process or change the `PORT` in both files and recompile.
- **Compilation Errors:** Ensure you are using the correct `g++` command, including `-std=c++11`. Check for typos in your code. The linking errors mentioned earlier might require the `-stdlib=libc++` flag on macOS if compiling locally.
- **Permission Denied (`tcpdump`):** Make sure you run `tcpdump` with `sudo` or use the `capture-packets` script which handles `sudo` internally.

Optional Challenges

1. Modify the client to send a message *to* the server after connecting. Modify the server to read this message and print it.
2. Change the server IP address in the client to connect to a different machine running the server (requires firewall adjustments).
3. Experiment with different ports.