Lab 3: The "Callback" Connection - Bi-Directional Networking

Objective:

In this lab, you will build a system where your programs both initiate and receive connections. This will help you understand:

1. Implementing both TCP client and server logic.
2. How connection information can be exchanged to enable "callback" connections.
3. The flow of communication when roles are reversed.
4. The importance of correct IP addressing and port usage on a local network.

Scenario Overview:

1. **Your Server (`student_server`):** You will write a simple TCP server that starts up and listens for an incoming connection on a port you choose.
2. **Your Client (`student_client`):** You will write a TCP client that connects to a central "Registry Server" (run by the instructor). Your client will tell the Registry Server the IP address and port number where *your server* is listening.
3. **Instructor's Registry Server:** This server will receive your server's information. Then, the Registry Server will attempt to make a *new* TCP connection *back to your server*.
4. **Observation:** Your `student_server` should then accept this incoming connection from the instructor's server and print a confirmation.

Part 1: Implement Your Listening Server (`student_server.cpp`)

Goal: Create a TCP server that listens on a port you choose and waits to accept one connection.

1. **Choose Your Server Port:**
   - Select a unique port number for your server to listen on. A good range is between **10000 and 20000**.
   - **Write down this port number, you'll need it for your client later!** Let's call this `STUDENT_SERVER_PORT`.

2. **Create `student_server.cpp`:**
    - Include necessary headers: `<iostream>`, `<cstring>`, `<unistd.h>`, `<sys/socket.h>`, `<netinet/in.h>`, `<arpa/inet.h>`.
    - Define `STUDENT_SERVER_PORT` with the port number you chose.
3. **`main()` Function Logic:**

    a. **Create Socket:** Use `socket(AF_INET, SOCK_STREAM, 0)`. Remember to check for errors after this call and every subsequent socket call!

    b. **Set `SO_REUSEADDR` Socket Option:** This is good practice to allow your server to restart quickly.

    ~~~~
    ```cpp
    int optval = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &optval,
    sizeof(optval));
    ```
    ~~~~

    c. **Prepare `sockaddr_in` Structure for Your Server:**

    ~~~~
    *   `memset` the structure to zero.
    *   Set `sin_family` to `AF_INET`.
    *   Set `sin_addr.s_addr` to `INADDR_ANY` (this means your
    server will listen on all available network interfaces of your
    machine/container).
    *   Set `sin_port` to `htons(STUDENT_SERVER_PORT)`.
    ~~~~

    d. **Bind:** Call `bind()` to associate your socket with the address and port. **Error check!**

    e. **Listen:** Call `listen()` to enable incoming connection requests. A backlog of `5` is fine. **Error check!**

    f. **Print Listening Message:**

    ~~~~
    ```cpp
    std::cout << "Student server listening on port " <<
    STUDENT_SERVER_PORT << "..." << std::endl;
    ```
    ~~~~

    g. **Accept Connection:**

    ~~~~
    *   Declare a `struct sockaddr_in instructor_addr;` and
    `socklen_t instructor_addr_len = sizeof(instructor_addr);`.
    *   Call `int instructor_conn_fd = accept(server_fd, (struct
    sockaddr *)&instructor_addr, &instructor_addr_len);`. This
    call will block until a connection comes in. **Error check!**
    ~~~~

~~~
*   If `accept()` is successful, print a confirmation:
    ```cpp
    char instructor_ip_str[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &instructor_addr.sin_addr,
instructor_ip_str, INET_ADDRSTRLEN);
    std::cout << "Connection accepted from instructor's server
at "
              << instructor_ip_str << ":" <<
ntohs(instructor_addr.sin_port)
              << std::endl;
    ```
~~~

h. **(Optional) Receive a Message:** If the instructor's server sends a message, you can add a `recv()` call here to read and print it.
i. **Close Sockets:** Close `instructor_conn_fd` and then `server_fd`.

4. **Compile Your Server:**

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

Part 2: Implement Your Reporting Client (`student_client.cpp`)

Goal: Create a TCP client that connects to the instructor's Registry Server and sends it the IP address and port where your `student_server` is listening.

1. **Instructor's Registry Server Details:**
   - Your instructor will provide the IP address (`INSTRUCTOR_REGISTRY_IP`) and port (`INSTRUCTOR_REGISTRY_PORT`) for their central Registry Server.
2. **Create `student_client.cpp`:**
   - Include necessary headers (similar to the server, plus `<string>` and `<vector>` if you use them for IP parsing).
   - Define `INSTRUCTOR_REGISTRY_IP` and `INSTRUCTOR_REGISTRY_PORT` with the values provided by your instructor.
   - Define `STUDENT_SERVER_PORT` with the *same port number you used in your `student_server.cpp`*.
3. **Determine Your Server's Reachable IP Address:**
   - Since we are all on the same local network, you need to find the IP address of the machine (or dev container) where your `student_server` will be running.
   - In a terminal on that machine/container, type:

```
ip addr
```

Look for your primary network interface (e.g., `eth0`, `en0`, `wlan0`). Find the `inet` address listed (e.g., `inet 192.168.1.105/24`). This is the IP address you will use.

- **Store this IP address as a string in your client code.** Let's call it `MY_SERVER_IP_STRING`.

4. **`main()` Function Logic:**

a. **Create Socket:** `socket(AF_INET, SOCK_STREAM, 0)`. **Error check!**
b. **Prepare `sockaddr_in` for Instructor's Registry Server:**

```
*   `memset` the structure to zero.
*   Set `sin_family` to `AF_INET`.
*   Set `sin_port` to `htons(INSTRUCTOR_REGISTRY_PORT)`.
*   Use `inet_pton(AF_INET, INSTRUCTOR_REGISTRY_IP,
&registry_server_addr.sin_addr);` to convert the IP string.
**Error check `inet_pton``'s return value!**
```

c. **Connect:** Call `connect()` to establish a connection to the instructor's Registry Server. **Error check!**
d. **Prepare Your Information Message:**

~~~
*   Create a string that combines your server's IP address and
port, separated by a colon.
*   Example format: `"192.168.1.105:15000"`
```cpp
std::string my_server_ip = MY_SERVER_IP_STRING; // The IP you
found in step 3
int my_server_port = STUDENT_SERVER_PORT;
std::string message_to_send = my_server_ip + ":" +
std::to_string(my_server_port);
~~~
```

e. **Send Message:**

```
*   Call `send(client_sock, message_to_send.c_str(),
message_to_send.length(), 0);`. **Error check!**
*   Print a confirmation:
    ```cpp
    std::cout << "Sent to Registry Server: " <<
message_to_send << std::endl;
    ```
```

f. **(Optional) Receive Confirmation:** If the instructor's server sends a confirmation back, you can add a `recv()` call here.
```

g. **Close Socket:** `close(client_sock)`.

5. **Compile Your Client:**

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

Part 3: Testing Procedure

1. **Instructor:** Ensure your Registry Server is running and ready to receive connections and make callbacks.
2. **You (Student) - Terminal 1:**
   - Navigate to your project directory.
   - Run your compiled server:

     ```
     ./student_server
     ```
   - It should print: `Student server listening on port <YOUR_STUDENT_SERVER_PORT>...`
3. **You (Student) - Terminal 2 (Ensure `MY_SERVER_IP_STRING` is correct in `student_client.cpp`):**
   - Navigate to your project directory.
   - Run your compiled client:

     ```
     ./student_client
     ```
   - It should connect to the instructor's Registry Server and print the message it sent (e.g., `Sent to Registry Server: 192.168.1.105:15000`).
4. **Observe Terminal 1 (`student_server`):**
   - Shortly after your client sends its information, your `student_server` should print the message indicating it has accepted a connection from the instructor's server (e.g., `Connection accepted from instructor's server at <INSTRUCTOR_IP>:<SOME_PORT>`).
5. **Troubleshooting:**
   - **Client can't connect to Registry Server:** Double-check instructor's IP/Port. Is the Registry Server running? Any firewalls blocking your *outgoing* connection (less common)?
   - **Student Server doesn't receive callback:**
     - Did your client send the correct IP address for your server machine/container?
     - Is your `student_server` actually listening on the port you told the client to send?

  - Is there a firewall on your machine/container blocking *incoming* connections to `STUDENT_SERVER_PORT`? (You might need to add an exception).
  - Did the instructor's server correctly parse your IP/Port string?

Part 4: `tcpdump` Investigation (Optional, but Highly Recommended)

If you want to see the network traffic:

1. **Before running `./student_server` (or in a separate terminal):**
   Start `tcpdump` to watch for traffic on your server's port:
   ```
   sudo tcpdump -i any -nnX port <YOUR_STUDENT_SERVER_PORT>
   ```
   When the instructor's server connects back, you should see the TCP handshake and any data.
2. **Before running `./student_client` (or in a separate terminal):**
   Start `tcpdump` to watch for traffic to/from the instructor's registry server:
   ```
   sudo tcpdump -i any -nnX host <INSTRUCTOR_REGISTRY_IP> and
   port <INSTRUCTOR_REGISTRY_PORT>
   ```
   You'll see your client connect, send data, and then the connection close.

Discussion Points:

- What was the sequence of connections in this lab?
- Why was it necessary for your client to tell the Registry Server where your server was listening?
- If your `student_server` was behind a home router with NAT (Network Address Translation), what IP address would your client have needed to send for the instructor's server to reach it? What else would need to be configured on your home router?
- Consider the security implications of a program that accepts instructions on where to connect back.

Good luck! This lab requires careful attention to detail with IP addresses and ports.