CSCN72020 – Lab 3-W23

Your First Client/Server Program

Overview

The purpose of this lab is to allow you to put together a basic Client/Server application (your first one) using Berkley Sockets with a virtual network configured using NAT. This document will walk you through step-by-step the setup and code you "always" need to create a client/server application. It will work on a TCP/IP protocol and transmit a single "Hello World" string between the two applications over two different networks using a NAT port forwarding rule.

> *NOTE: This document has been written to create your Client for Windows and your Server for Ubuntu Linux. If you are working on a MAC or Linux/Unix based OS for the client, these instructions will only get you part of the way there. Have a look at the differences in the libraries for the server application code.*

PART 1: Setting up your Virtual Machine

To begin you will need to create a Virtual Machine running Ubuntu Linux using Virtual Box.

If you have not done so already, download a copy of the Oracle VM VirtualBox Manager Software and install it on your local computer. You can find it by searching the internet for "Download Oracle VirtualBox".

Your Virtual Machine needs to run Ubuntu Linux. You can download an *.ISO file for Ubuntu by searching the internet for "download Ubuntu ISO file".

Using VirtualBox, create a new Virtual Machine called "Ubuntu Server" and install the Ubuntu ISO you just downloaded. Make sure once you have setup the Virtual Machine you do a quick update of the OS to make sure you are running the latest and greatest Ubuntu Linux.

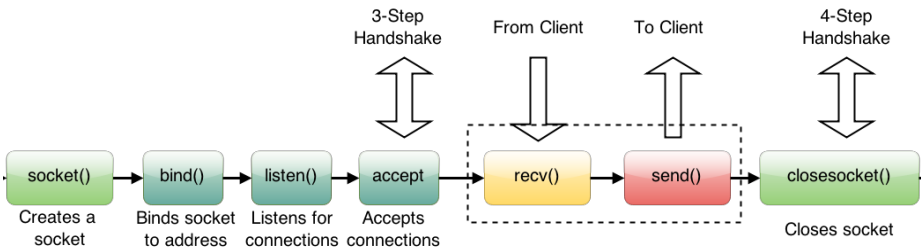
> *NOTE: It is assumed you have the knowledge from previous courses to install and setup a Virtual Machine. If you are having any problems, please touch base with a friend or the Professor for help.*

Run your new Virtual Machine and install the following software:

- Net-tools
- G++
- Visual Studio Code (or any other editor you prefer)

PART 2: Writing the Server

As we discussed, starting up any type of data communications requires you to follow a set of operations. Order of operations is critical when coding up a client/server architecture. Here is the order we walked through:



Let's create the server together. First thing you need to do is create a Server.cpp file in your Visual Studio Code (or other editor) on the Virtual Machine. Once created add the following headers to your project file (do not use any header files *.h).

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

Now let's go through each block one-by-one

Socket ()

Next you need to create a socket that we can configure as a "listening" socket. I.E. the front door into your application. In order to do this you need to know the Transport Layer service you are planning to use. In this case TCP (a streaming service). Create your socket using the following code:

```
int ServerSocket;
ServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (ServerSocket == -1) {
          cout << "ERROR:  Failed to create ServerSocket" << std::endl;
          return 0;
}
```

Bind()

As discussed in class, in order for data packets to reach your application, the Operating System must know the valid IP Addresses (in our case IPv4) and the port number. This information goes in the TCP and IP packet headers. Remember, we don't create the headers, but we must provide critical information for them to be configured by the library. The following code will bind (or register) your server application to accept connections from any IP address (INADDER_ANY) and link your application to a pre-selected port number.

```
sockaddr_in SvrAddr;
SvrAddr.sin_family = AF_INET;
SvrAddr.sin_addr.s_addr = INADDR_ANY;
SvrAddr.sin_port = htons(27000);
if (bind(ServerSocket, (struct sockaddr *)&SvrAddr, sizeof(SvrAddr)) == -1)
{
          close(ServerSocket);
          cout << "ERROR:  Failed to bind ServerSocket" << std::endl;
          return 0;
}
```

Listen()

This block is only required by TCP. Why? TCP is a connection oriented protocol right? That means the server must be "listening" for an incoming connection – the SYN request to start the three-way handshake. So we must configure the socket we have created and bound to listen for incoming requests.

```
if (listen(ServerSocket, 1) == -1) {
        close(ServerSocket);
        cout << "ERROR:  listen failed to configure ServerSocket" << std::endl;
        return 0;
}
```

Accept()

This is another TCP only block. It's part of the three-way handshake process. When a SYN request comes into the server application, the accept will complete the handshaking process and provide the connection with a unique socket and port number (this is the return of the accept call). Note, accept is a blocking operation. Once your server calls accept the server application will block waiting for an interrupt from the OS stating it has received a packet destined for your application.

```
int ConnectionSocket;
ConnectionSocket = -1;
if ((ConnectionSocket = accept(ServerSocket, NULL, NULL)) == -1) {
        close(ServerSocket);
        return 0;
}
```

Send/Receive Data

Once you have made your connection it's time to send and receive data. This is simply done by calling the Send() and Receive() functions. I'm going to leave this part to you to figure out. Here are a couple of hints:

1. You need to know the socket being used
2. You need to know the address of where your data is located
3. You need to know the size (in bytes) of the data being transmitted/received

The function declarations look as follows:

```
recv(<socket to use>, <address to data>, <size of data>, 0);

send((<socket to use>, <address to data>, <size of data>, 0);
```

Write some code that can be used to receive a string with "Hello World" from the client, and print the received string to standard out.

Cleanup – Close the Sockets

It is always important to close out and cleanup any socket communication within your programs. That's because without the close, the four-way handshake (for TCP connections) will not be completed. It could also unregisters the port number with the operating system. Leaving it open for somebody else to use (or even better, your next run of your application).

```
close(ConnectionSocket);
close(ServerSocket);
```

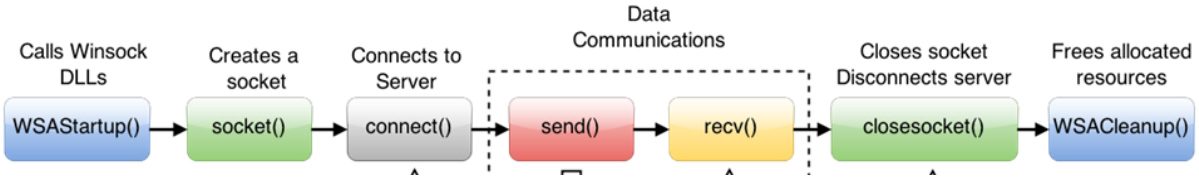
Now compile and run your Server application. If all works, your code should compile and the server should run waiting for a SYN request from a client.

PART 3: Writing the Client

At this point your server should be complete, compiling (with no errors or warnings) and running – waiting for a SYN request from the client. It's now time to create the client

Create a new project/solution in Visual Studio. Add a new source file called Client.cpp. Again, don't use any header files (*.h).

Just like servers, clients need to follow a set of operations. If you look at the client diagram below, you will notice it's very similar to the server, with the exception of the Connect() box and the WSACleanup. Since the Client will be running on Windows, we need some extra steps to enable and configure the Windows Socket Library.



WSAStartup()

This block is responsible for starting up and configuring the WinSock Dynamically Linked Library, which is connected to your project by using the #pragma statement from the previous step. To start this up we need to declare a WSADATA object and provide the address of that object into the library's startup function as follows:

```
WSADATA wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
          cout << "ERROR:  Failed to start WSA" << std::endl;
          return 0;
```

Socket ()

Next you need to create a socket that we can use to make "the call" to the Server. In order to do this you need to know the Transport Layer service you are planning to use. In this case TCP (a streaming service). Create your socket using the following code:

```
SOCKET ClientSocket;
ClientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (ClientSocket == INVALID_SOCKET) {
          WSACleanup();
          cout << "ERROR:  Failed to create ServerSocket" << std::endl;
          return 0;
}
```

Connect()

This is different. On the server side you needed to bind and configure a socket for listening. On the client side, you need to start the three-way handshake process to create a connection. To do that we use the connect function as follows:

```
sockaddr_in SvrAddr;
SvrAddr.sin_family = AF_INET;
SvrAddr.sin_port = htons(27500);
SvrAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
if ((connect(ClientSocket, (struct sockaddr *)&SvrAddr, sizeof(SvrAddr))) == SOCKET_ERROR) {
          closesocket(ClientSocket);
          WSACleanup();
          cout << "ERROR:  Connection attempted failed" << std::endl;
          return 0;
}
```

Before you call connect, you will notice we have to create an instance of a sockaddr_in object. This object contains critical information needed by the library to setup the TCP/IP headers correct. More specifically:

- The family, which specifies the protocol that will be used
- The port number the server application is bound to
- The IP address (IPV4) the server application is located on. In this case, because it's on a different network, we will send the request to the NAT by using localhost

Send/Receive Data

Once you have made your connection it's time to send and receive data. This is simply done by calling the Send() and Receive() functions. I'm going to leave this part to you to figure out. Here are a couple of hints:

1. You need to know the socket being used
2. You need to know the address of where your data is located
3. You need to know the size (in bytes) of the data being transmitted/received

The function declarations look as follows:

```
recv(<socket to use>, <address to data>, <size of data>, 0);
send((<socket to use>, <address to data>, <size of data>, 0);
```

Write some code that will send the string "Hello World" to the server.

Cleanup – Closesocket and Wsacleanup

Don't forget to cleanup your sockets and winsock libraries. What calls do you think you need for the client to complete this activity? Write the source code.

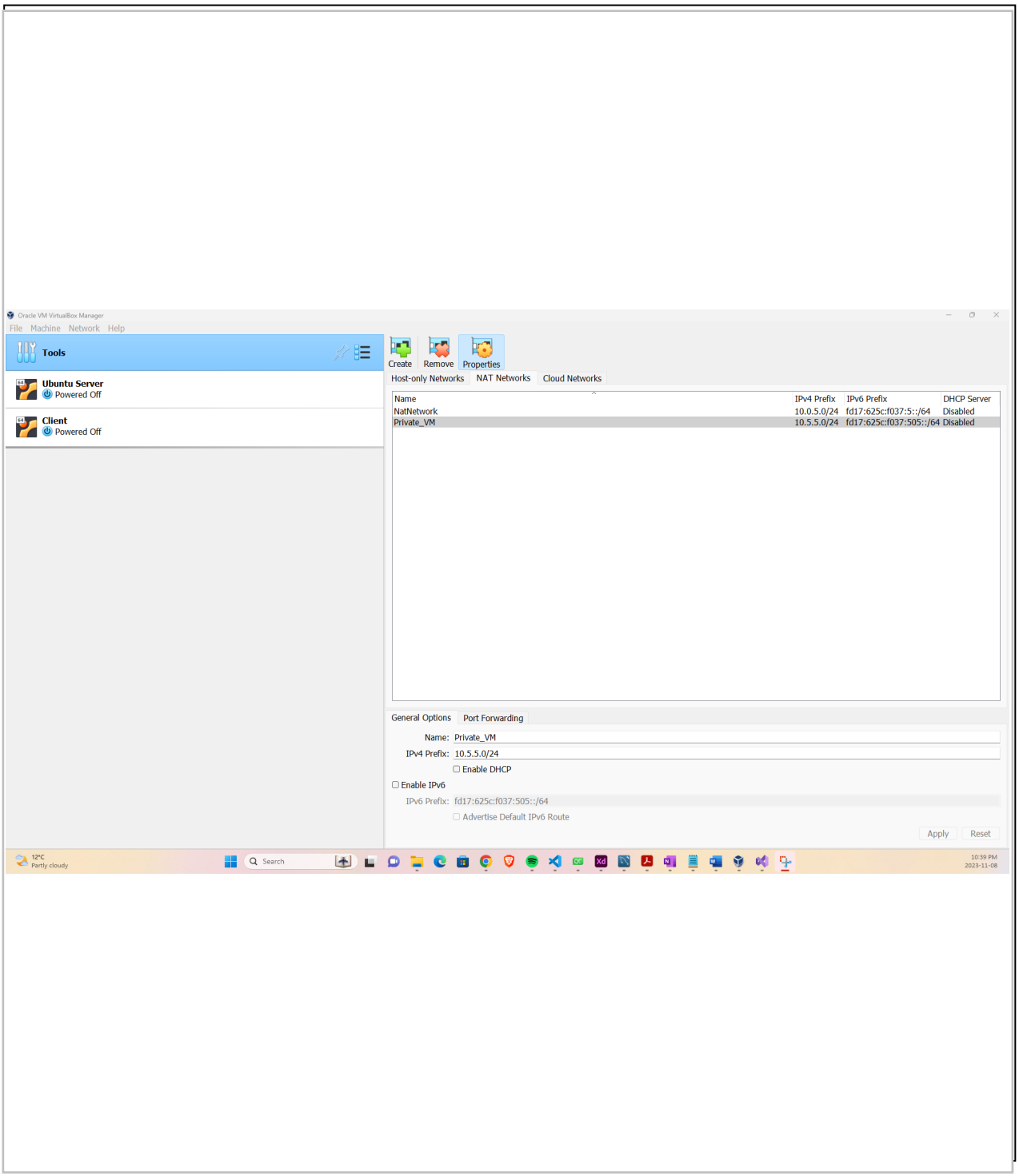
At this point you should be able to compile your code with no errors and no warnings.

PART4 Setting up the Virtual Network

Make sure you have powered off any virtual machines before completing the network setup. Using the processes taught in class, setup a virtual private network with NAT and port_forwarding rules.

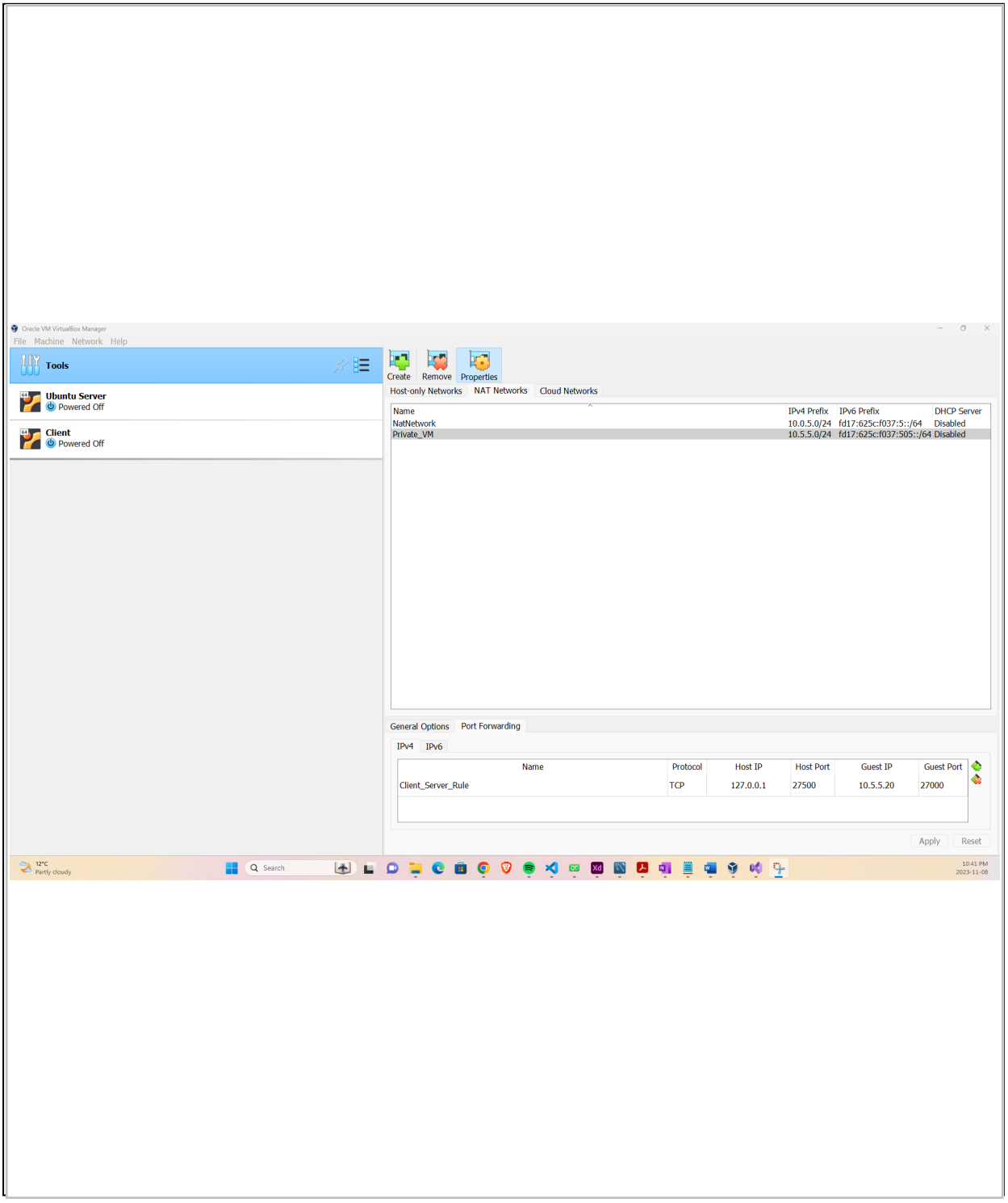
Step #1: Setup a NAT Network

Start the Oracle VirtualBox Manager software and under Tools→ NAT Networks, add a network called **"Private_VM"** and set network IPv4 Prefix to 10.5.5.0/24. Take a screenshot of the NAT Networks tab showing your "General Options" setup for your NAT Network. Upload the screenshot into the box below:



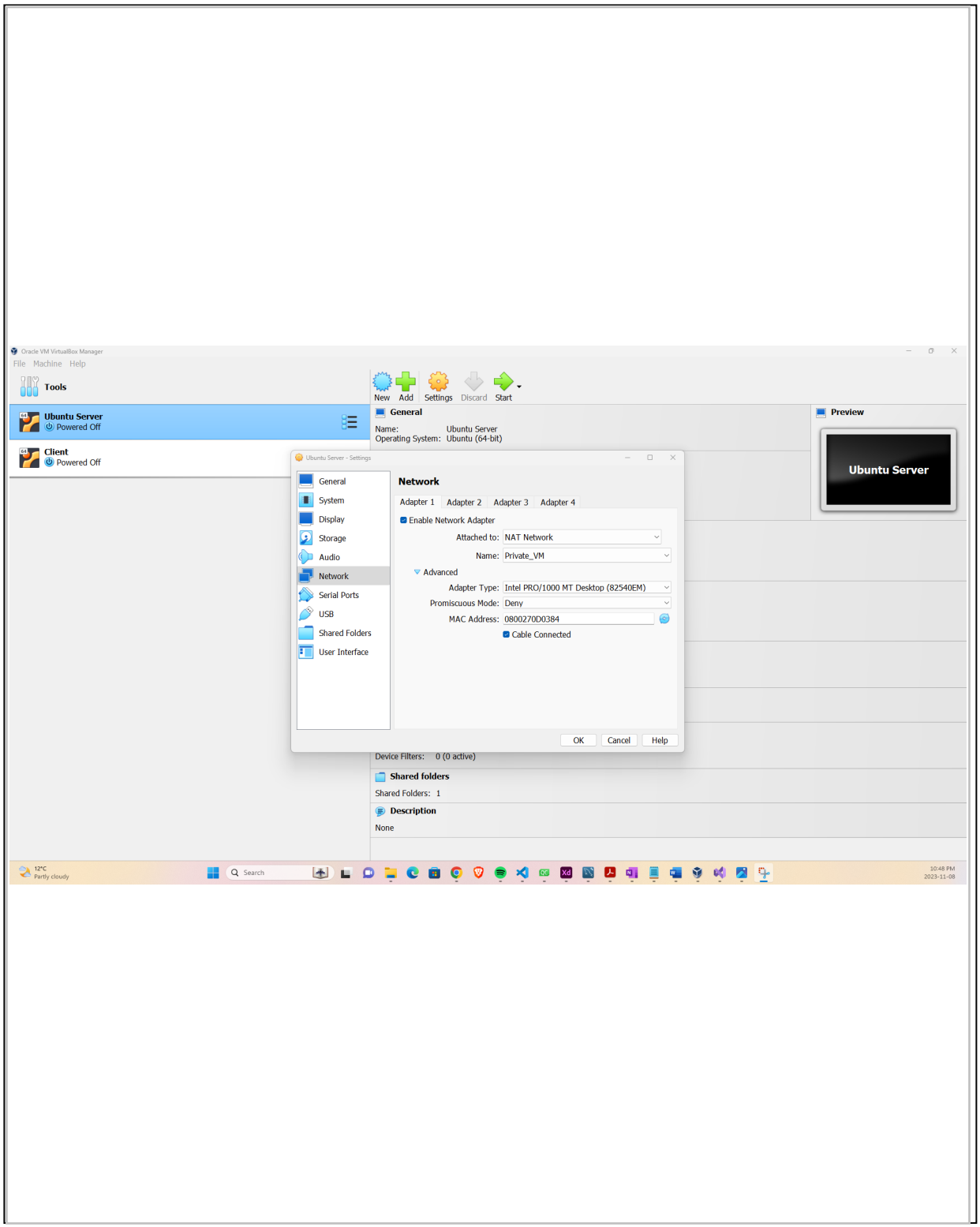
Step #2: Setup a Port Forwarding Rule

Next you need to configure a Port Forwarding rule on the NAT to allow the IP address differences between the bridge network and the NAT network to occur. Look at your client/server code. Pay close attention to the Transport Layer information. Setup a NAT rule that will allow TCP/IP connections and data to flow from the localhost on your Windows PC to the Server application on your Linux VM. Take a screenshot of the NAT Networks tab showing your "Port Forwarding" setup. Upload the screenshot into the box below:



Step #3: Connect your Virtual Machine

Now that you have configured your NAT Network in the VirtualBox Manager, it is now time to connect your Ubuntu Linux Virtual Machine to your private NAT Network. Under settings on your virtual machine, select the Network options and attach your "Adapter 1" to the NAT Network **"Private_VM"**. Take a screenshot of your "Adapter 1" settings. Upload the screenshot into the box blow:

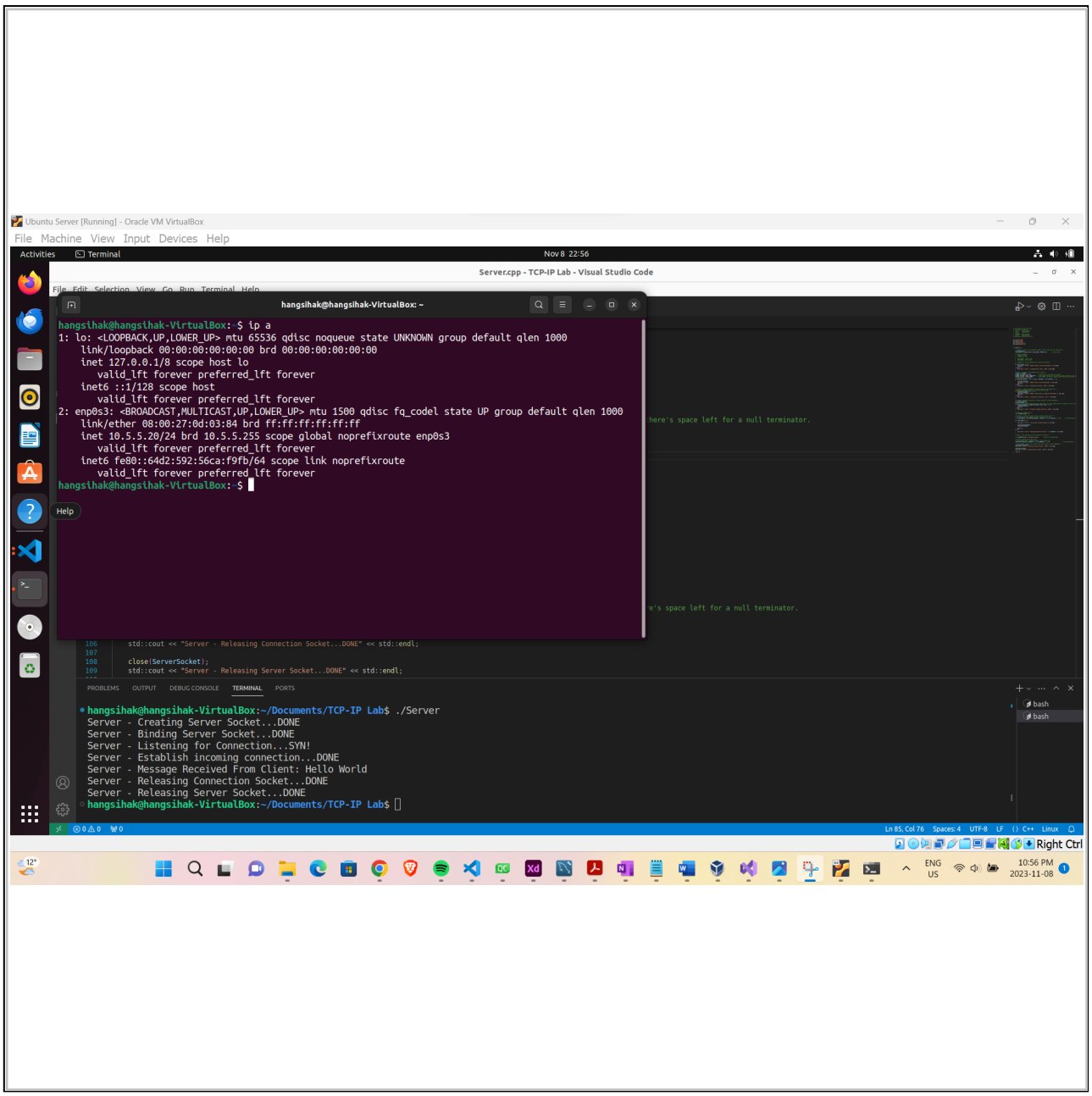


Step #4: Setup a Static IP on your Virtual Machine

Power on your Virtual Machine and log in. Once logged in, click on the upper right corner of the screen and select "Wired Connected → Wired Settings" from the drop down menu. This will open a dialog box showing the connection information. Open up the "settings" for the Wired connection and select the "IPv4 " tab. On this tab configure a manual IP address as follows:

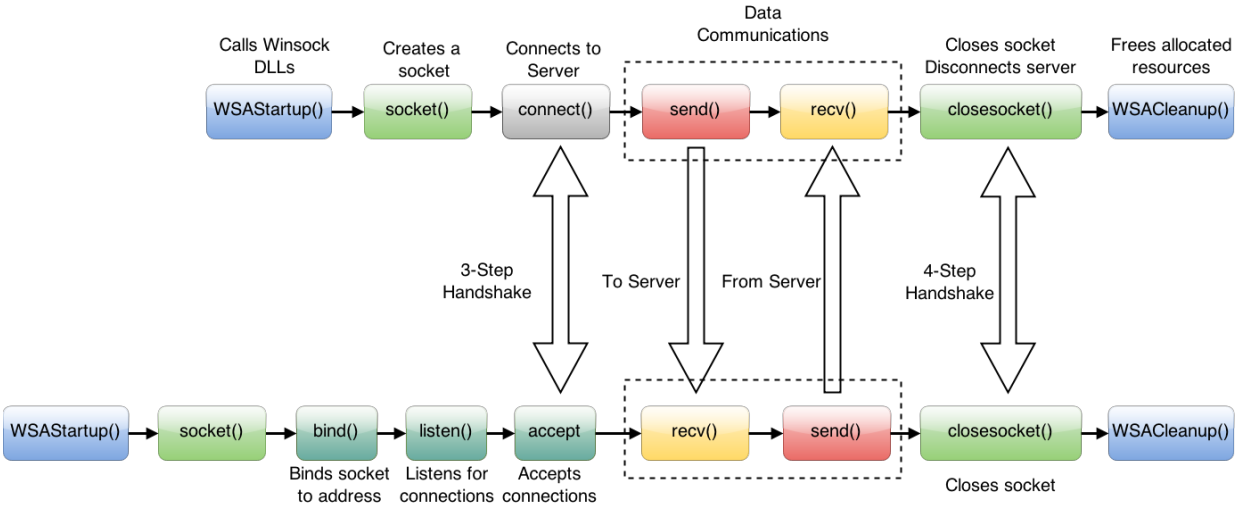
- Select Manual
- IP Address: 10.5.5.20
- Netmask: 255.255.255.0
- Gateway: Leave it blank (this network does not have a default gateway)

Click on "Apply" to apply the changes and open a Terminal Window and type in the command "ip a". Take a screenshot showing the terminal window with the output of the command. Upload the screenshot into the box below:



PART 5 Running the Applications

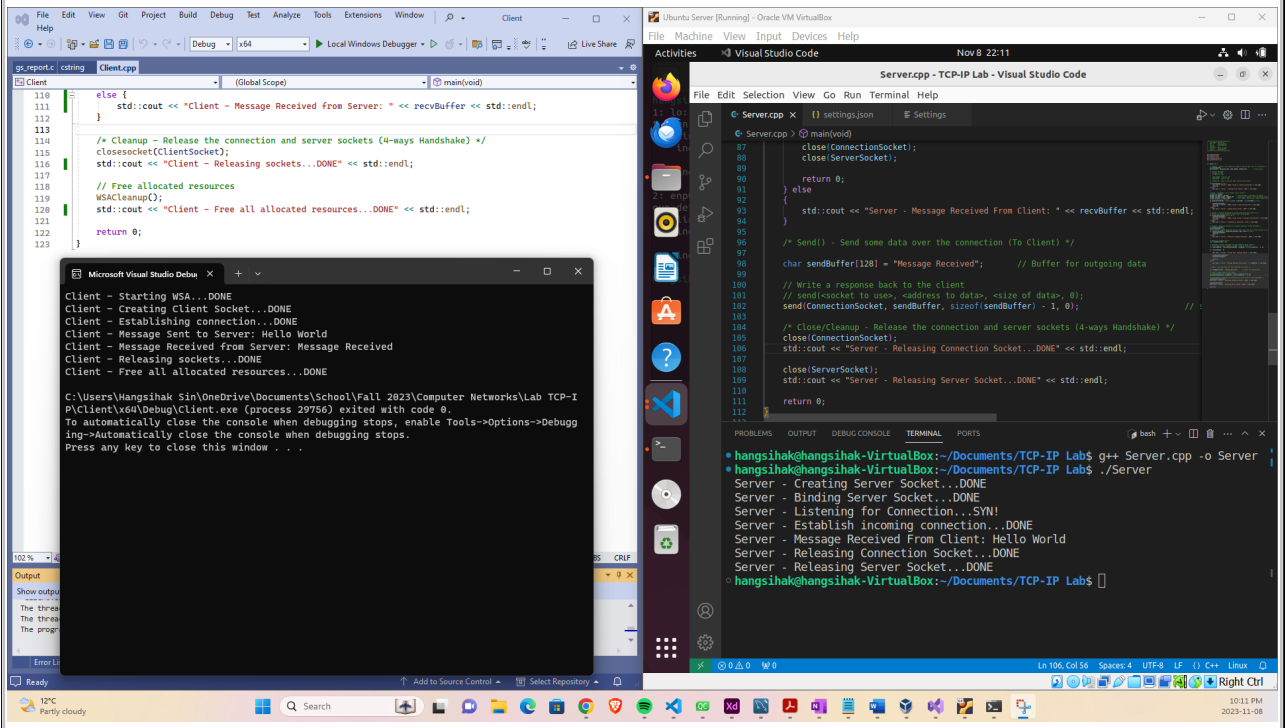
You should now have a Server application (running on a Ubuntu Virtual Machine) and a Client application (running on your local Windows Machine). It's that time. Let's run the applications and see what happens. As I have said before, order of operations is critical when dealing with data communications. Which application do you think you should run first? Here's a hint.



If you said Client, you should probably think about some extra tutoring for this course. 😊 The Server MUST be run first. Otherwise, the client will fail to create a connection because the SYN request will timeout.

Run your server. Run your client. Did your server receive and print "Hello World"? If not, why? How would you start to debug the problem?

Take a screenshot with your VM showing the execution of your server application receiving and printing "Hello World" in a terminal window. Upload the screenshot into the box below.


Client.cpp
Client - Starting WSA...DONE
Client - Creating Client Socket...DONE
Client - Establishing connection...DONE
Client - Message Sent to Server: Hello World
Client - Message Received from Server: Message Received
Client - Releasing sockets...DONE
Client - Free all allocated resources...DONE
C:\Users\Hangsihak Sin\OneDrive\Documents\School\Fall 2023\Computer Networks\Lab TCP-IP\Client\x64\Debug\Client.exe (process 29756) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
Server.cpp - TCP-IP Lab - Visual Studio Code
hangsihak@hangsihak-VirtualBox:~/Documents/TCP-IP Lab$ g++ Server.cpp -o Server
hangsihak@hangsihak-VirtualBox:~/Documents/TCP-IP Lab$ ./Server
Server - Creating Server Socket...DONE
Server - Binding Server Socket...DONE
Server - Listening for Connection...SYN!
Server - Establish incoming connection...DONE
Server - Message Received From Client: Hello World
Server - Releasing Connection Socket...DONE
Server - Releasing Server Socket...DONE
hangsihak@hangsihak-VirtualBox:~/Documents/TCP-IP Lab$

Rubric

See eConestoga

What to Hand In

Once you have completed your activity upload the following files ***INDIVIDUALLY*** to eConestoga:

- Copy of this PDF with a screenshot
- Copy of your Client.CPP file
- Copy of your Server.cpp file