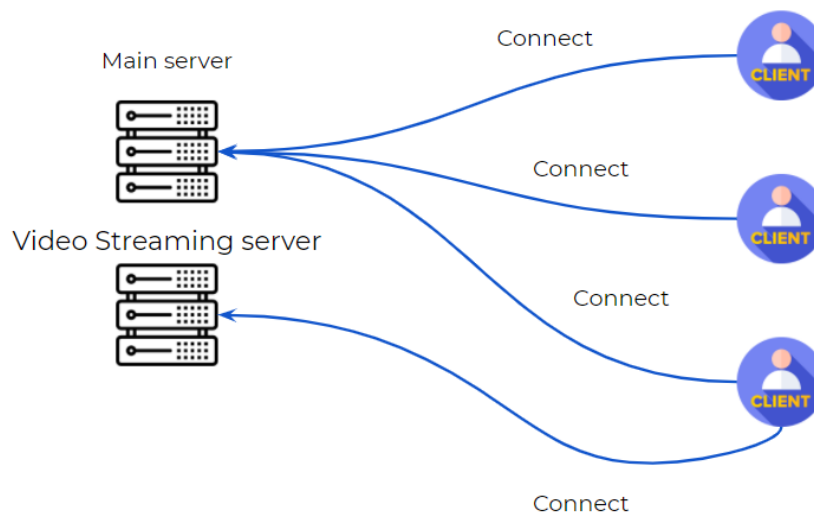


Network and Distributed Computing

Final Project

I. Chat Application

The first part of the project asks us to create a chat application with video streaming functionality and many more using socket programming. Socket programming shows how to use socket APIs to establish communication links between remote and local processes. This project is fully implemented using python programming language and a couple of important modules. The diagram below will show how the chat application works.



The chat application have the following functionalities:

- Messaging with users connected to the same server.
- Sending files and downloading.
- Video streaming from one client to others.

The application has two servers where one is used for normal chatting and file downloading, whereas the other server is used for video streaming only. Lets see the implementation details for the main chatting server.

- 1) The modules used on the main server are **socket**, **threading**, **string**, **re** and **os**. The **socket** module is used for server socket and listening on a specific port, the **threading**

module is used to enable the server to handle multiple requests from different end points. This will allow our program not to freeze when getting many requests at a time and keeps concurrency, the **string** module is used to access basic string functions and define a data type for function arguments, the **re** module is used to compile and compare regular expression this is mainly used when files are sent through the chat, the last module is **os** gives us the functionality to traverse path and different operating system operations which is used while downloading files.

```
# imports
import socket
import threading
import string
import re
import os
```

- 2) Once the modules are imported then define program constants like the IP of the server, the port and an empty list which will hold all users which are currently connected to the server. To get the IP we can use the two **socket** module functions **gethostbyname()** and **gethostname()** .

```
# constants
IP = socket.gethostbyname(socket.gethostname())
PORT = 4433
users = []
```

- 3) Let's talk about the functions: the code has 8 functions, 7 helper functions and 1 main function. `check_if_message_contains_file`, `read_file`, `write_data`, `receive_messages`, `send_msg`, `broadcast_msg`, `handle_clients` and `main`.

- a) The `check_if_message_contains_file` function checks if a file path is found inside the chat message. This helps us to identify whether a message is a file path or just an ordinary message. If it's a file we will try to save it otherwise just display the message to the other users.

```
def check_if_message_contains_file(message:
string):
    if re.findall(r'(\./.*?\.[\w:]+)', string):
```

```

        return True
    return False

```

- b) The function `read_file` reads a file as a binary file and then returns the data.

```

def read_file(file):
    with open(file, 'rb') as f:
        data = f.read()
    return data

```

- c) The function `write_data` will write data returned from `read_file` function into a given filename.

```

def write_data(data, filename):
    path = os.path.join(os.getcwd(), 'downloads',
filename)
    with open(path, 'wb+') as f:
        f.write(data)

```

- d) The `receive_messages` function keeps listening for incoming messages, and calls the `check_if_message_contains_file` function to check if a message is file path or a normal message. If it's a file, get the file name then write the data into another file with the same file name and save it into the downloads directory. Otherwise just broadcast the message to the other users.

```

def receive_messages(client: socket.socket,
username: string):
    while True:
        message = client.recv(2048).decode('utf-8')
        if message != '':
            if os.path.isfile(message):
                filename = os.path.basename(message)
                msg = username + '>' + filename
                data = read_file(message)
                write_data(data, filename)

```

```

        broadcast_msg(msg)
    else:
        msg = username + '>' + message
        broadcast_msg(msg)
    else:
        print("[!] message is empty")

```

- e) `send_msg` receives two arguments: client socket and message which is a string and then sends the string message to the provided client.

```

def send_msg(client: socket.socket, message:
string):
    client.sendall(message.encode())

```

- f) `broadcast_msg` takes a string message as an argument and loops through the current online users and then calls the `send_msg` function which sends the message to each user in the list.

```

def broadcast_msg(message: string):
    for user in users:
        send_msg(user[1], message)

```

- g) `handle_clients` takes client socket as an argument and keeps looping and receiving messages. The first time it will accept the username input then prompts the users they have joined the chat. Then using threading, keep running the function to accept incoming messages.

```

def handle_clients(client: socket.socket):
    while True:
        username = client.recv(2048).decode('utf-8')
        if username != '':
            users.append((username, client))
            prompt_message = "admin>" + f"{username}
added to the chat"
            broadcast_msg(prompt_message)
            break

```

```

        else:
            print("[!] no user joined the chat yet")
            threading.Thread(target=receive_messages,
                             args=(client, username,)).start()

```

- h) The `main` function creates the server socket object with normal IPv4 address and TCP connection setting. It will keep looping to accept incoming clients who want to join the chat and using the threading module call the function `handle_clients` to keep things concurrent.

```

def main():
    server = socket.socket(socket.AF_INET,
                           socket.SOCK_STREAM)

    try:
        server.bind((IP, PORT))
        print(f"[/] server is running {IP}:{PORT}")
    except ConnectionError as e:
        print("[!] unable to make connection")

    server.listen(5)

    while True:
        client, address = server.accept()
        print(f"connection made with {address[0]}:{address[1]}")
        threading.Thread(target=handle_clients,
                         args=(client,)).start()

```

The video streaming server has similar functionality with no functions at all. There are three different modules used here compared with the main chatting server. The **open-cv** module which is used for image and video processing, the **pickle** module which is used for serializing and deserializing python object structures and the **struct** module which is used to convert native python data types like string and integers into a string of bytes and vice versa.

- 1) First importing the modules is a must thing to do so that the whole program can function as expected.

```
import socket
import cv2
import pickle
import struct
```

- 2) Define the server socket object with normal IPv4 IP address and TCP connection, define the IP address and port constants as well.

```
server = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

IP = socket.gethostbyname(socket.gethostname())
PORT = 9999
```

- 3) Bind the server IP and keep listening for incoming connections.

```
server.bind((IP, PORT))
server.listen(5)

print(f"[/]server is listening on {IP}:{PORT}")
```

- 4) Finally keep looping for any client who wants to make the video stream. Once there is a client access the webcam to start recording. We will get the images and frames from the video using the **read()** function then convert the frames into bytes using the pickle module. We will then send the converted bytes to all other clients through the server.

```
while True:
    client, address = server.accept()
    if client:
        video = cv2.VideoCapture(0)
        while video.isOpened():
            image, frame = video.read()
            a = pickle.dumps(frame)
            message = struct.pack("Q", len(a)) + a
            client.sendall(message)
            cv2.imshow('Ongoing Video Call', frame)
```

```

key = cv2.waitKey(1) & 0xFF
if key == ord('q'):
    client.close()

```

The Client side implementation is quite different from the Server side implementation. For the client we don't actually listen for incoming connections we just simply connect to the server and accept messages sent from the main server. The implementation is given below:

- 1) As usual we will import the modules that we will work with: pickle, socket, struct, threading, string, tkinter and cv2 are the modules used on the client side the only two new modules here are the pickle module which is used for implementing binary protocol when serializing and deserializing python objects and tkinter which is a python module used for Desktop App development were in this case we will use it to create our client side UI.

```

import pickle
import socket
import struct
import threading
import string
import tkinter as tk
import tkinter.scrolledtext as scr_text
import tkinter.messagebox as msg_box
from tkinter import *
from tkinter import filedialog
import cv2

```

- 2) Then define our constants like the server IP, PORT and font for our UI.

```

# constants
IP = '192.168.118.1'
PORT = 4433
font = ("Terminal", 13)

```

- 3) Create our client socket with a normal IPv4 address and TCP connection.

```

# client socket object

```



```
client = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
```

4) The client side implementation have 7 functions

a) `show_message` is used to show messages for the user using the tkinter module functionalities in a message box.

```
def show_message(message: string):
    m_box.config(state=tk.NORMAL)
    m_box.insert(tk.END, message + '\n')
    m_box.config(state=tk.DISABLED)
```

b) `file_upload_action` opens a file uploading window for the user on a button click so that users or clients can choose the files they want to send over the server.

```
def file_upload_action(event=None):
    file = filedialog.askopenfilename(title='select
file', filetypes=(("Text files", "*.txt"), ("Tll
files", "*..*")))
    m_textbox.insert(0, file)
```

c) `video_call_request` creates a new socket to connect to the video streaming server with a normal IPv4 address. We will use the struct module to pack and unpack incoming data from the video streaming, receive 4K video data and use the cv2 module to show the images and we will keep doing this until the streaming is ended.

```
def video_call_request():
    video_client = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
    port = 9999
    video_client.connect((IP, port))

    data = b""
    payload_size = struct.calcsize("Q")
```

```

while True:
    while len(data) < payload_size:
        packet = video_client.recv(4 * 1024)
        if not packet: break
        data += packet
    packed_msg_size = data[:payload_size]
    data = data[payload_size:]
    msg_size = struct.unpack("Q",
packed_msg_size)[0]
    while len(data) < msg_size:
        data += video_client.recv(4 * 1024)
    frame_data = data[:msg_size]
    data = data[msg_size:]
    frame = pickle.loads(frame_data)
    cv2.imshow('VideoCall', frame)
    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
        break
    if cv2.getWindowProperty('VideoCall', 4) <
1:
        break
    video_client.close()

```

- d) `connect` connects to the main chatting server, asks the user for a username and once the username is sent then shows the user they have joined the chat room and keep listening for any incoming connections.

```

def connect():
    try:
        client.connect((IP, PORT))
        print("[/] connection success")
        show_message("admin$~ connection success")
    
```

```

except ConnectionError as e:
    msg_box.showerror("error", "unable to make
connection")
    username = u_textbox.get()
    if username != '':
        client.sendall(username.encode())
    else:
        msg_box.showerror("error", "username is
empty")

threading.Thread(target=listen_for_messages_from_se
rver, args=(client,)).start()
    u_textbox.config(state=tk.DISABLED)
    u_button.config(state=tk.DISABLED)

```

- e) `send_message` gets the message input from the message input box and sends it to the server so that it can be broadcasted to all other users.

```

def send_message():
    message = m_textbox.get()
    if message != '':
        client.sendall(message.encode())
        m_textbox.delete(0, len(message))
    else:
        msg_box.showerror("error", "message is
empty")

```

- f) `listen_for_messages_from_server` listens for messages that are sent from the server, this usually means messages that are sent from other users on the chat room. This function receives a message and calls the `show_message` function to display the message.

```

def listen_for_messages_from_server(i_client:
socket.socket):
    while True:
        message =
i_client.recv(2048).decode('utf-8')
        if message != '':
            username = message.split('>')[0]
            content = message.split('>')[1]
            show_message(f"{username}$~ {content}")
        else:
            msg_box.showerror("error", "no message
received")

```

- g) The `main` function just calls **mainloop()** function which keeps the UI running until it's closed by the user.

```

# main function
def main():
    root.mainloop()

```

- 5) One final implementation is the UI, the UI implementation doesn't really have a lot of logic it's just calling tkinter functions and passing in parameters.

```

# UI setup
root = tk.Tk()
root.geometry("800x600")
root.title("NPU internal chat application")
logo = PhotoImage(file="./assets/img/logo.png")
root.iconphoto(False, logo)
root.resizable(False, False)
root.grid_rowconfigure(0, weight=1)
root.grid_rowconfigure(1, weight=4)

```

```

root.grid_rowconfigure(2, weight=1)

# Frames
t_frame = tk.Frame(root, width=600, height=100,
bg="#464EB8")
t_frame.grid(row=0, column=0, sticky=tk.NSEW)
m_frame = tk.Frame(root, width=600, height=400,
bg="#0f1a29")
m_frame.grid(row=1, column=0, sticky=tk.NSEW)
b_frame = tk.Frame(root, width=600, height=100,
bg="#464EB8")
b_frame.grid(row=2, column=0, sticky=tk.NSEW)

# components
u_label = tk.Label(t_frame, text="Username", font=font,
bg="#464EB8", fg="white")
u_label.pack(side=tk.LEFT, padx=10)
u_textbox = tk.Entry(t_frame, font=font, bg="white",
fg="black", width=23)
u_textbox.pack(side=tk.LEFT)
u_button = tk.Button(t_frame, text="Join Chat",
font=font, bg="#0f2642", fg="white", command=connect)
u_button.pack(side=tk.LEFT, padx=25)

m_textbox = tk.Entry(b_frame, font=font, bg="white",
fg="black", width=38)
m_textbox.pack(side=tk.LEFT, padx=10)
m_button = tk.Button(b_frame, text="Send", font=font,
bg="#0f2642", fg="white", command=send_message)
m_button.pack(side=tk.LEFT, padx=10)

```

```

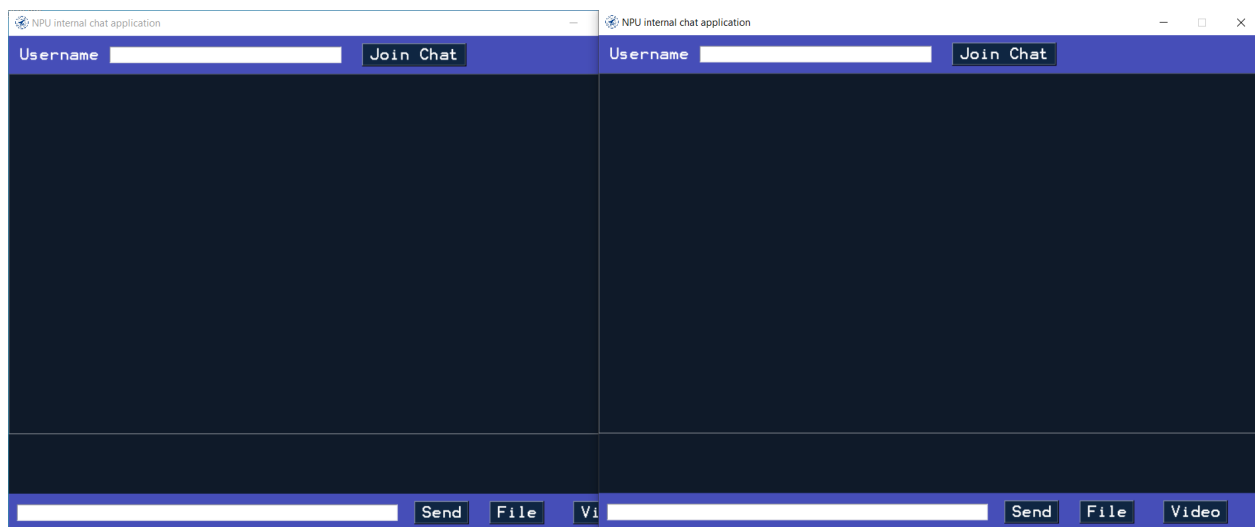
f_button = tk.Button(b_frame, text="File", font=font,
bg="#0f2642", fg="white", command=file_upload_action)
f_button.pack(side=tk.LEFT, padx=15)

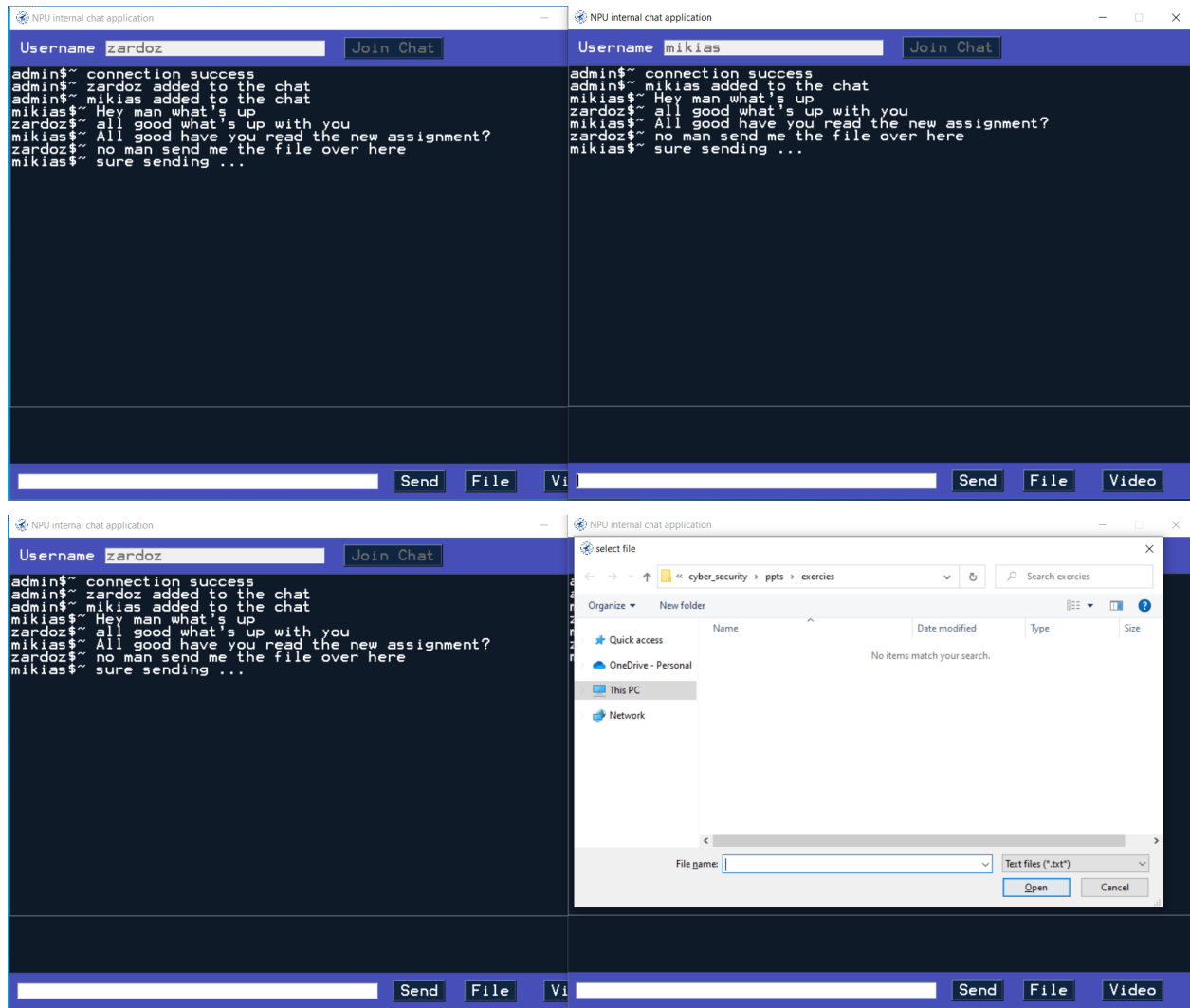
v_button = tk.Button(b_frame, text="Video", font=font,
bg="#0f2642", fg="white", command=video_call_request)
v_button.pack(side=tk.LEFT, padx=20)

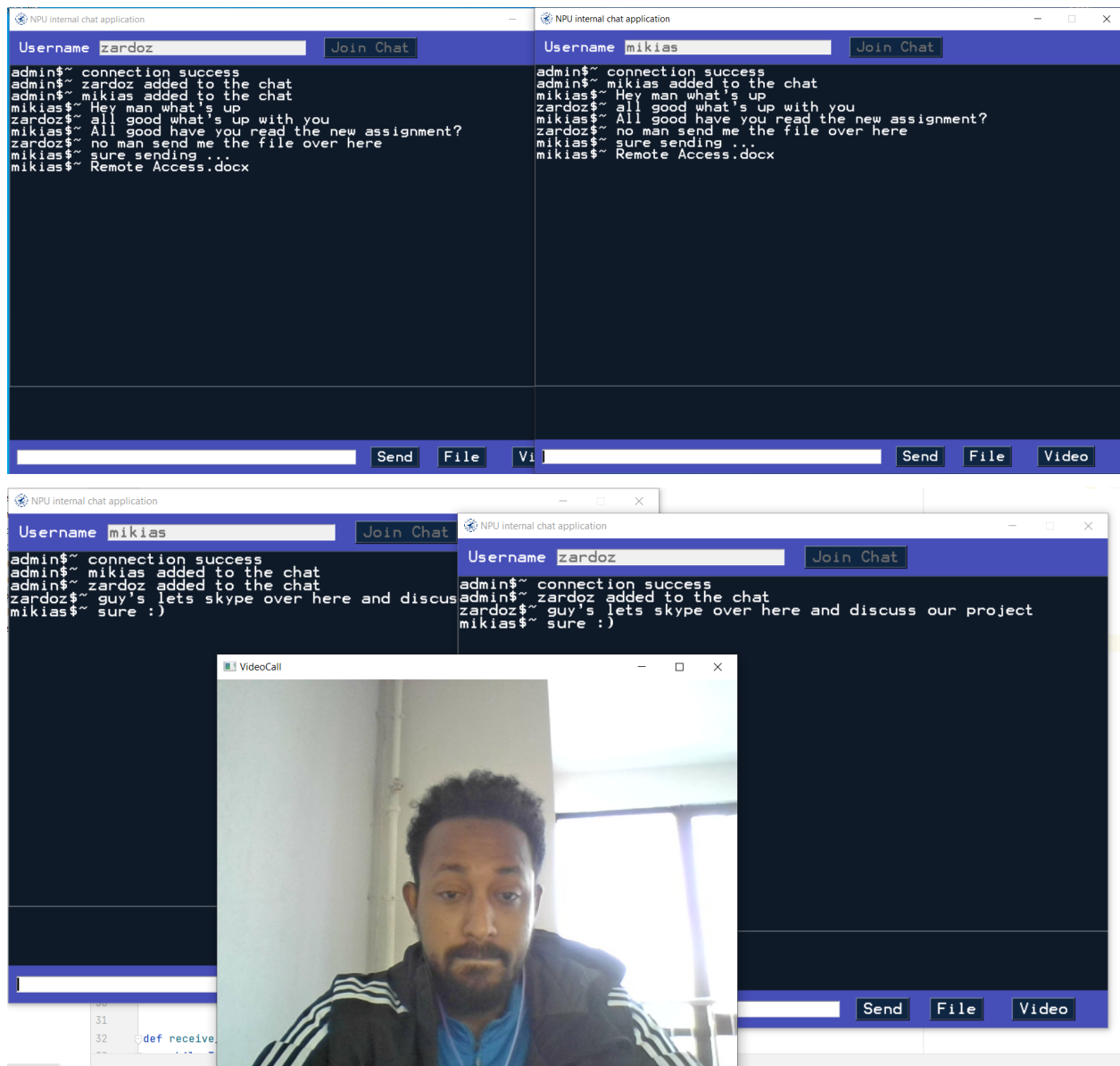
m_box = scr_text.ScrolledText(m_frame, font=font,
bg="#0f1a29", fg="white", width=67, height=26.5)
m_box.config(state=tk.DISABLED)
m_box.pack(side=tk.TOP)

```

Once we start running our programs we will have a minimal application with messaging, file sharing and video streaming functionalities. It's always important to run our chatting server and video streaming server first then run the client side so that there won't be any connection related problems.







II. Designing Access Control List

What is ACL?

ACL stands for network Access Control List which is a combination of rules which will either allow access to a computer environment or deny it. When using ACLs we allow only traffic with an address in our list to access the network. This ensures that only allowed traffic will enter our network. This will keep bad actors away from the network.

Why do we use ACL?

We use ACLs to protect our network environment from unknown traffic. ACLs are capable of denying access to incoming traffic from unknown sources. This will protect our network from hackers and other bad actors who want to compromise our network infrastructure.

Where can you place an ACL?

ACLs can be placed in switches and routers. When they are placed in switches and routers they will filter traffic using predefined rules defined by the network administrator. Usually they are placed on edge routers of our network. This will help us to traffic before it enters our system. To do this we can use a router with pre configured ACL, between the demilitarized zone (DMZ) and the internet. The DMZ contains a number of devices like web servers, VPNs etc...

What are the main components of ACL?

- 1) **Sequence Number**: a number which is used to identify the ACL entry.
- 2) **ACL name**: a name which is used to define an ACL entry.
- 3) **Remark**: comments which are used to give some detail about the ACL entry.
- 4) **Statement**: definition of rule where we either permit or deny a traffic source using a wildcard mask or address.
- 5) **Network protocol**: This defines the network transmission protocols TCP and UDP
- 6) **Source or destination**: defines the traffic source or destination using IP address or range of IP address.
- 7) **Log**: used for maintaining a log when an ACL match is found.
- 8) **Other criteria**: for more advanced ACLs control traffic using Type of Service (ToS), IP precedence, and differentiated services codepoint (DSCP) priority.

What are types of ACLs?

1) Standard ACL

The standard ACL protects the network using only source address. This is used for simple network configurations and deployments which doesn't really provide a strong security.

2) Extended ACL

Here the traffic can be blocked using source and destination address which comes from a single host or an entire network. We can also block traffic based on the transmission protocol like TCP, UDP, ICMP etc...

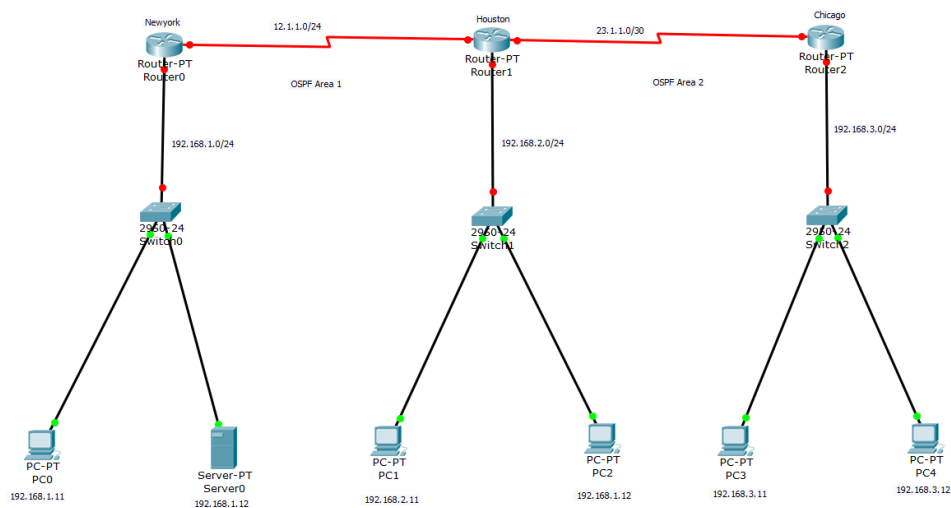
3) Dynamic ACL

Dynamic ACLs depend on extended ACLs, which are referred to as “Lock and Key” and can be used for specific timeframes. They permit access to users with proper authentication credentials.

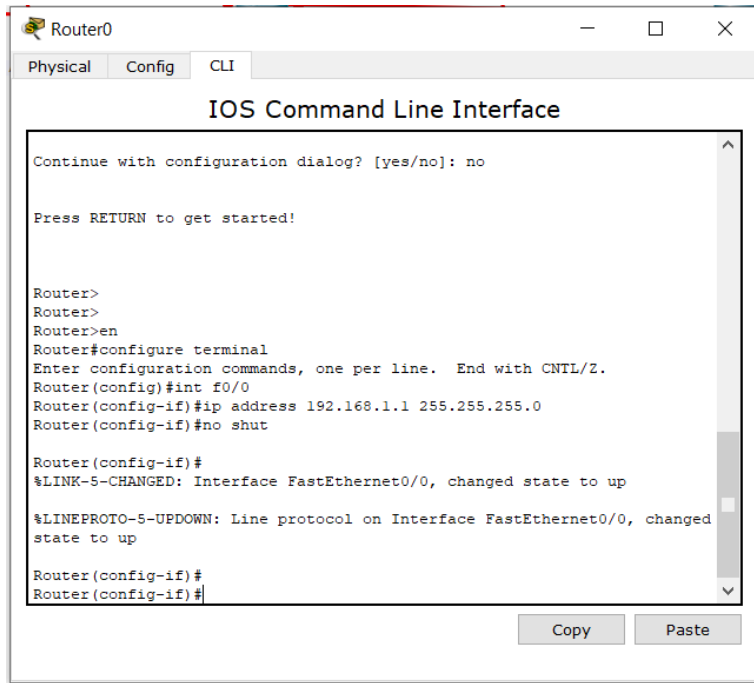
4) Reflexive ACL

These ACLs are also called IP session ACLs, which filter traffic based on upper layer session information. They check if the session originated inside the permitted traffic bound or not. Outbound ACL traffic is denied access whereas inbound traffic is permitted to enter the network.

We are asked to configure ACLs on a given network technology where the components are connected using OSPF on different areas. Once we place the components on the workspaces and connect them with cables we will get a network topology as shown below.



We can then configure our routers since they are going to be as a default gateway for our components on each network.



Router0

Physical Config CLI

IOS Command Line Interface

```
Continue with configuration dialog? [yes/no]: no

Press RETURN to get started!

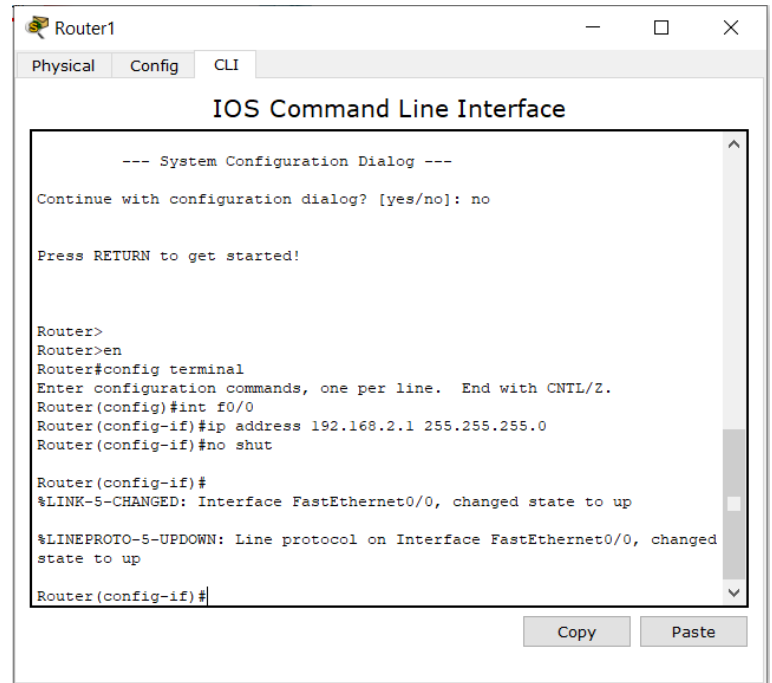
Router>
Router>
Router>en
Router#configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
Router(config)#int f0/0
Router(config-if)#ip address 192.168.1.1 255.255.255.0
Router(config-if)#no shut

Router(config-if)#
%LINK-5-CHANGED: Interface FastEthernet0/0, changed state to up

%LINEPROTO-5-UPDOWN: Line protocol on Interface FastEthernet0/0, changed
state to up

Router(config-if)#
Router(config-if)#
```

Copy Paste



Router1

Physical Config CLI

IOS Command Line Interface

```
--- System Configuration Dialog ---

Continue with configuration dialog? [yes/no]: no

Press RETURN to get started!

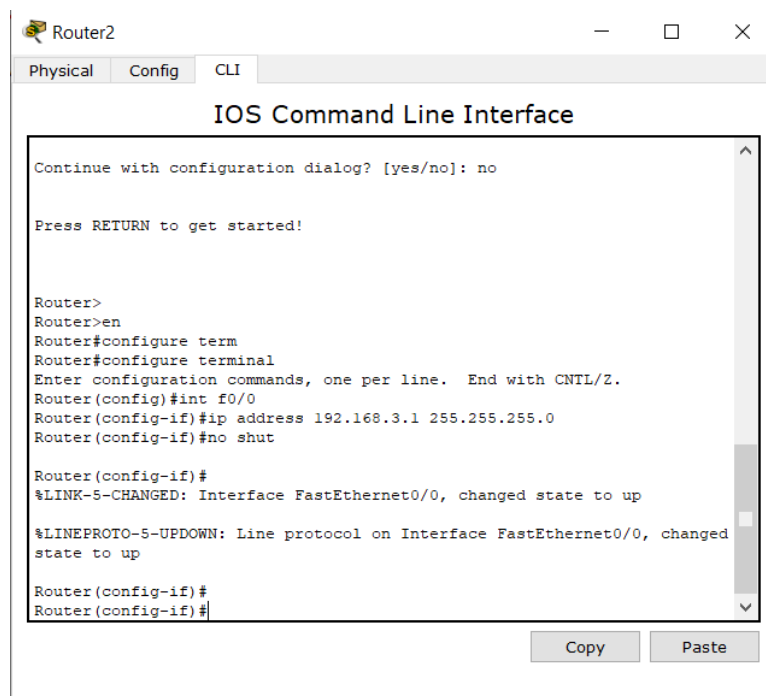
Router>
Router>en
Router#config terminal
Enter configuration commands, one per line. End with CNTL/Z.
Router(config)#int f0/0
Router(config-if)#ip address 192.168.2.1 255.255.255.0
Router(config-if)#no shut

Router(config-if)#
%LINK-5-CHANGED: Interface FastEthernet0/0, changed state to up

%LINEPROTO-5-UPDOWN: Line protocol on Interface FastEthernet0/0, changed
state to up

Router(config-if)#
Router(config-if)#
```

Copy Paste



Router2

Physical Config CLI

IOS Command Line Interface

```
Continue with configuration dialog? [yes/no]: no

Press RETURN to get started!

Router>
Router>en
Router#configure term
Router#configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
Router(config)#int f0/0
Router(config-if)#ip address 192.168.3.1 255.255.255.0
Router(config-if)#no shut

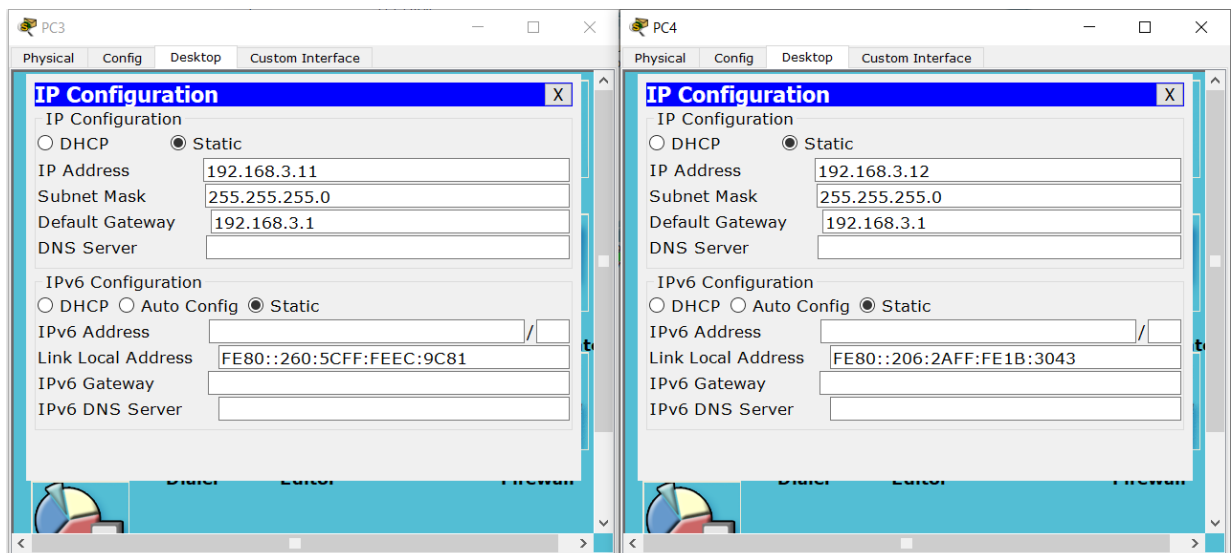
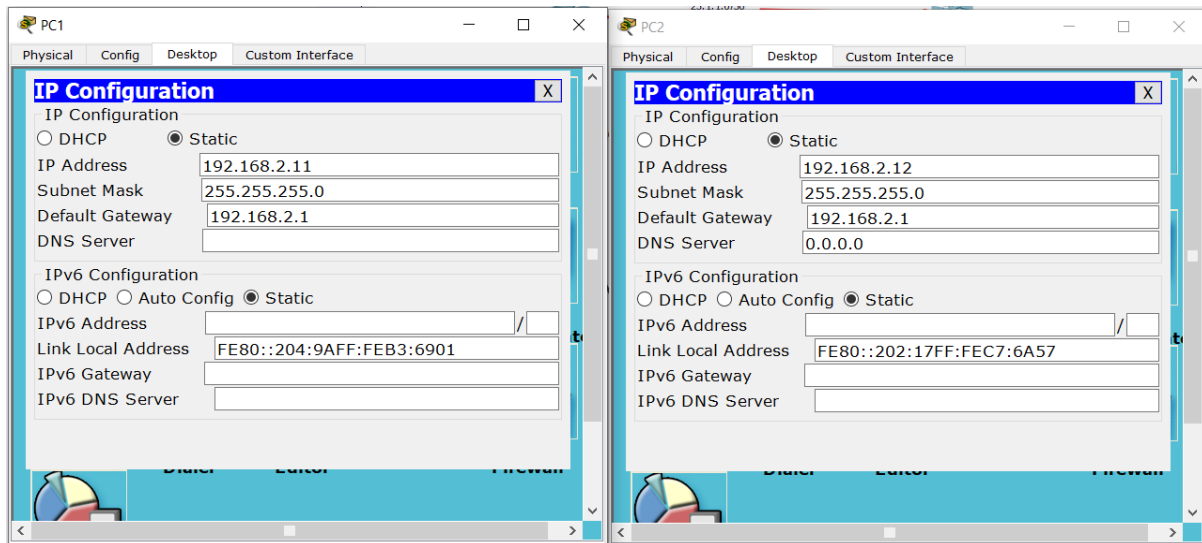
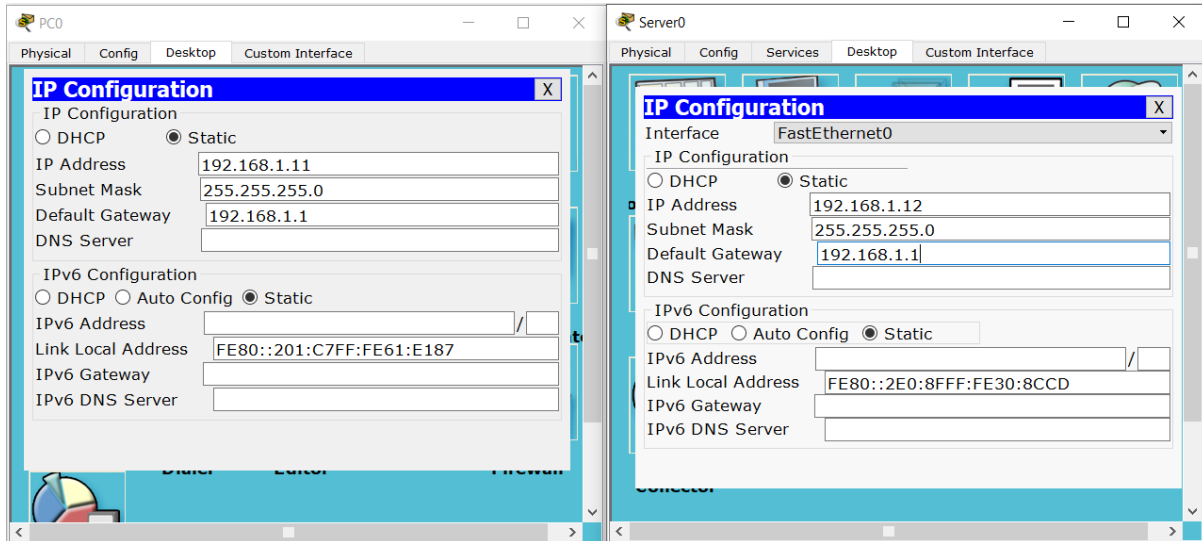
Router(config-if)#
%LINK-5-CHANGED: Interface FastEthernet0/0, changed state to up

%LINEPROTO-5-UPDOWN: Line protocol on Interface FastEthernet0/0, changed
state to up

Router(config-if)#
Router(config-if)#
```

Copy Paste

Once we configure our routers then we can move our individual computers and configure them accordingly their IP address, subnet mask and default gateway address in this case the IP address of our router.



The next thing to do is to configure the serial ports of our routers. These ports are connected to the outside internet or WAN rather than LAN which give internet access to the computers connected with them.

Router2 configuration window showing the Serial2/0 interface settings. The interface is configured with IP Address 23.1.1.20 and Subnet Mask 255.0.0.0. The clock rate is set to 'Not Set'. The port status is 'On' and duplex is 'Full Duplex'. The Tx Ring Limit is 10.

Equivalent IOS Commands:

```
ip address 23.1.1.20 255.0.0.0
Router(config-if)#
Router(config-if)#exit
Router(config)#interface Serial2/0
Router(config-if)#
```

Router1 configuration window showing the Serial2/0 interface settings. The interface is configured with IP Address 12.1.1.20 and Subnet Mask 255.0.0.0. The clock rate is set to 'Not Set'. The port status is 'On' and duplex is 'Full Duplex'. The Tx Ring Limit is 10.

Equivalent IOS Commands:

```
Router(config)#interface Serial2/0
Router(config-if)#
Router(config-if)#exit
Router(config)#interface Serial2/0
Router(config-if)#
```

Router1 configuration window showing the Serial3/0 interface settings. The interface is configured with IP Address 23.1.1.10 and Subnet Mask 255.0.0.0. The clock rate is set to 64000. The port status is 'On' and duplex is 'Full Duplex'. The Tx Ring Limit is 10.

Equivalent IOS Commands:

```
to up
Router(config-if)#exit
Router(config)#interface Serial3/0
Router(config-if)#
```

Router0 configuration window showing the Serial2/0 interface settings. The interface is configured with IP Address 12.1.1.10 and Subnet Mask 255.0.0.0. The clock rate is set to 64000. The port status is 'On' and duplex is 'Full Duplex'. The Tx Ring Limit is 10.

Equivalent IOS Commands:

```
to up
Router(config-if)#exit
Router(config)#interface Serial2/0
Router(config-if)#
```

To configure our OSPF on each individual router, OSPF stands for (Open Shortest Path First) which is a network configuration method that allows nodes to communicate with each other from different areas or regions.

The image displays four screenshots of the Cisco IOS Command Line Interface (CLI) for different routers, showing the configuration of OSPF (Open Shortest Path First) protocol.

Router0: The CLI shows the configuration of OSPF on interface Serial3/0. The configuration includes setting the interface to 'no shutdown', assigning an IP address of 23.1.1.10 with a subnet mask of 255.0.0.0, and setting the clock rate to 64000. The OSPF process is then configured with 'router ospf 10', and the network 192.168.1.0 0.255.255.255 area 1 is added. The configuration is saved to the startup configuration.

Router1: The CLI shows the configuration of OSPF on interface Serial3/0. The configuration includes setting the interface to 'no shutdown', assigning an IP address of 23.1.1.10 with a subnet mask of 255.0.0.0, and setting the clock rate to 64000. The OSPF process is then configured with 'router ospf 10', and the network 192.168.1.0 0.255.255.255 area 1 is added. The configuration is saved to the startup configuration.

Router2: The CLI shows the configuration of OSPF on interface Serial3/0. The configuration includes setting the interface to 'no shutdown', assigning an IP address of 23.1.1.10 with a subnet mask of 255.0.0.0, and setting the clock rate to 64000. The OSPF process is then configured with 'router ospf 10', and the network 192.168.1.0 0.255.255.255 area 1 is added. The configuration is saved to the startup configuration.

Router3: The CLI shows the configuration of OSPF on interface Serial3/0. The configuration includes setting the interface to 'no shutdown', assigning an IP address of 23.1.1.10 with a subnet mask of 255.0.0.0, and setting the clock rate to 64000. The OSPF process is then configured with 'router ospf 10', and the network 192.168.1.0 0.255.255.255 area 1 is added. The configuration is saved to the startup configuration.

Once we finish configuring our OSPF settings then we can start pinging and test if the connection works as expected. For this we can go to every computer and test the connection from the nearest node to the farthest node. This will help to be sure that each node on the network is functioning as expected with any issue.

PC0

Physical Config Desktop Custom Interface

Command Prompt

```
Reply from 192.168.1.1: bytes=32 time=1ms TTL=255
Reply from 192.168.1.1: bytes=32 time=0ms TTL=255
Reply from 192.168.1.1: bytes=32 time=0ms TTL=255
Reply from 192.168.1.1: bytes=32 time=1ms TTL=255

Ping statistics for 192.168.1.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 1ms, Average = 0ms

PC>ping 12.1.1.10

Pinging 12.1.1.10 with 32 bytes of data:

Reply from 12.1.1.10: bytes=32 time=1ms TTL=255
Reply from 12.1.1.10: bytes=32 time=0ms TTL=255
Reply from 12.1.1.10: bytes=32 time=0ms TTL=255
Reply from 12.1.1.10: bytes=32 time=0ms TTL=255

Ping statistics for 12.1.1.10:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 1ms, Average = 0ms

PC>ping 12.1.1.20

Pinging 12.1.1.20 with 32 bytes of data:

Reply from 12.1.1.20: bytes=32 time=1ms TTL=254
Reply from 12.1.1.20: bytes=32 time=1ms TTL=254
Reply from 12.1.1.20: bytes=32 time=1ms TTL=254
Reply from 12.1.1.20: bytes=32 time=2ms TTL=254

Ping statistics for 12.1.1.20:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 2ms, Average = 1ms

PC>ping 192.168.2.11

Pinging 192.168.2.11 with 32 bytes of data:

Reply from 192.168.2.11: bytes=32 time=1ms TTL=126
Reply from 192.168.2.11: bytes=32 time=1ms TTL=126
Reply from 192.168.2.11: bytes=32 time=3ms TTL=126
Reply from 192.168.2.11: bytes=32 time=3ms TTL=126

Ping statistics for 192.168.2.11:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 3ms, Average = 1ms

PC>ping 192.168.2.12

Pinging 192.168.2.12 with 32 bytes of data:
```

PC2

Physical Config Desktop Custom Interface

Command Prompt

```
PC>ping 12.1.1.20

Pinging 12.1.1.20 with 32 bytes of data:

Reply from 12.1.1.20: bytes=32 time=0ms TTL=255
Reply from 12.1.1.20: bytes=32 time=0ms TTL=255
Reply from 12.1.1.20: bytes=32 time=0ms TTL=255
Reply from 12.1.1.20: bytes=32 time=0ms TTL=255

Ping statistics for 12.1.1.20:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

PC>ping 12.1.1.10

Pinging 12.1.1.10 with 32 bytes of data:

Reply from 12.1.1.10: bytes=32 time=0ms TTL=254
Reply from 12.1.1.10: bytes=32 time=0ms TTL=254
Reply from 12.1.1.10: bytes=32 time=1ms TTL=254
Reply from 12.1.1.10: bytes=32 time=1ms TTL=254

Ping statistics for 12.1.1.10:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 9ms, Average = 4ms

PC>ping 192.168.1.11

Pinging 192.168.1.11 with 32 bytes of data:

Reply from 192.168.1.11: bytes=32 time=10ms TTL=126
Reply from 192.168.1.11: bytes=32 time=1ms TTL=126
Reply from 192.168.1.11: bytes=32 time=2ms TTL=126
Reply from 192.168.1.11: bytes=32 time=1ms TTL=126

Ping statistics for 192.168.1.11:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 10ms, Average = 3ms

PC>ping 192.168.1.12

Pinging 192.168.1.12 with 32 bytes of data:

Reply from 192.168.1.12: bytes=32 time=10ms TTL=126
Reply from 192.168.1.12: bytes=32 time=1ms TTL=126
Reply from 192.168.1.12: bytes=32 time=1ms TTL=126
Reply from 192.168.1.12: bytes=32 time=1ms TTL=126

Ping statistics for 192.168.1.12:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 10ms, Average = 3ms

PC>
```

```
PC4
Physical Config Desktop Custom Interface

Command Prompt

C:\>ping 23.1.1.20
Pinging 23.1.1.20: bytes=32 time=1ms TTL=255
Reply from 23.1.1.20: bytes=32 time=0ms TTL=255
Reply from 23.1.1.20: bytes=32 time=0ms TTL=255
Reply from 23.1.1.20: bytes=32 time=1ms TTL=255
Ping statistics for 23.1.1.20:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 1ms, Average = 0ms
PC>ping 23.1.1.10
Pinging 23.1.1.10 with 32 bytes of data:
Reply from 23.1.1.10: bytes=32 time=1ms TTL=254
Reply from 23.1.1.10: bytes=32 time=1ms TTL=254
Reply from 23.1.1.10: bytes=32 time=1ms TTL=254
Reply from 23.1.1.10: bytes=32 time=5ms TTL=254
Ping statistics for 23.1.1.10:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 5ms, Average = 3ms
PC>ping 192.168.2.1
Pinging 192.168.2.1 with 32 bytes of data:
Reply from 192.168.2.1: bytes=32 time=1ms TTL=254
Reply from 192.168.2.1: bytes=32 time=1ms TTL=254
Reply from 192.168.2.1: bytes=32 time=7ms TTL=254
Reply from 192.168.2.1: bytes=32 time=7ms TTL=254
Ping statistics for 192.168.2.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 7ms, Average = 3ms
PC>ping 192.168.2.11
Pinging 192.168.2.11 with 32 bytes of data:
Reply from 192.168.2.11: bytes=32 time=1ms TTL=126
Reply from 192.168.2.11: bytes=32 time=1ms TTL=126
Reply from 192.168.2.11: bytes=32 time=10ms TTL=126
Reply from 192.168.2.11: bytes=32 time=3ms TTL=126
Ping statistics for 192.168.2.11:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 10ms, Average = 3ms
PC>ping 192.168.2.12
Pinging 192.168.2.12 with 32 bytes of data:
```

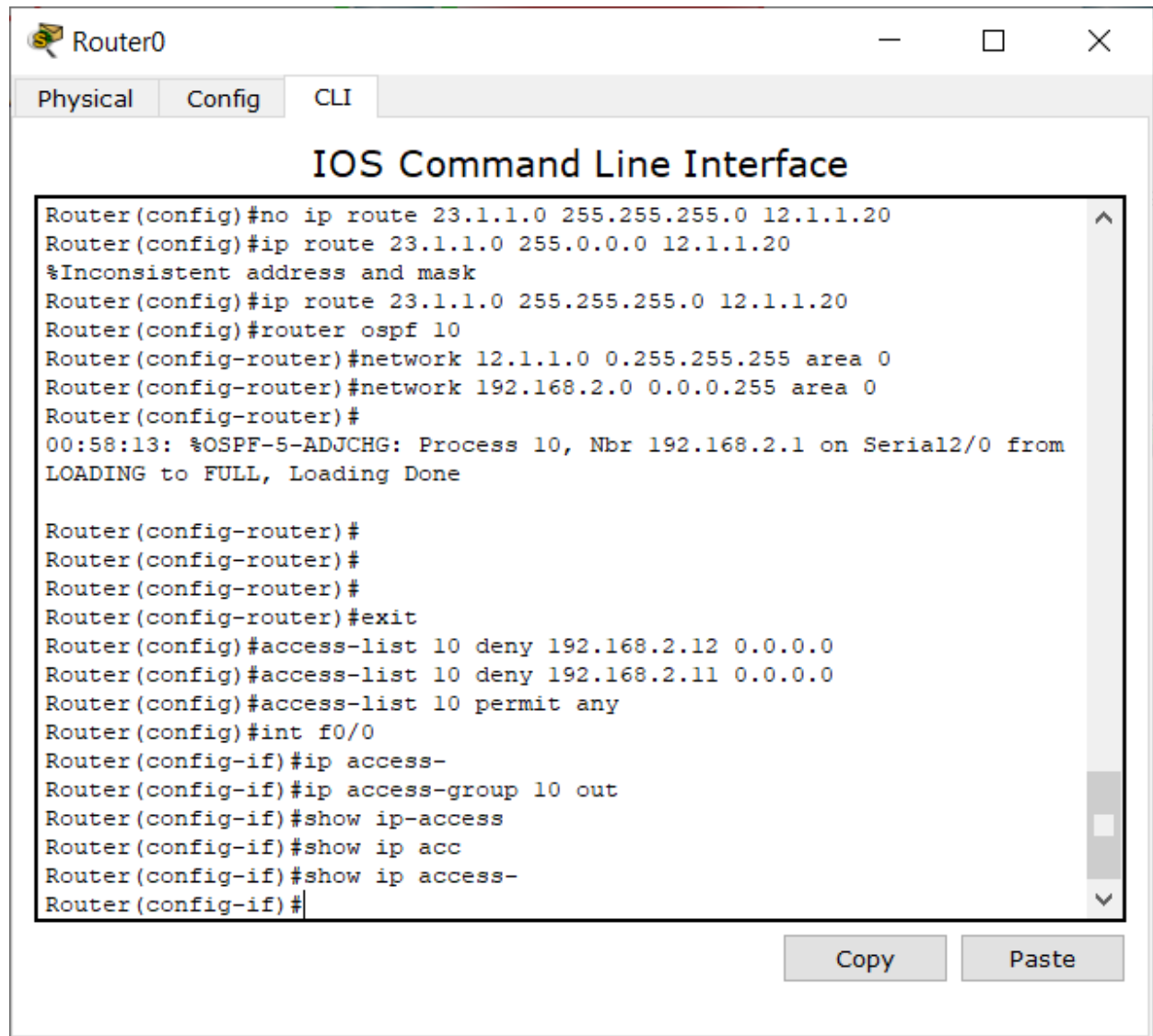
```
PC4
Physical Config Desktop Custom Interface

Command Prompt

PC>ping 12.1.1.20
Pinging 12.1.1.20 with 32 bytes of data:
Reply from 12.1.1.20: bytes=32 time=1ms TTL=254
Reply from 12.1.1.20: bytes=32 time=2ms TTL=254
Reply from 12.1.1.20: bytes=32 time=1ms TTL=254
Reply from 12.1.1.20: bytes=32 time=1ms TTL=254
Ping statistics for 12.1.1.20:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 2ms, Average = 1ms
PC>ping 12.1.1.10
Pinging 12.1.1.10 with 32 bytes of data:
Reply from 12.1.1.10: bytes=32 time=2ms TTL=253
Reply from 12.1.1.10: bytes=32 time=2ms TTL=253
Reply from 12.1.1.10: bytes=32 time=2ms TTL=253
Reply from 12.1.1.10: bytes=32 time=2ms TTL=253
Ping statistics for 12.1.1.10:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 2ms, Maximum = 2ms, Average = 2ms
PC>ping 192.168.1.1
Pinging 192.168.1.1 with 32 bytes of data:
Reply from 192.168.1.1: bytes=32 time=12ms TTL=253
Reply from 192.168.1.1: bytes=32 time=2ms TTL=253
Reply from 192.168.1.1: bytes=32 time=2ms TTL=253
Reply from 192.168.1.1: bytes=32 time=2ms TTL=253
Ping statistics for 192.168.1.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 2ms, Maximum = 12ms, Average = 4ms
PC>ping 192.168.1.12
Pinging 192.168.1.12 with 32 bytes of data:
Request timed out.
Reply from 192.168.1.12: bytes=32 time=2ms TTL=125
Reply from 192.168.1.12: bytes=32 time=2ms TTL=125
Reply from 192.168.1.12: bytes=32 time=2ms TTL=125
Ping statistics for 192.168.1.12:
    Packets: Sent = 4, Received = 3, Lost = 1 (25% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 2ms, Maximum = 2ms, Average = 2ms
PC>
```

Once we make sure that all the nodes are connected and there is no issue then we can continue configuring the ACLs on the router where we want to permit and deny network traffic. According to the requirement we want to deny traffic coming from the Houston

network block to the Newyork network block, to do that we will apply ACL on the Newyork router.



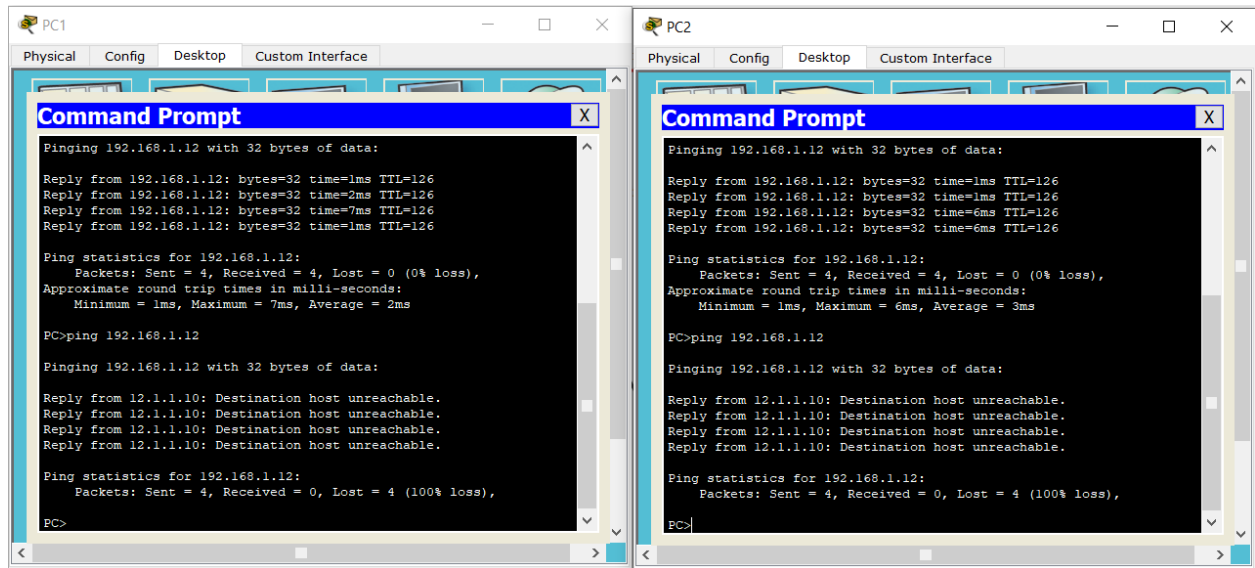
The screenshot shows a window titled "Router0" with three tabs: "Physical", "Config", and "CLI". The "CLI" tab is active, displaying the "IOS Command Line Interface". The terminal text shows the following commands and output:

```
Router(config)#no ip route 23.1.1.0 255.255.255.0 12.1.1.20
Router(config)#ip route 23.1.1.0 255.0.0.0 12.1.1.20
%Inconsistent address and mask
Router(config)#ip route 23.1.1.0 255.255.255.0 12.1.1.20
Router(config)#router ospf 10
Router(config-router)#network 12.1.1.0 0.255.255.255 area 0
Router(config-router)#network 192.168.2.0 0.0.0.255 area 0
Router(config-router)#
00:58:13: %OSPF-5-ADJCHG: Process 10, Nbr 192.168.2.1 on Serial2/0 from
LOADING to FULL, Loading Done

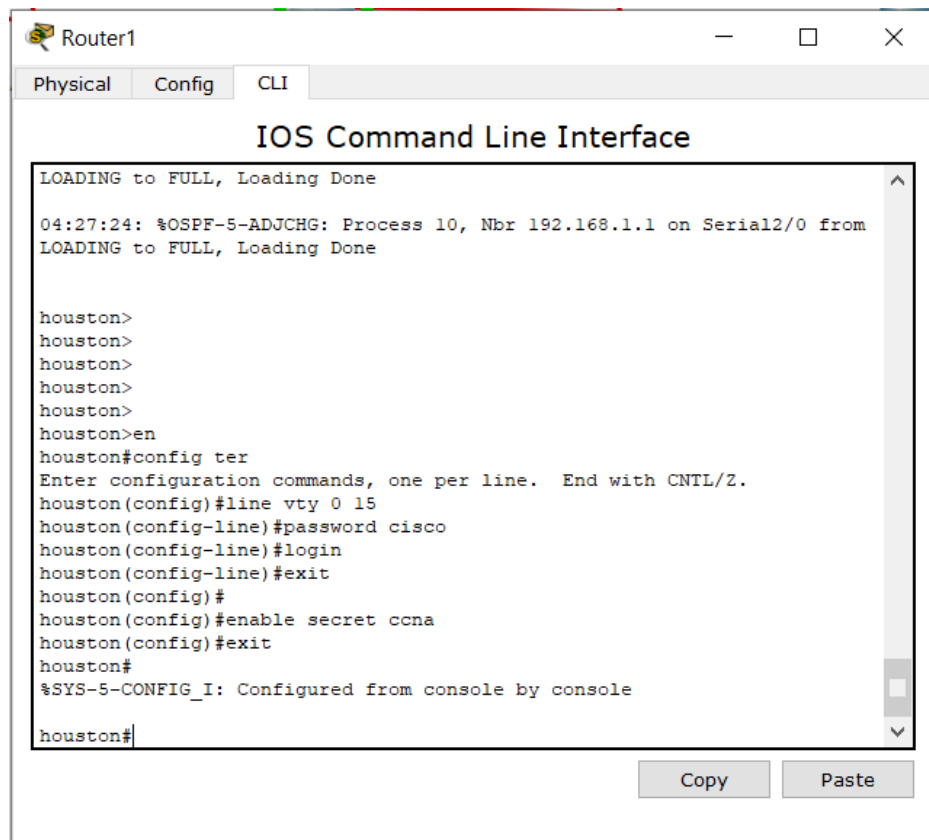
Router(config-router)#
Router(config-router)#
Router(config-router)#
Router(config-router)#exit
Router(config)#access-list 10 deny 192.168.2.12 0.0.0.0
Router(config)#access-list 10 deny 192.168.2.11 0.0.0.0
Router(config)#access-list 10 permit any
Router(config)#int f0/0
Router(config-if)#ip access-
Router(config-if)#ip access-group 10 out
Router(config-if)#show ip-access
Router(config-if)#show ip acc
Router(config-if)#show ip access-
Router(config-if)#
```

At the bottom right of the CLI window, there are two buttons: "Copy" and "Paste".

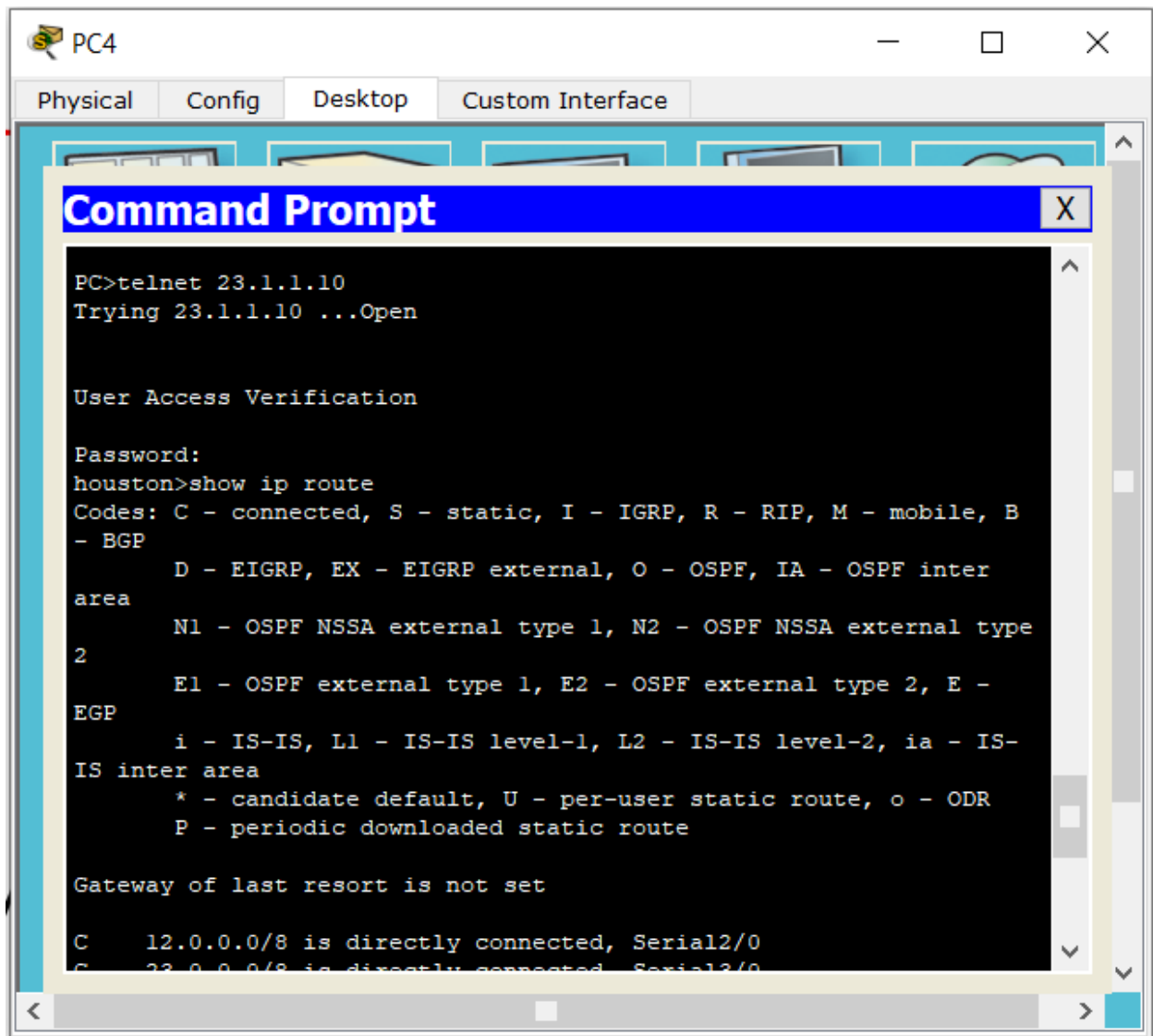
Once we configure the ACL we can go ahead and try to ping using computers from the Houston block to the computers inside the Newyork block. The ping result shows that the destination host is unreachable.



The second requirement is configure the telnet service on the Houston router and block any attempt to access telnet from the Chicago block, and configure the Newyork router telnet service and give access to the computers on the Chicago block to get telnet service on the Newyork router, to achieve this we will configure extended ACLs. First we will set up the telnet service on the Houston router.



Once we set up the telnet password we can try to connect to the router using the PC inside the Chicago block and it works perfectly.



The screenshot shows a window titled "PC4" with tabs for "Physical", "Config", "Desktop", and "Custom Interface". The "Desktop" tab is active, displaying a "Command Prompt" window. The terminal session is as follows:

```
PC>telnet 23.1.1.10
Trying 23.1.1.10 ...Open

User Access Verification

Password:
houston>show ip route
Codes: C - connected, S - static, I - IGRP, R - RIP, M - mobile, B
- BGP
      D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter
area
      N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type
2
      E1 - OSPF external type 1, E2 - OSPF external type 2, E -
EGP
      i - IS-IS, L1 - IS-IS level-1, L2 - IS-IS level-2, ia - IS-
IS inter area
      * - candidate default, U - per-user static route, o - ODR
      P - periodic downloaded static route

Gateway of last resort is not set

C    12.0.0.0/8 is directly connected, Serial2/0
C    23.0.0.0/8 is directly connected, Serial3/0
```

Then we can configure our extended ACL to filter traffic and block telnet service from being accessed for computers from the Chicago block.

```
Router2
Physical Config CLI
IOS Command Line Interface

Router(config)#access-list 100 deny ?
  ahp      Authentication Header Protocol
  eigrp     Cisco's EIGRP routing protocol
  esp      Encapsulation Security Payload
  gre      Cisco's GRE tunneling
  icmp     Internet Control Message Protocol
  ip       Any Internet Protocol
  ospf     OSPF routing protocol
  tcp      Transmission Control Protocol
  udp      User Datagram Protocol
Router(config)#access-list 100 deny tcp ?
  A.B.C.D  Source address
  any      Any source host
  host     A single source host
Router(config)#access-list 100 deny tcp 192.168.3.0 0.0.0.255 ?
  A.B.C.D  Destination address
  any      Any destination host
  eq       Match only packets on a given port number
  gt       Match only packets with a greater port number
  host     A single destination host
  lt       Match only packets with a lower port number
  neq      Match only packets not on a given port number
  range    Match only packets in the range of port numbers
Router(config)#access-list 100 deny tcp 192.168.3.0 0.0.0.255 23.1.1.10 eq ?
  % Unrecognized command
Router(config)#access-list 100 deny tcp 192.168.3.0 0.0.0.255 host 23.1.1.10 eq ?
  <0-65535> Port number
  ftp      File Transfer Protocol (21)
  pop3     Post Office Protocol v3 (110)
  smtp     Simple Mail Transport Protocol (25)
  telnet   Telnet (23)
  www      World Wide Web (HTTP, 80)
Router(config)#access-list 100 deny tcp 192.168.3.0 0.0.0.255 host 23.1.1.10 eq 23
Router(config)#access-list 100 deny tcp 192.168.3.0 0.0.0.255 host 192.168.2.1 eq 23
Router(config)#access-list 100 permit any
  ^
% Invalid input detected at '^' marker.

Router(config)#access-list 100 permit ip any any
Router(config)#
Router(config)#
Router(config)#do show access
Router(config)#exit
Router#
$SYS-5-CONFIG_I: Configured from console by console

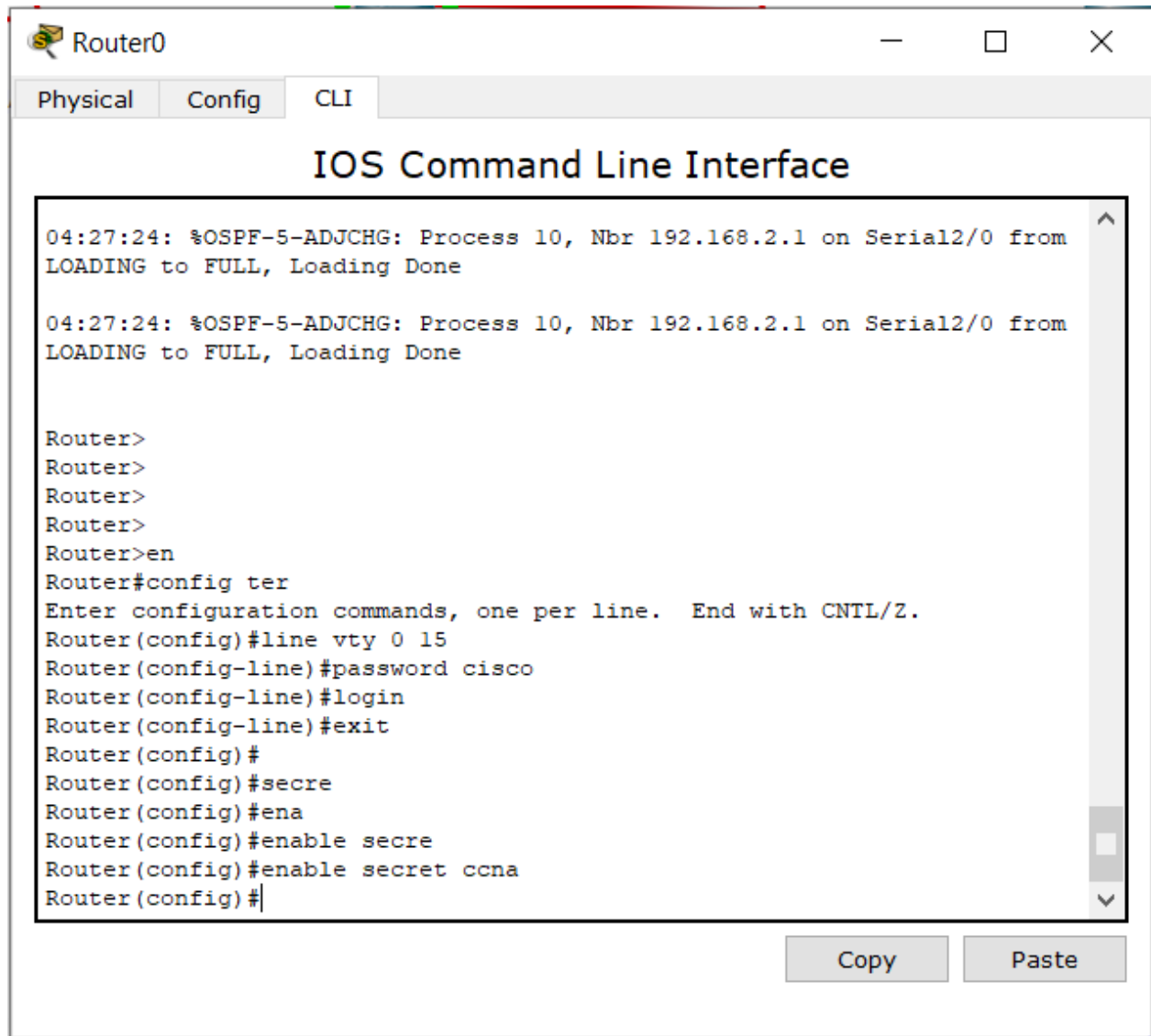
Router#config-ter
Router#config ter
Enter configuration commands, one per line. End with CNTL/Z.
Router(config)#do sh acc
Extended IP access list 100
  10 deny tcp 192.168.3.0 0.0.0.255 host 23.1.1.10 eq telnet
  20 deny tcp 192.168.3.0 0.0.0.255 host 192.168.2.1 eq telnet
  30 permit ip any any
Router(config)#int f0/0
Router(config-if)#ip acc
Router(config-if)#ip access-group 100 in
```

Once the ACL is configured we can go ahead and try to connect to the router through a computer from the Chicago block and we can see that it's blocked and can no longer access

telnet on the Houston network.

[illegible]

Then configure the Newyork router's telnet service and set the password



Once we set the password for telnet we can try to access it using a computer inside the Chicago network and it should work fine.

```

PC>ping 12.1.1.10

Pinging 12.1.1.10 with 32 bytes of data:

Reply from 12.1.1.10: bytes=32 time=2ms TTL=253
Reply from 12.1.1.10: bytes=32 time=3ms TTL=253
Reply from 12.1.1.10: bytes=32 time=2ms TTL=253
Reply from 12.1.1.10: bytes=32 time=3ms TTL=253

Ping statistics for 12.1.1.10:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 3ms, Average = 2ms

PC>ping 192.168.1.1

Pinging 192.168.1.1 with 32 bytes of data:

Reply from 192.168.1.1: bytes=32 time=19ms TTL=253
Reply from 192.168.1.1: bytes=32 time=2ms TTL=253
Reply from 192.168.1.1: bytes=32 time=2ms TTL=253
Reply from 192.168.1.1: bytes=32 time=2ms TTL=253

Ping statistics for 192.168.1.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 19ms, Average = 6ms

PC>telnet 12.1.1.10
Trying 12.1.1.10 ...Open

User Access Verification

Password:
Router>en
Password:
Router#show ip route
Codes: C - connected, S - static, I - IGRP, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2, E - EGP
       i - IS-IS, L1 - IS-IS level-1, L2 - IS-IS level-2, ia - IS-IS inter area
       * - candidate default, U - per-user static route, o - ODR
       P - periodic downloaded static route

Gateway of last resort is not set

C    12.0.0.0/8 is directly connected, Serial2/0
    23.0.0.0/8 is variably subnetted, 2 subnets, 2 masks
O IA   23.0.0.0/8 [110/128] via 12.1.1.20, 00:47:11, Serial2/0
S     23.1.1.0/24 [1/0] via 12.1.1.20
C    192.168.1.0/24 is directly connected, FastEthernet0/0
O IA  192.168.2.0/24 [110/65] via 12.1.1.20, 00:47:11, Serial2/0
S    192.168.3.0/24 [1/0] via 12.1.1.20
Router#

```