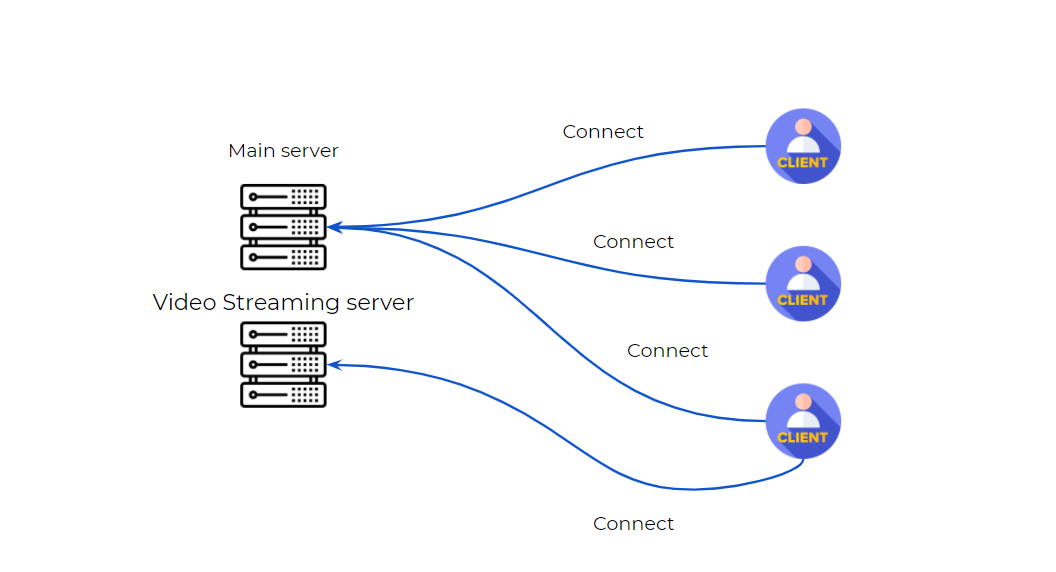
Network and Distributed Computing Final Project

# Chat Application

The first part of the project asks us to create a chat application with video streaming functionality and many more using socket programing. 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

1. 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** modulefunctions **gethostbyname()** and **gethostname()** .

*# constants*

IP = socket.gethostbyname(socket.gethostname())

PORT = 4433

users = []

1. 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.
   1. 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

* 1. 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

* 1. 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)

* 1. 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")

* 1. 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())

* 1. 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)

* 1. 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()

* 1. 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

1. 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

1. 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}")

1. 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

1. 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)

1. Create our client socket with a normal IPv4 address and TCP connection.

*# client socket object*

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

1. The client side implementation have 7 functions
   1. 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)

* 1. 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)

* 1. 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()

* 1. 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\_server, args=(client,)).start()

u\_textbox.config(state=tk.DISABLED)

u\_button.config(state=tk.DISABLED)

* 1. 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")

* 1. 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")

* 1. 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()

1. 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)

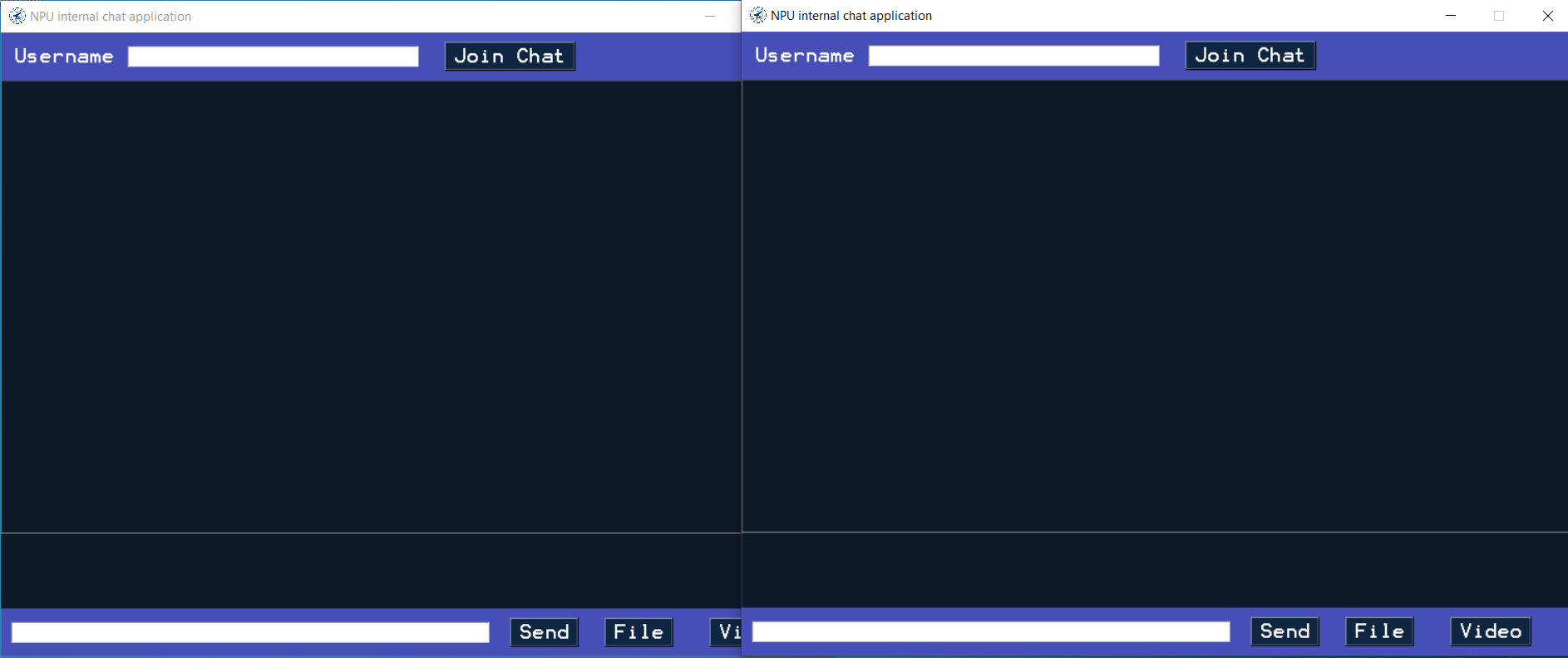
v\_button.pack(side=tk.LEFT, padx=20)

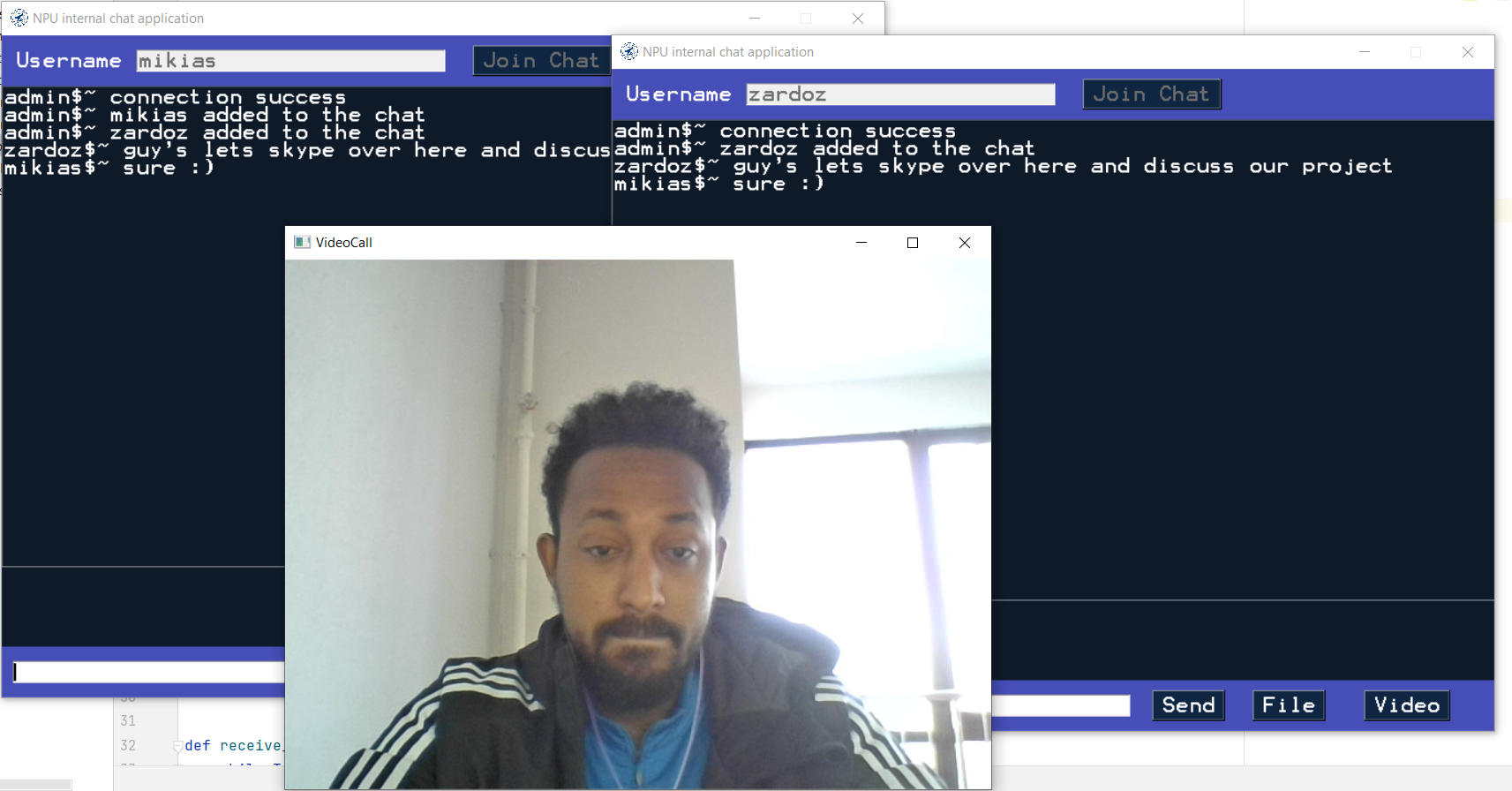
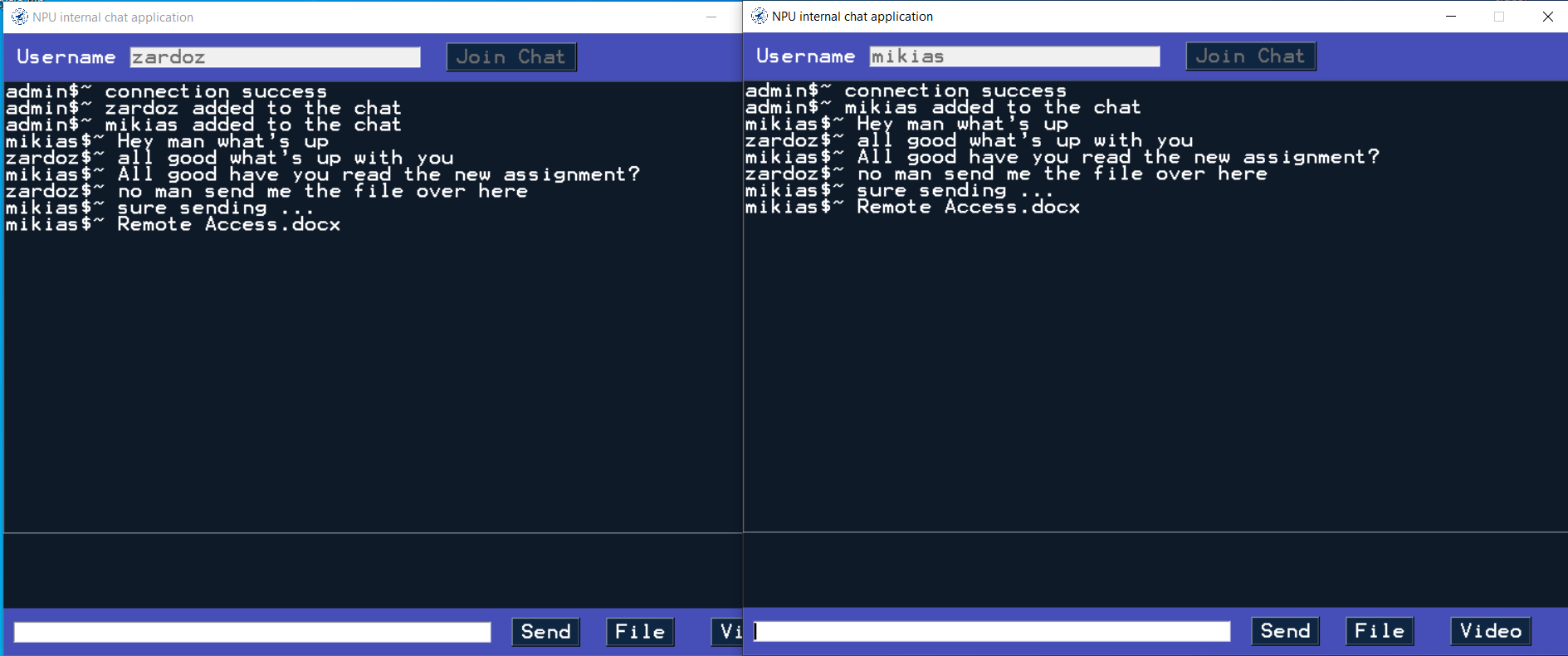
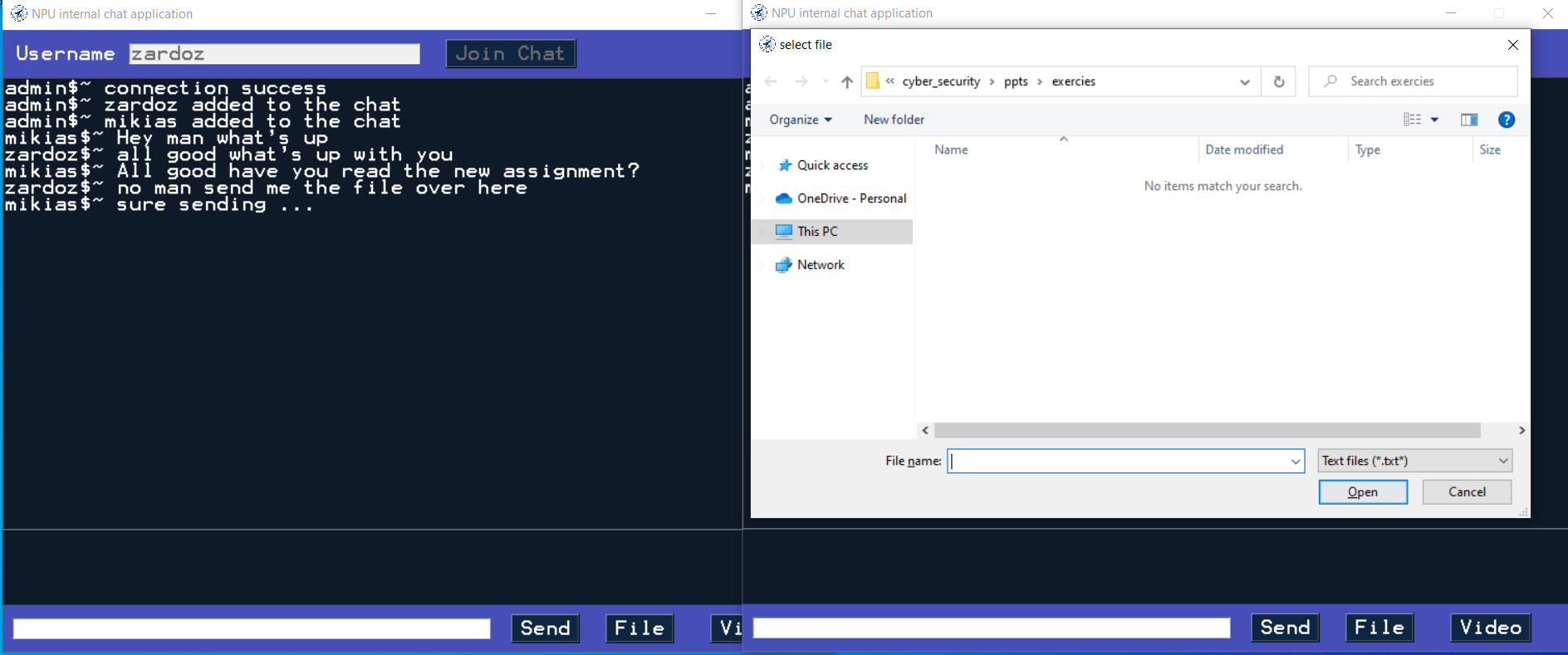
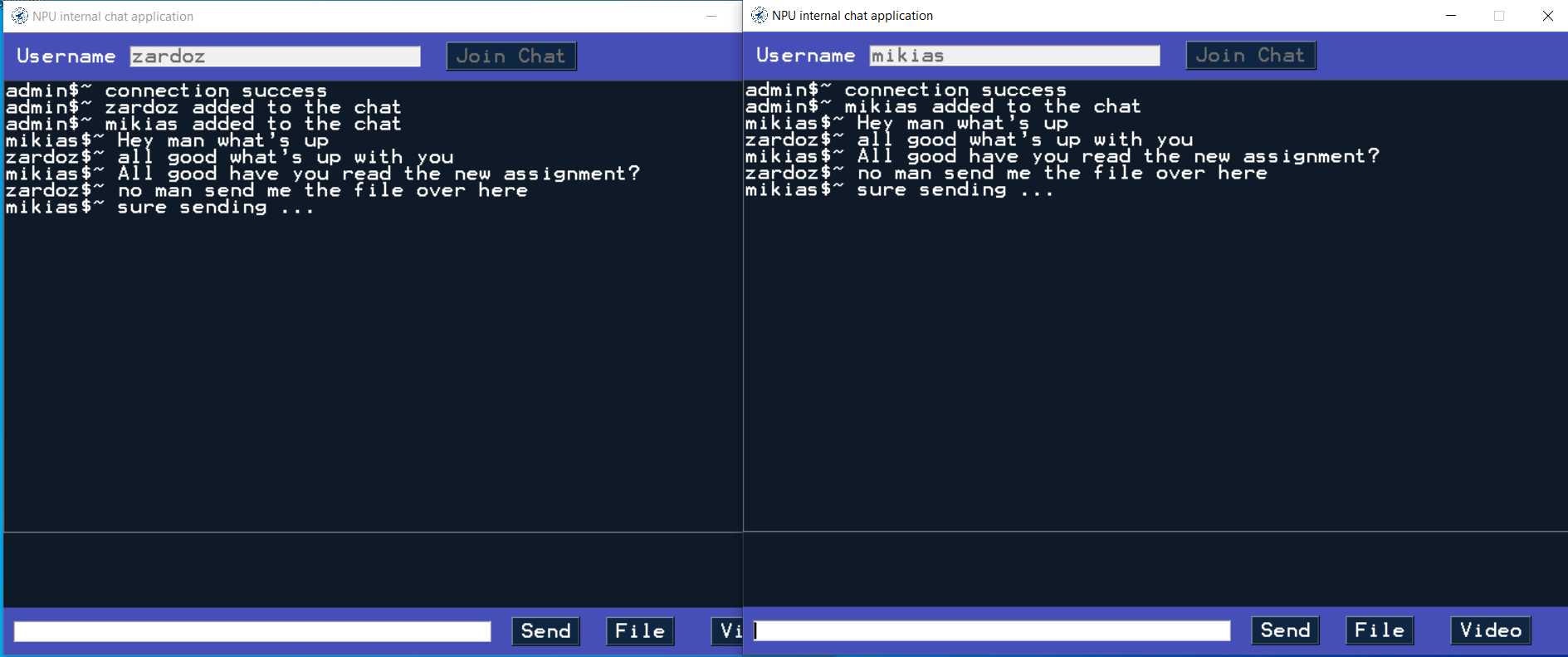
m\_box = scr\_text.ScrolledText(m\_frame, font=font, bg="#0f1a29", fg="white", width=67, height=26.5)

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.

1. **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…

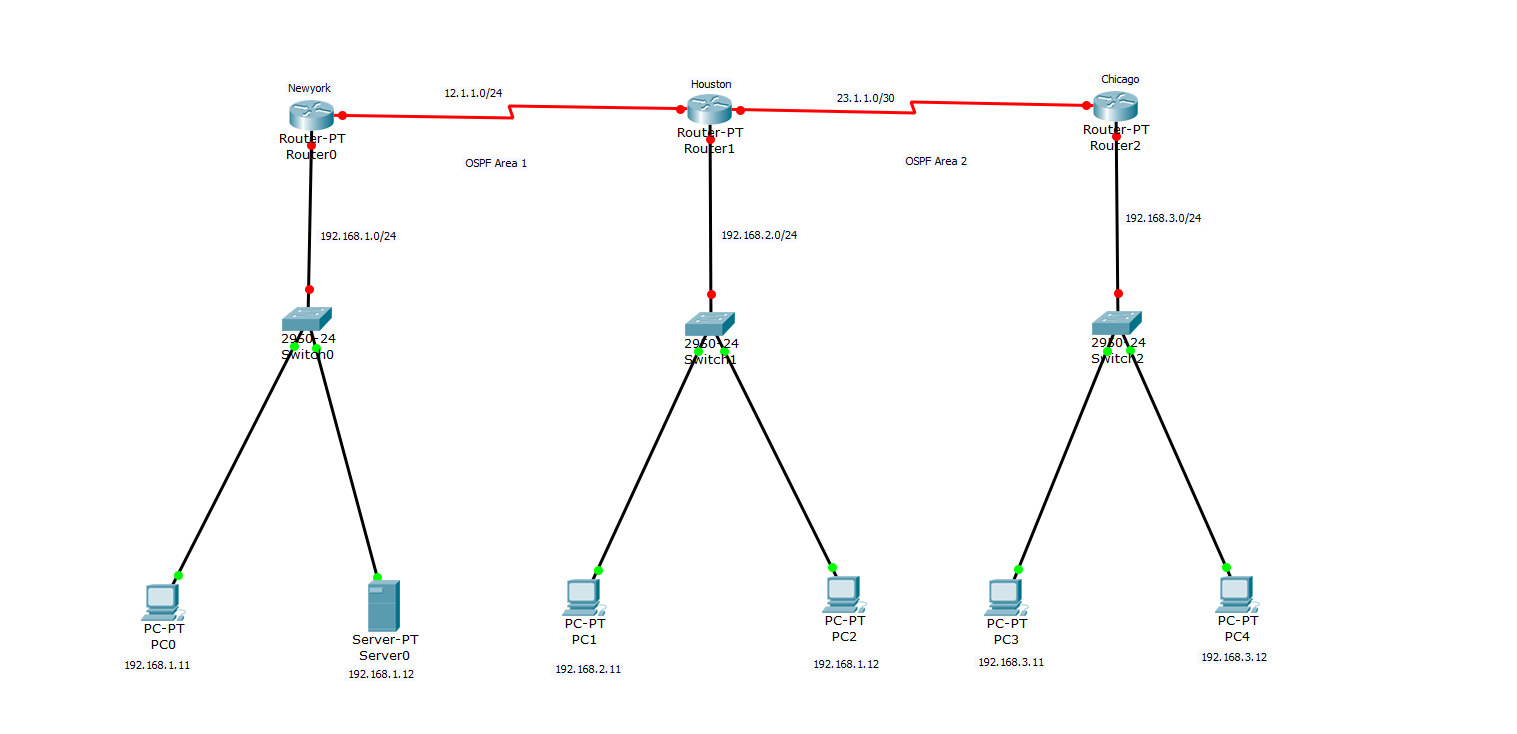
1. **Dynamic ACL**

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.

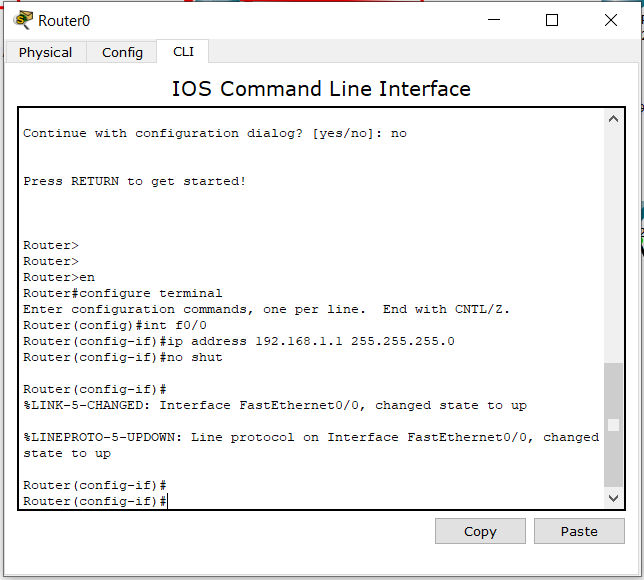
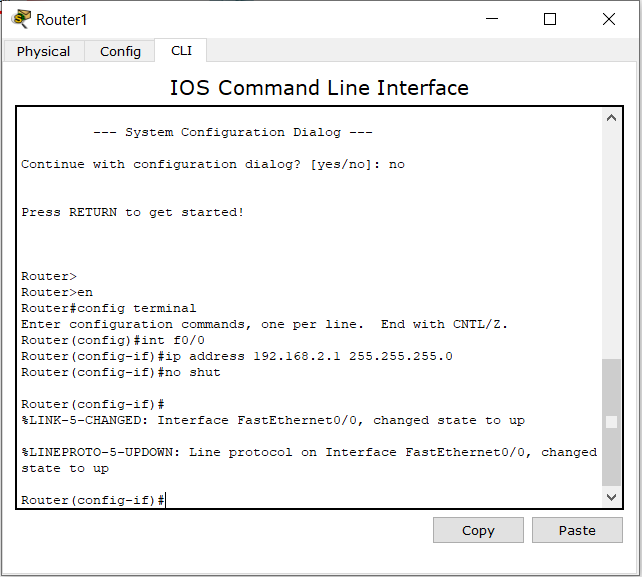
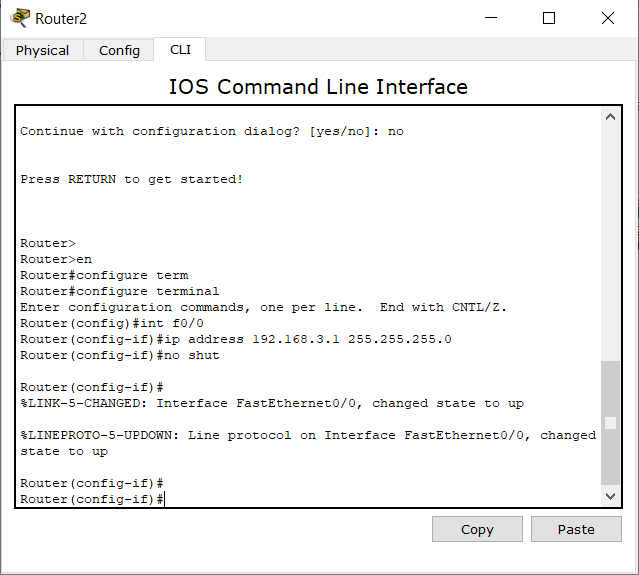
1. **Reflexive ACL**

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.

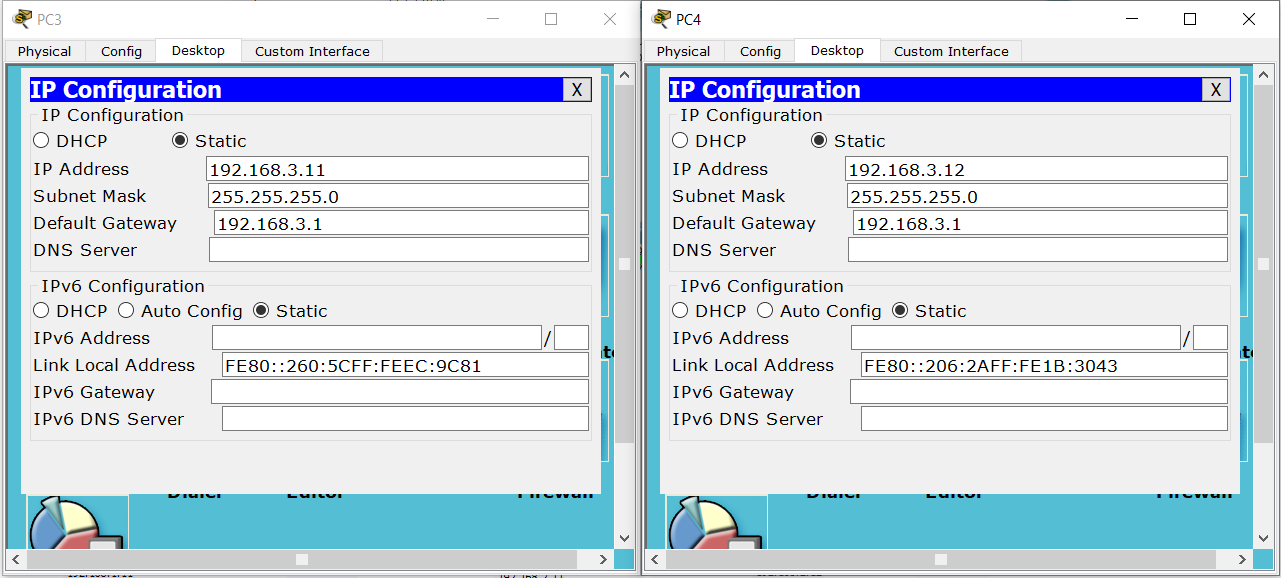
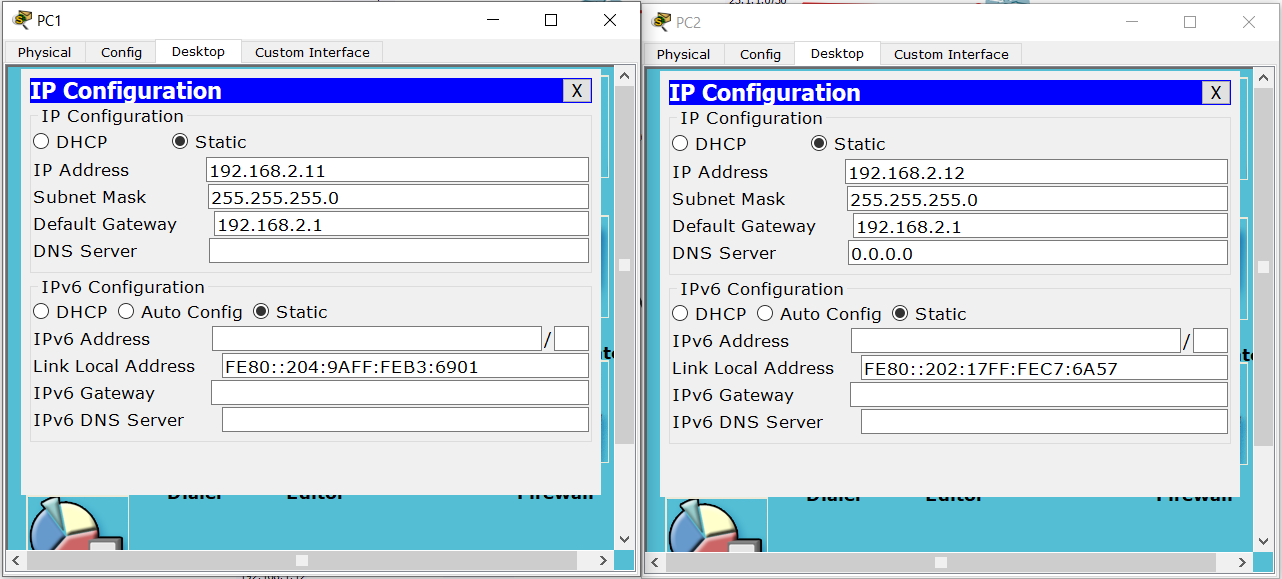
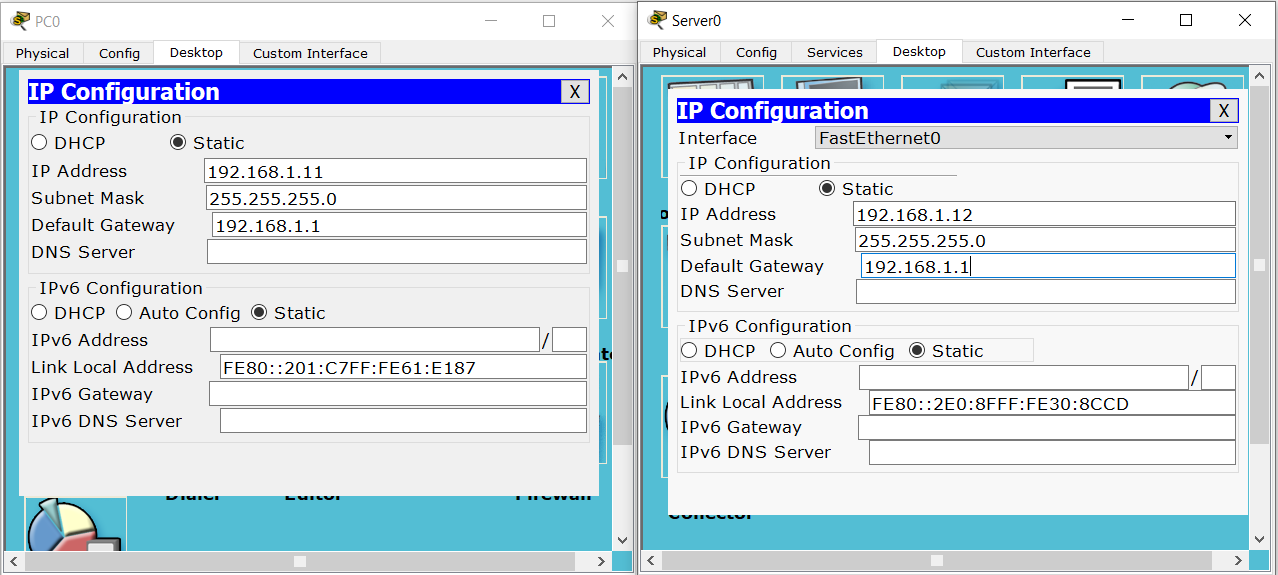
We are asked to configure ACLs on a given network technology where the components are connected using OSPF on different areas. Once we place the components on the workspaces and connect them with cables we will get a network topology as shown below.

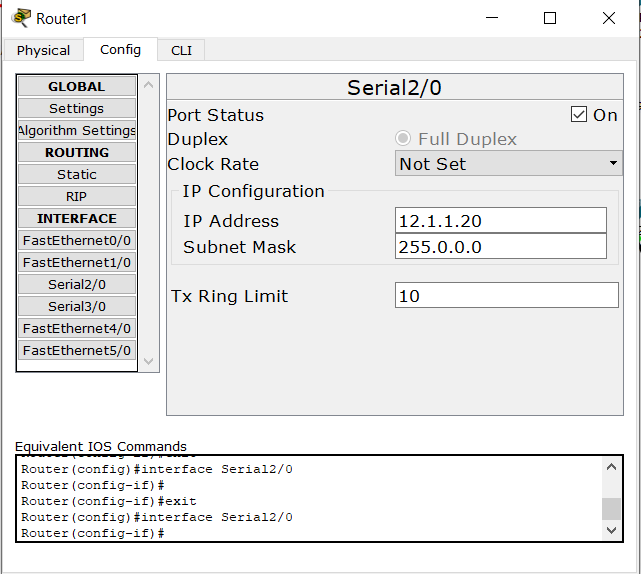
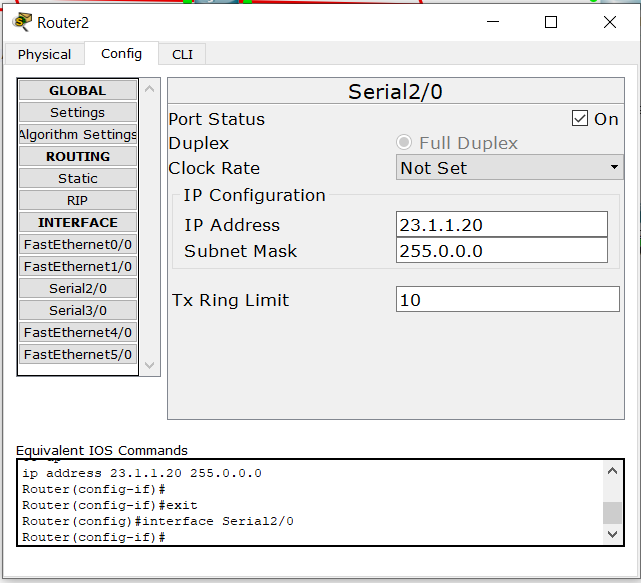


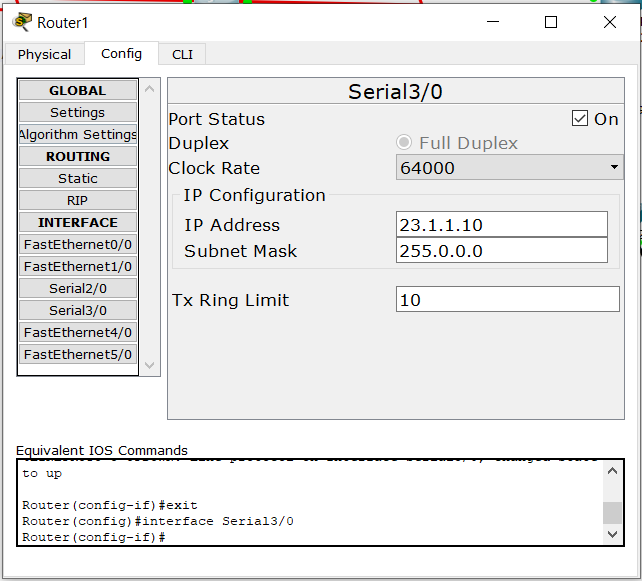
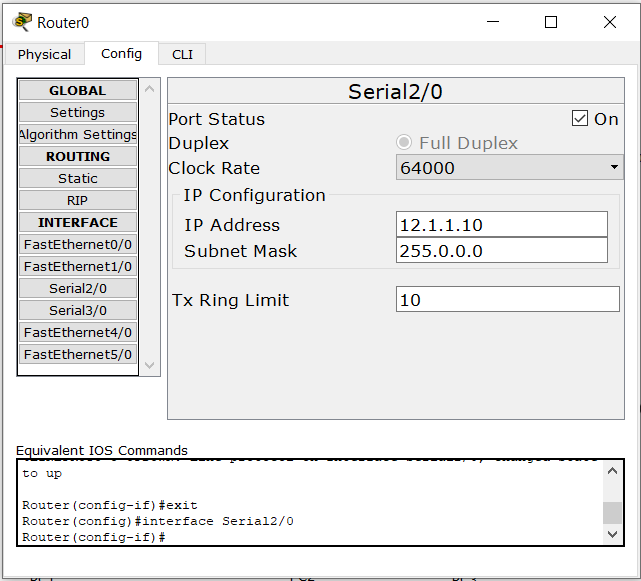
We can then configure our routers since they are going to bed as a default gateway for our components on each network.



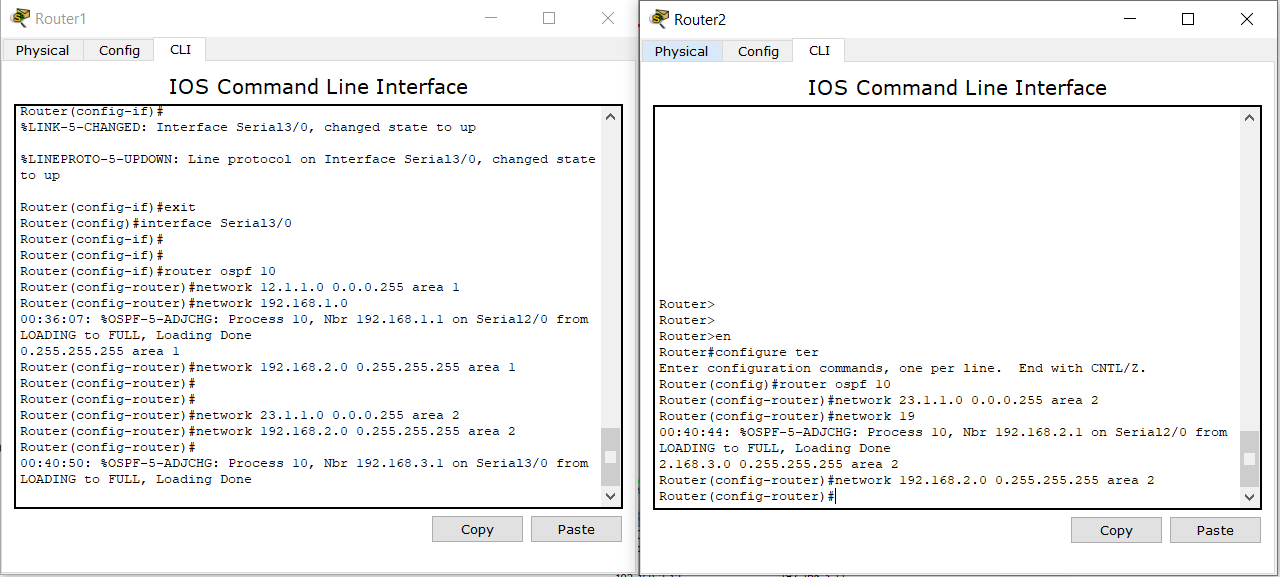
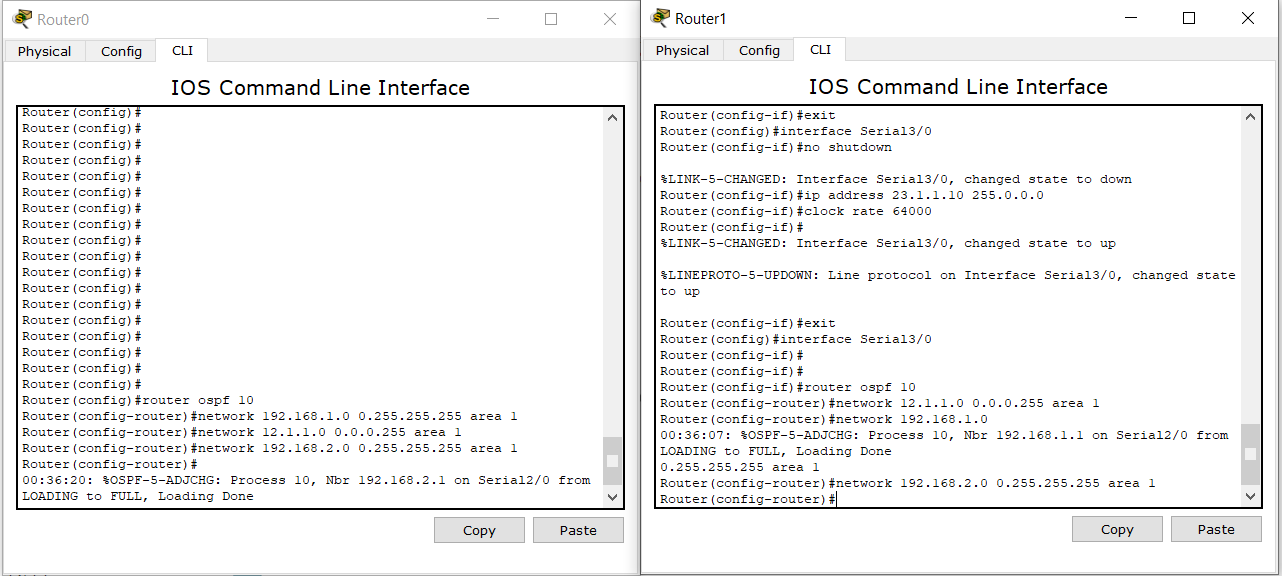
Once we configure our routers then we can move our individual computers and configure them accordingly their IP address, subnet mask and default gateway address in this case the IP address of our router.



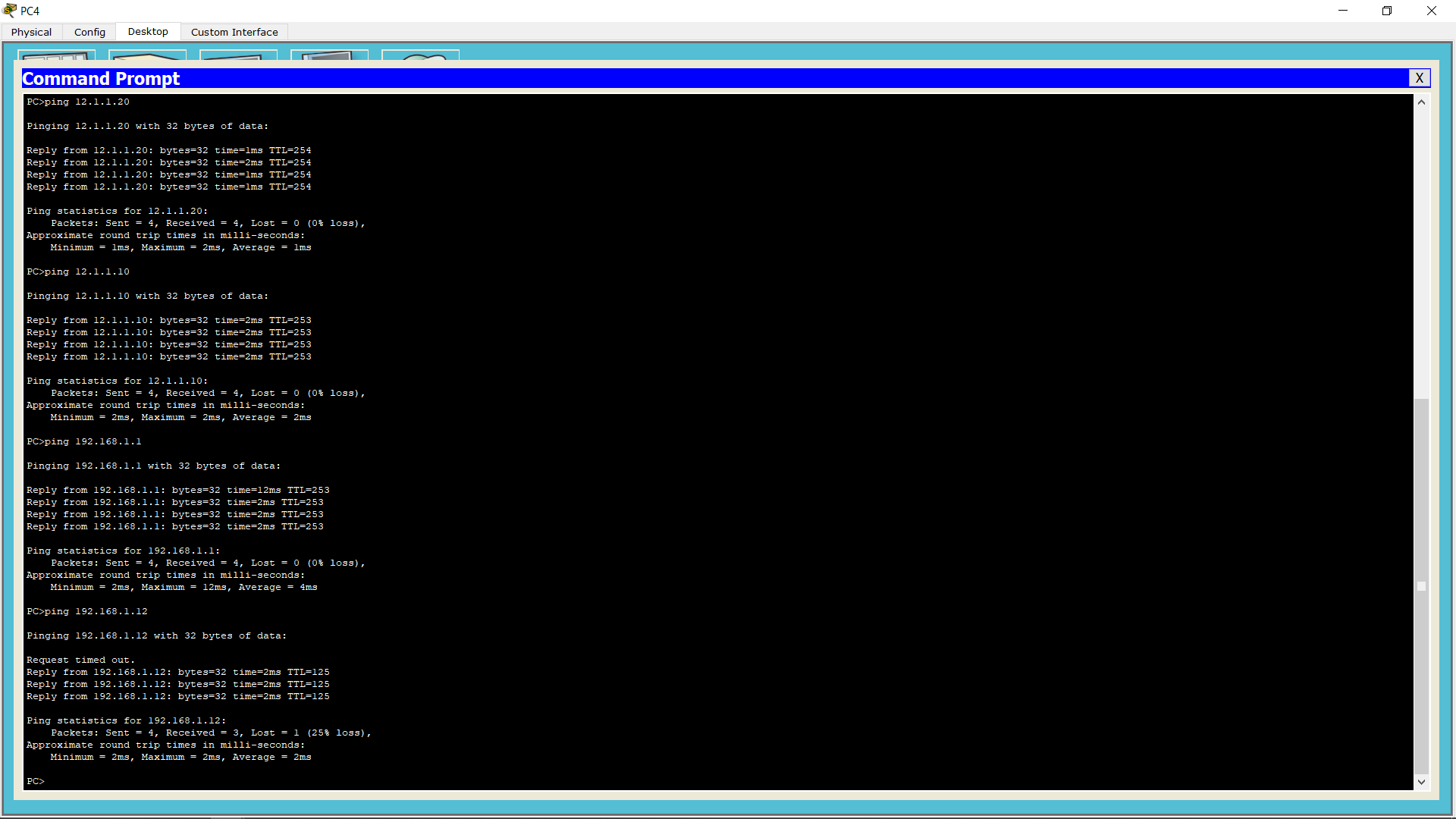
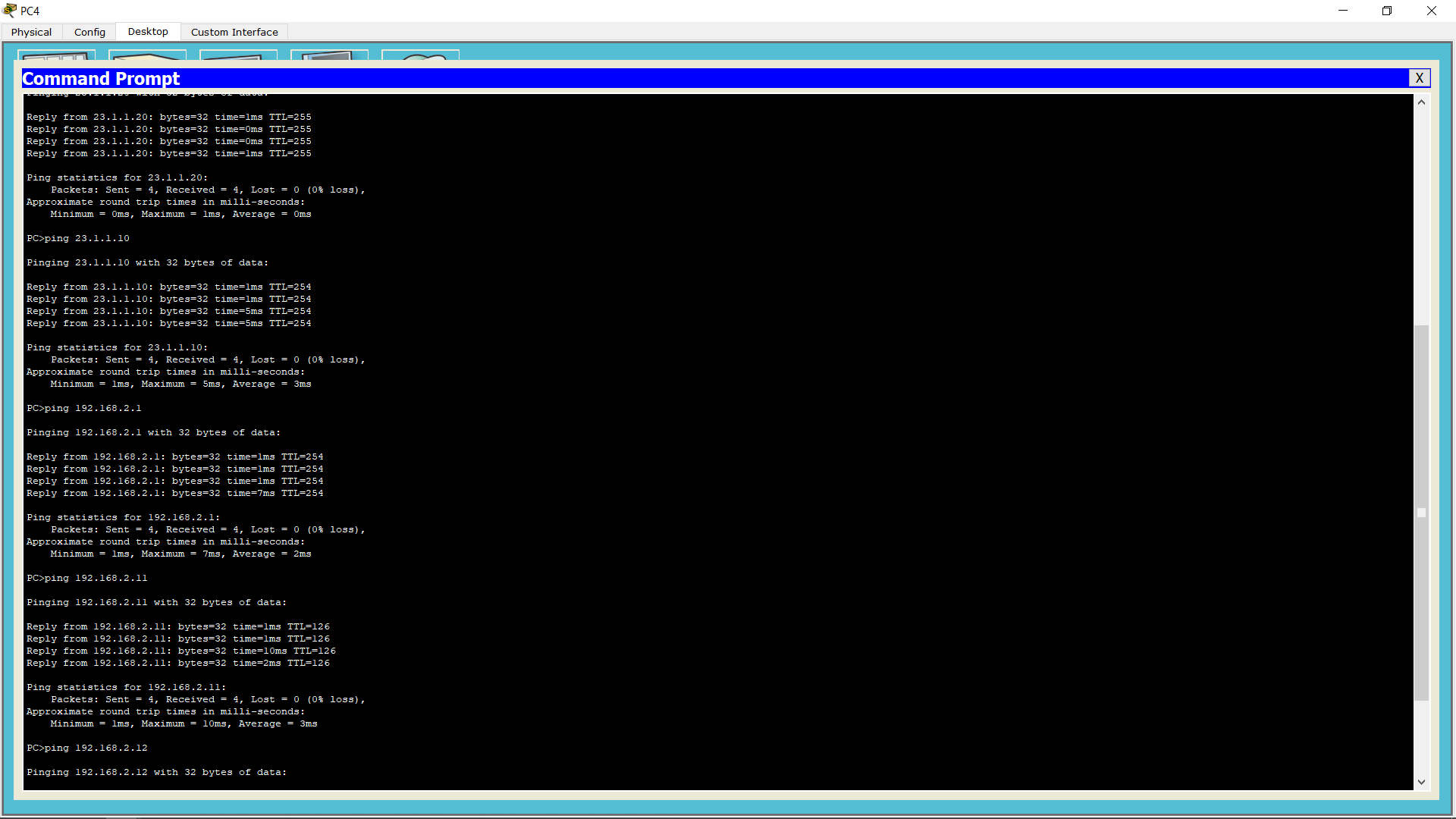
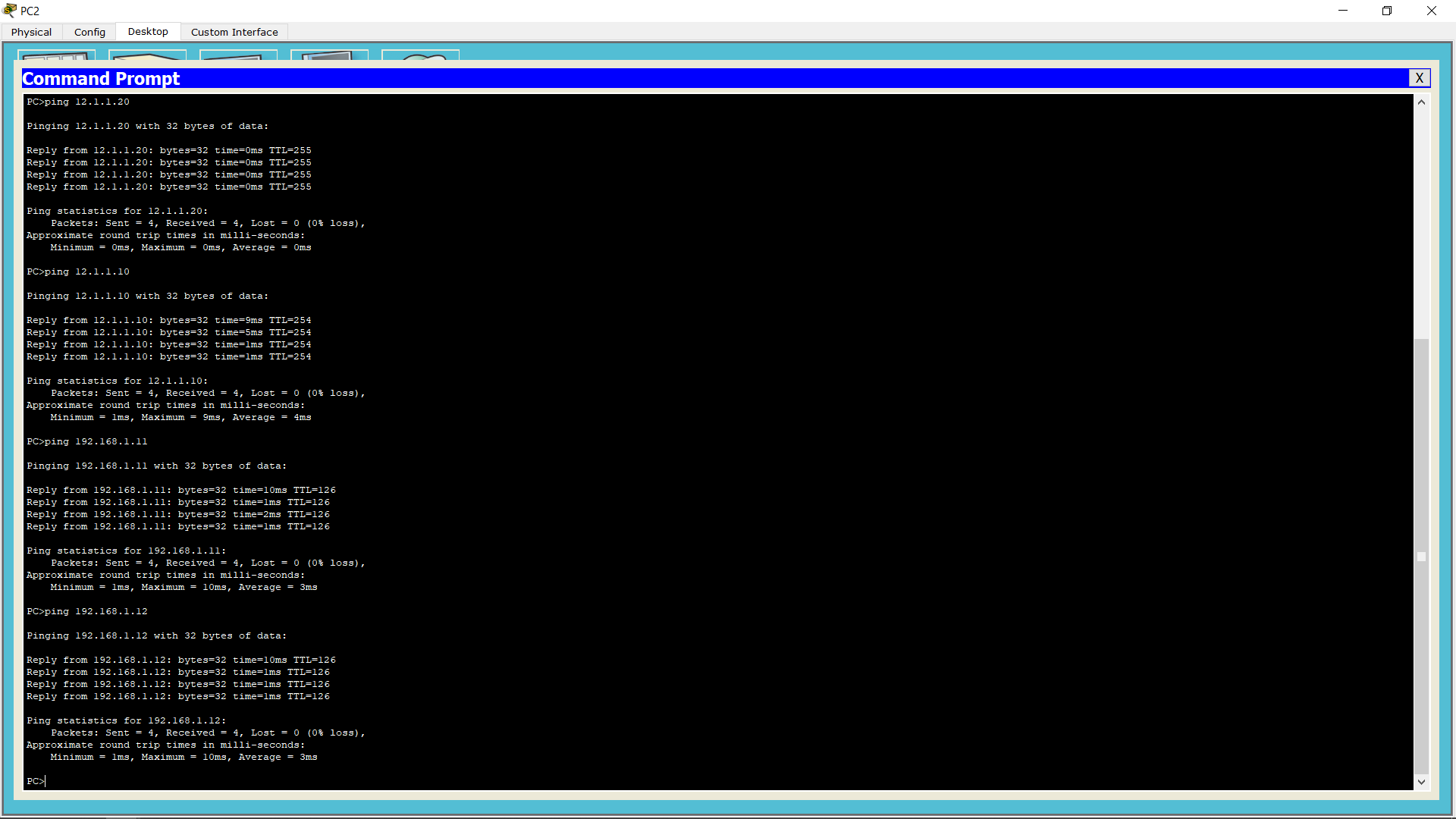
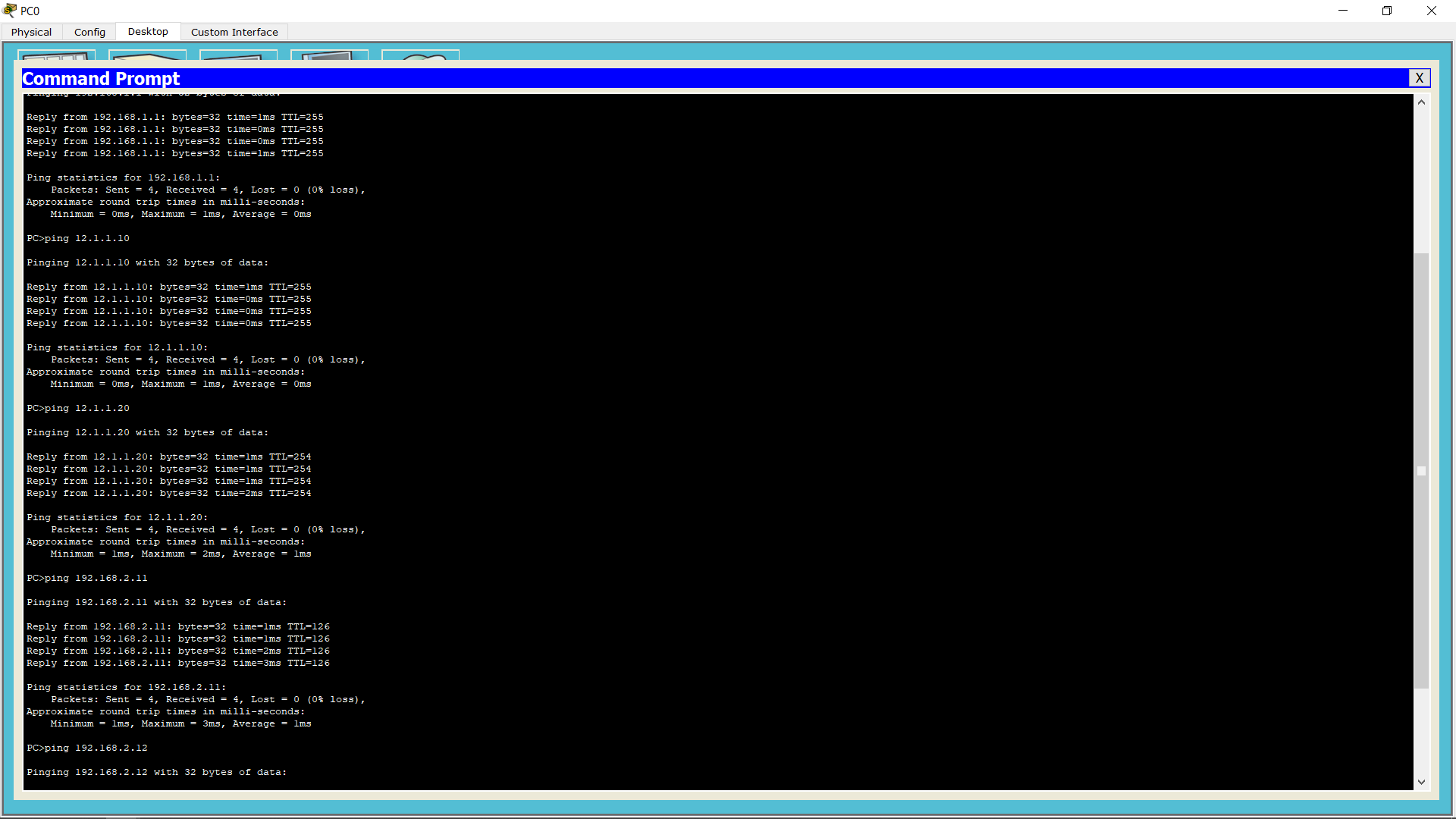
The next thing to do is to configure the serial ports of our routers. These ports are connected to the outside internet or WAN rather than LAN which give internet access to the computers connected with them.



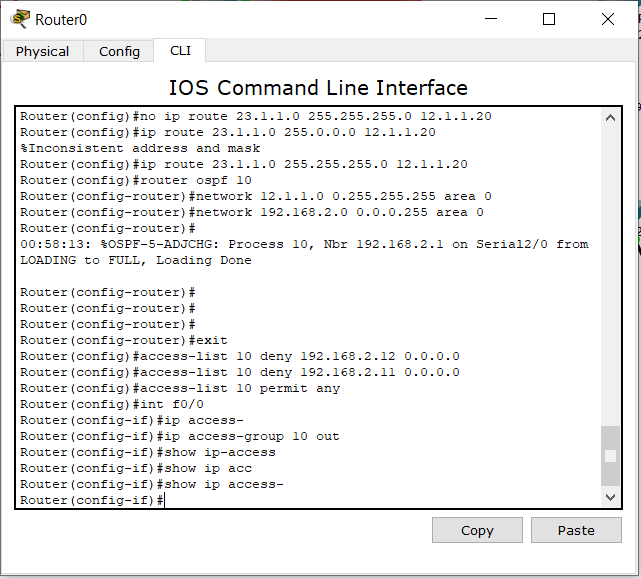
To configure our OSPF on each individual router, OSPF stands for (Open Shortest Path First) which is a network configuration method that allows nodes to communicate with each other from different areas or regions.



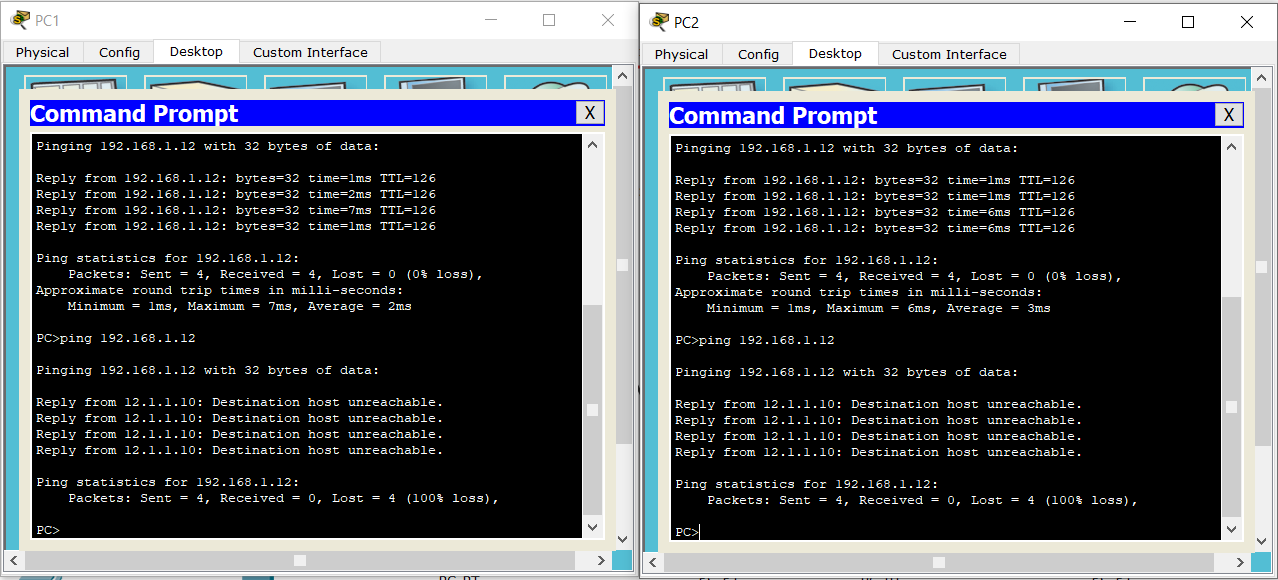
Once we finish configuring our OSPF settings then we can start pinging and test if the connection works as expected. For this we can go to every computer and test the connection from the nearest node to the farthest node. This will help to be sure that each node on the network is functioning as expected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

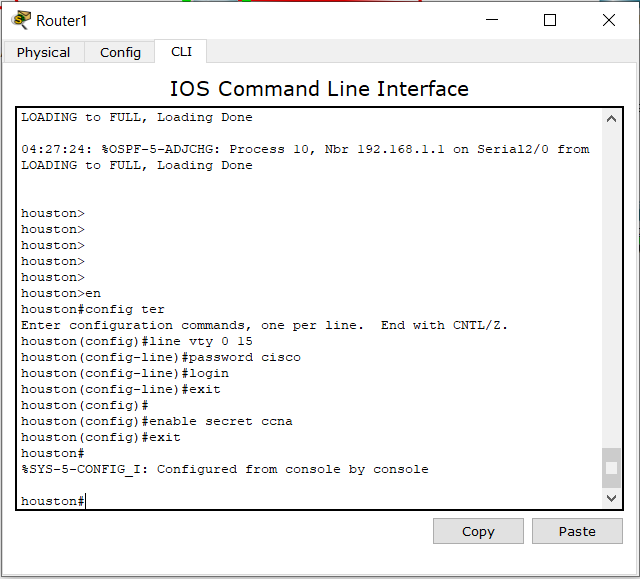


Once we make sure that all the nodes are connected and there is no issue then we can continue configuring the ACLs on the router where we want to permit and deny network traffic. According to the requirement we want to deny traffic coming from the Houston network block to the Newyork network block, to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

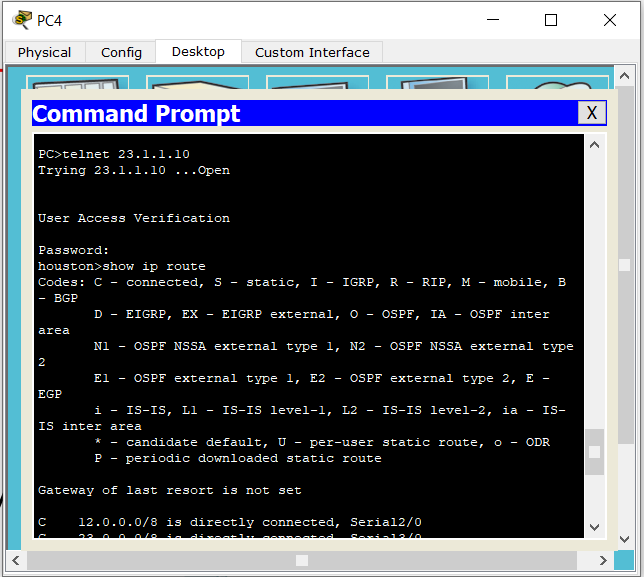


Once we configure the ACL we can go ahead and try to ping using computers from the Houston block to the computers inside the Newyork block. The ping result shows that the destination host is unreachable.

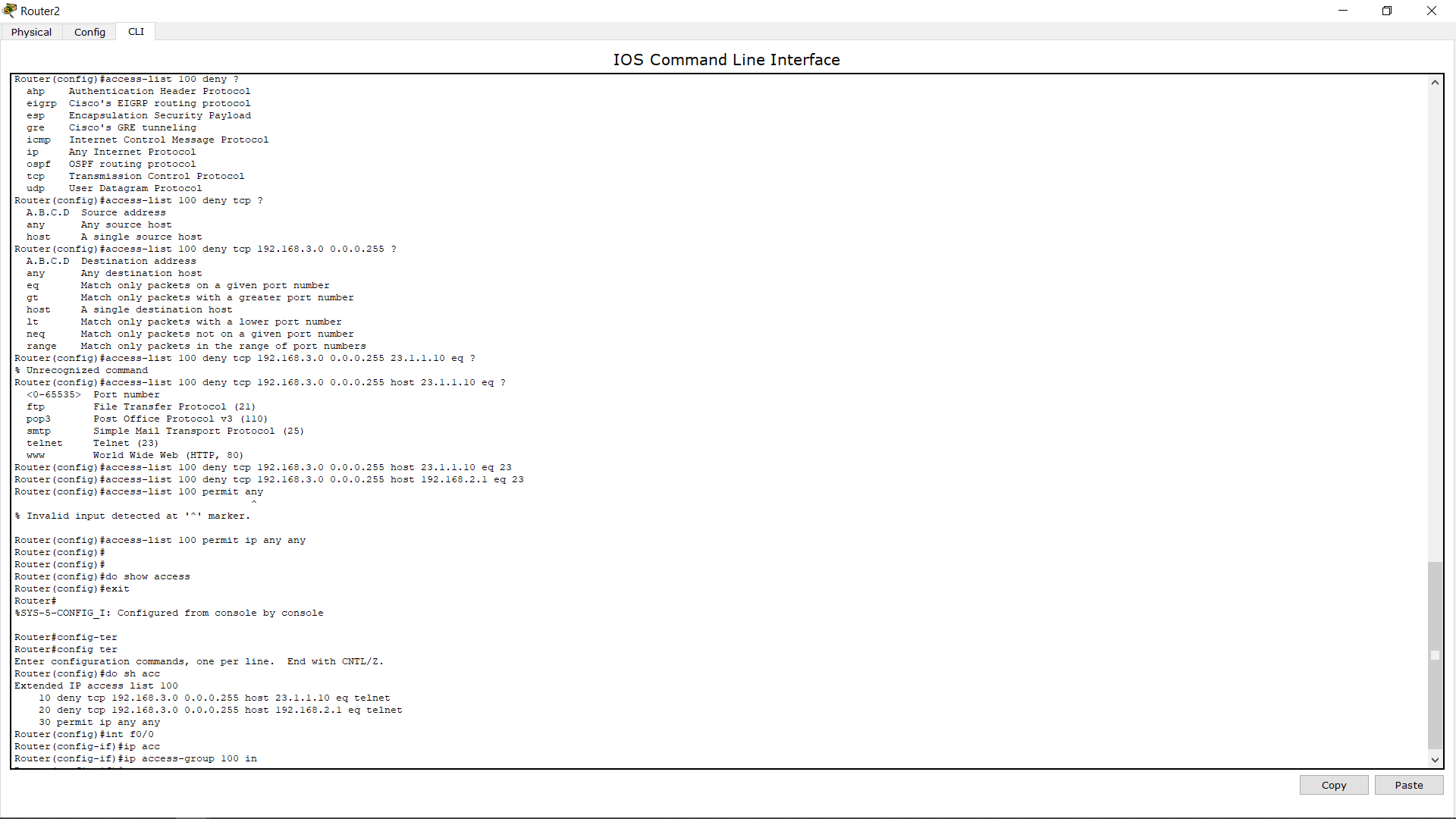


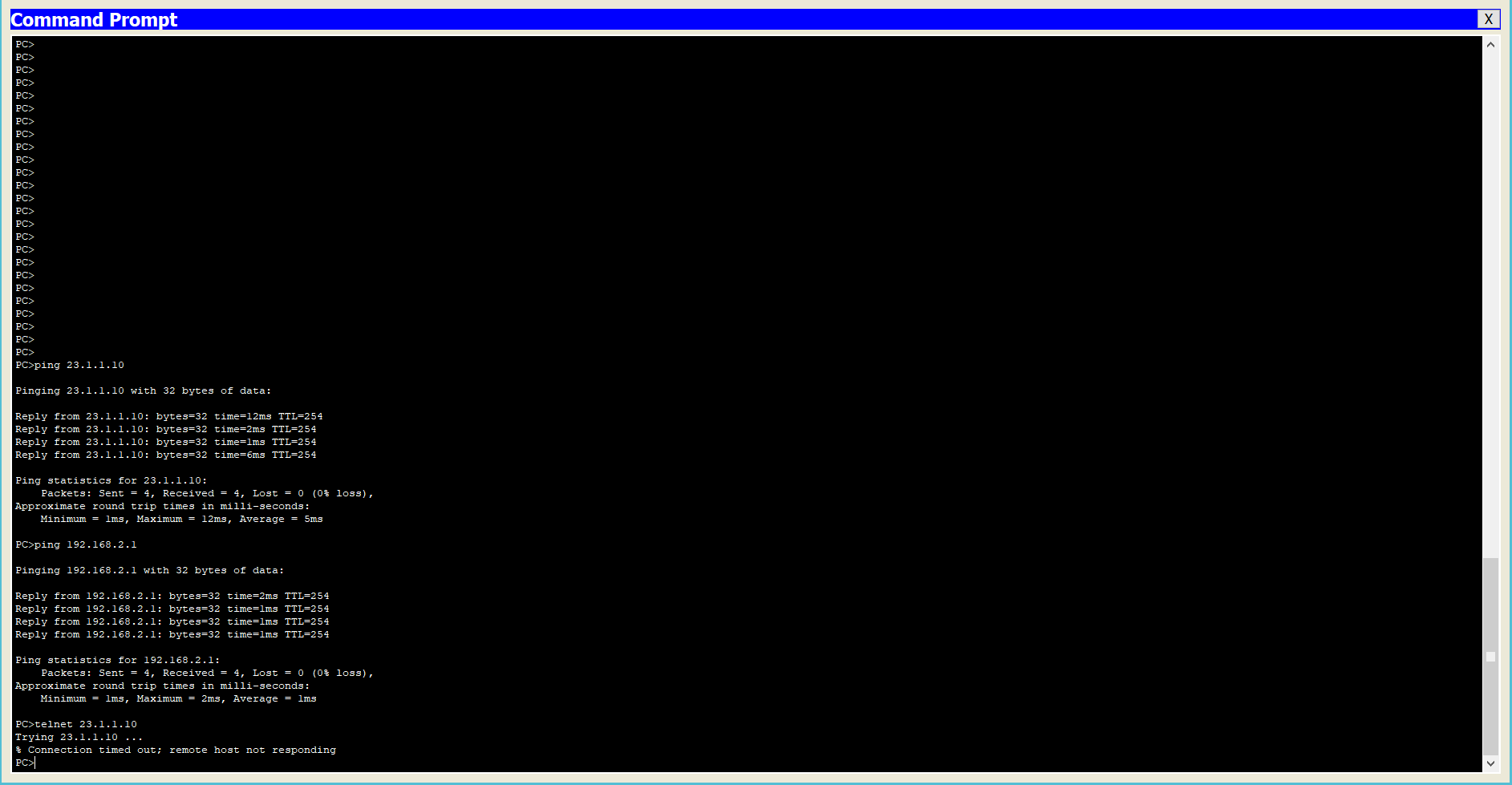
The second requirement is configure the telnet service on the Houston router and block any attempt to access telnet from the Chicago block, and configure the Newyork router telnet service and give access to the computers on the Chicago block to get telnet service on the Newyork router, to achieve this 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.

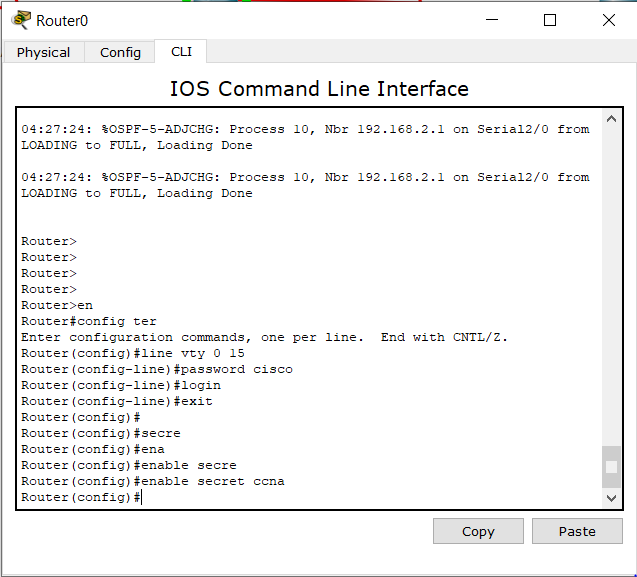


Then we can configure our extended ACL to filter traffic and block telnet service from being accessed for computers from the Chicago block.



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.

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.

