**Table of Contents**

[**Introduction:** 2](#_Toc165163485)

[**Setting up the environment:** 3](#_Toc165163486)

[**Codes:** 5](#_Toc165163487)

[**1.** **Master node:** 5](#_Toc165163488)

[**2.** **Worker node:** 6](#_Toc165163489)

[**3.** **Client GUI:** 8](#_Toc165163490)

[**4.** **Images functions middleware:** 11](#_Toc165163491)

[**5.** **Image processing module:** 13](#_Toc165163492)

[**Analysis of the codes:** 15](#_Toc165163493)

[**1.** **Master node:** 15](#_Toc165163494)

[**2.** **Worker node:** 15](#_Toc165163495)

[**3.** **Client GUI:** 16](#_Toc165163496)

[**4.** **Images functions middleware:** 17](#_Toc165163497)

[**5.** **Image processing module:** 18](#_Toc165163498)

[**Testing:** 19](#_Toc165163499)

**Table of figures**

[Figure 1 azure resources 3](file:///C:\Users\Mohamed%20Amr\Downloads\distributed%20project\reports\Phase%202.docx#_Toc165163680)

[Figure 2 RDP download 3](file:///C:\Users\Mohamed%20Amr\Downloads\distributed%20project\reports\Phase%202.docx#_Toc165163681)

[Figure 3 inbound rules 4](file:///C:\Users\Mohamed%20Amr\Downloads\distributed%20project\reports\Phase%202.docx#_Toc165163682)

[Figure 4 pinging machines 4](file:///C:\Users\Mohamed%20Amr\Downloads\distributed%20project\reports\Phase%202.docx#_Toc165163683)

[Figure 5 testing the TCP port 5](file:///C:\Users\Mohamed%20Amr\Downloads\distributed%20project\reports\Phase%202.docx#_Toc165163684)

Phase 2

# **Introduction:**

In Phase 2 of our project, we're taking steps to build the basic functions of our Distributed Image Processing System using Cloud Computing.

Over the past 2-3 weeks, we focused on laying down the groundwork. This means implementing the essential tasks for handling images, such as adjusting colors, and applying filters. These tasks fall under "image processing", as we're primarily concerned with altering images programmatically rather than manually.

Simultaneously, we organized the cloud environment. This involves creating virtual machines and configuring the necessary infrastructure to support our image processing tasks efficiently. Our goal is to ensure that our system can handle multiple tasks simultaneously without slowing down or encountering performance issues.

A key component of this phase is developing what we call a "worker thread." This thread will manage the distribution of image processing tasks across our cloud infrastructure, ensuring that each task is executed promptly and efficiently.

Throughout this phase, we'll keep the user experience in mind. We want users to be able to upload their images easily and apply basic image processing operations without any hassle. To achieve this, we'll follow user stories that guide our development process, ensuring that our system meets the needs and expectations of its users.

By the end of Phase 2, we aim to have a solid foundation for our Distributed Image Processing System. While it may not have all the bells and whistles yet, it will be capable of reliably processing images in the cloud, setting the stage for more advanced features in the future.

# **Setting up the environment:**

We used Microsoft azure to setup the cloud environment and creating the virtual machines.

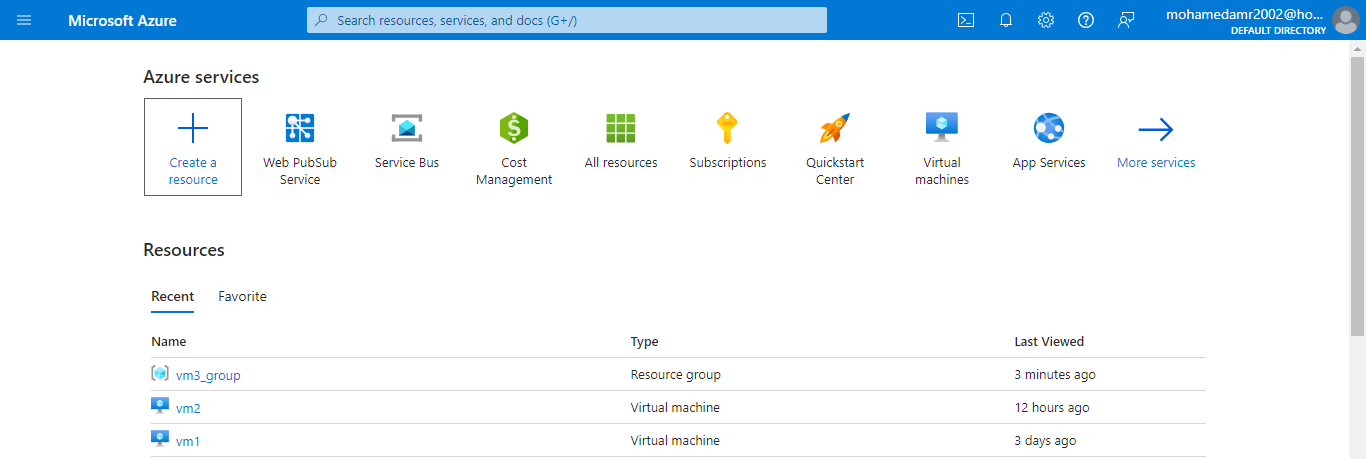
Firstly, we created two virtual machines one for the master node and the other for the worker node:

Figure azure resources

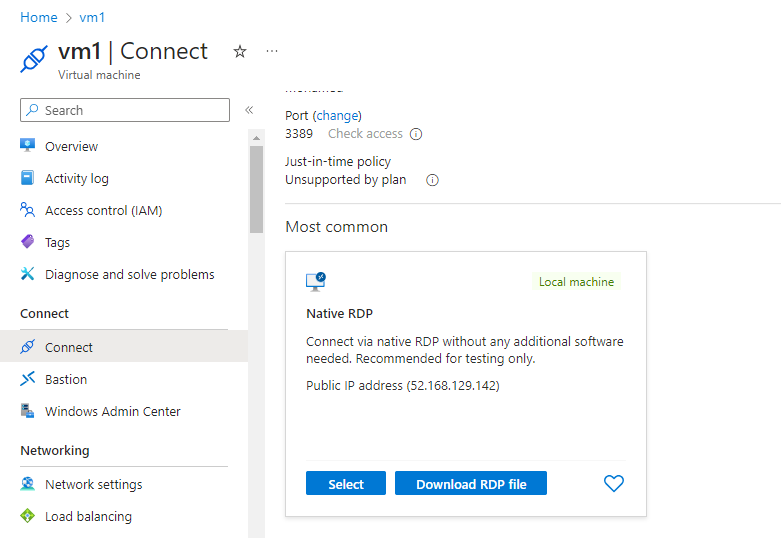
Then we downloaded RDP files for both machine to start them

Figure RDP download

Then we added inbound ICMP rules to ping the machines, then we added inbound TCP rule to open the port for accepting TCP messages form another computers in both machine, here is an example of one of them:

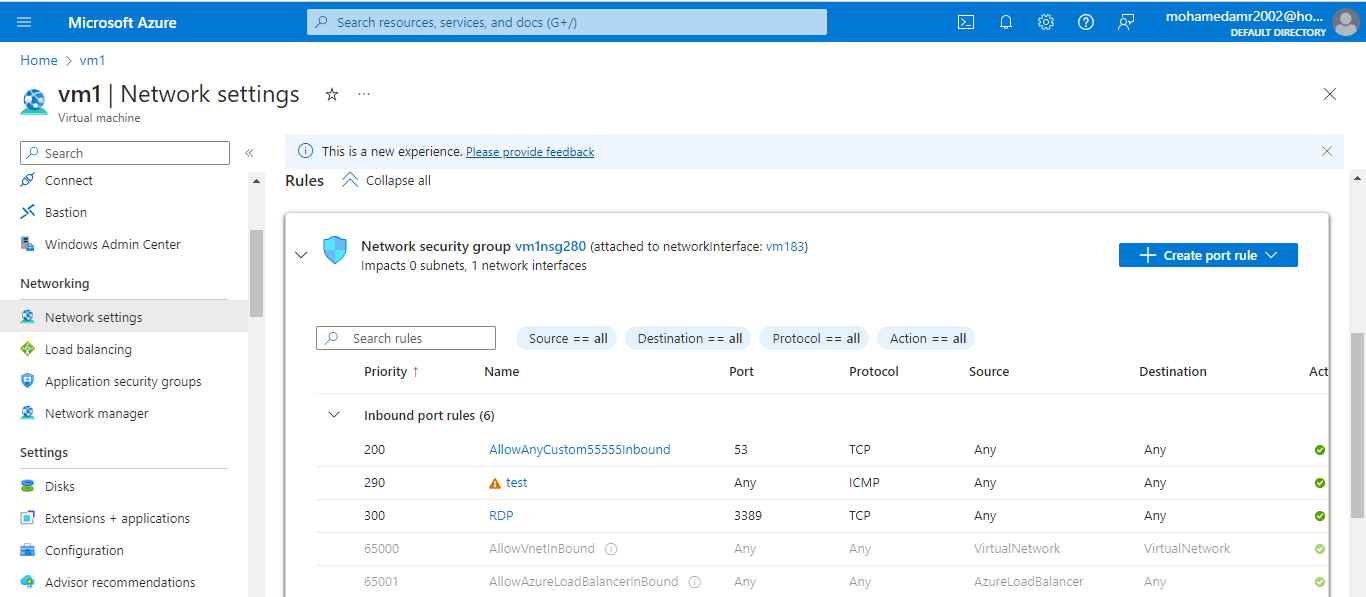


Figure inbound rules

Then we pinged the IP address of the machine to test the connectivity between two pcs

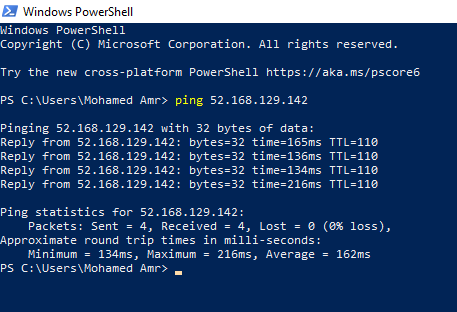


Figure pinging machines

Then we used zenmap application to test if the TCP port is opened for sending and receiving messages

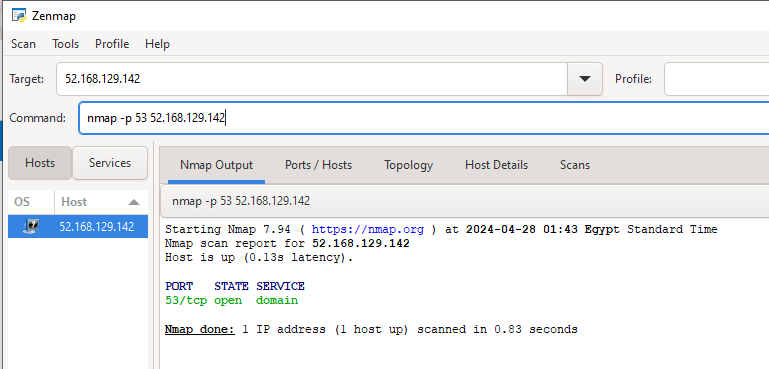


Figure testing the TCP port

After that we had two virtual machines ready for sending and receiving messages

# **Codes:**

## **Master node:**

import socket

import threading

from imageFunctionsMiddleware import \*

def recieveAndSendClient():

    server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    host = 'localhost'

    port = 12348

    server\_socket.bind((host, port))

    server\_socket.listen(5)

    print(f"Server listening on {host}:{port}")

    while True:

        client\_socket, addr = server\_socket.accept()

        operation=client\_socket.recv(2).decode('utf-8')

        imageBytes,\_=receive\_image(client\_socket)

        client\_thread = threading.Thread(target=sendImageToWorker, args=("localhost",12345,client\_socket,imageBytes,operation,addr))

        client\_thread.start()

def sendImageToWorker(server\_public\_ip,port,clientsockloggedonmaster,image\_bytes,operation,addr):

    print(f"Connection from: {addr}")

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

    client\_socket.connect((server\_public\_ip, port))

    segments = split\_image(5, image\_bytes)

    processed\_segments\_bytes = []

    for segment in segments:

        client\_socket.send(operation.encode('utf-8'))

        send\_image\_segments(client\_socket, segment)

        processed\_segment\_bytes, \_ = receive\_image(client\_socket)

        display\_image\_from\_bytes(processed\_segment\_bytes)

        processed\_segments\_bytes.append(processed\_segment\_bytes)

    combined\_image\_path = combine\_segments\_to\_bytes(processed\_segments\_bytes)

    display\_image\_from\_bytes(combined\_image\_path)

    send\_image\_segments(clientsockloggedonmaster,combined\_image\_path)

    client\_socket.close()

if \_\_name\_\_ == "\_\_main\_\_":

    recieveAndSendClient()

## **Worker node:**

import socket

import threading

from imageFunctionsMiddleware import \*

from imageProcessingModule import \*

def handle\_client(client\_socket, addr):

    print(f"Connection from: {addr}")

    while True:

        message=client\_socket.recv(2).decode('utf-8')

        if message == "":

            continue

        elif message=="q":

            print("client disconnected")

            break

        else:

            try:

                image\_bytes,length = receive\_image(client\_socket)

                if image\_bytes is not None:

                    if message == "gr":

                        processed\_image\_bytes = greyFilter(image\_bytes)

                        send\_image\_knownbytes(client\_socket, (processed\_image\_bytes, len(processed\_image\_bytes)))

                    elif message == "ed":

                        edges\_bytes = edgeDetection(image\_bytes)

                        send\_image\_knownbytes(client\_socket, (edges\_bytes, len(edges\_bytes)))

                    elif message == "fl":

                        filtered\_image\_bytes = imageFiltering(image\_bytes)

                        send\_image\_knownbytes(client\_socket, (filtered\_image\_bytes, len(filtered\_image\_bytes)))

                    else:

                        print(f"Unknown message: {message} enter right choice")

                        continue

            except Exception as e:

                print(f"Error receiving image: {e}")

                break

    client\_socket.close()

def main():

    server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    host = 'localhost'

    port = 12345

    server\_socket.bind((host, port))

    server\_socket.listen(5)

    print(f"Server listening on {host}:{port}")

    while True:

        client\_socket, addr = server\_socket.accept()

        client\_thread = threading.Thread(target=handle\_client, args=(client\_socket, addr))

        client\_thread.start()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

## **Client GUI:**

import tkinter as tk

from tkinter import filedialog

from PIL import Image, ImageTk

import io

import socket

import numpy as np

import threading

class ScrollableImageFrame(tk.Frame):

    def \_\_init\_\_(self, root):

        super().\_\_init\_\_(root)

        self.canvas = tk.Canvas(self)

        self.scrollbar = tk.Scrollbar(self, orient="vertical", command=self.canvas.yview)

        self.scrollable\_frame = tk.Frame(self.canvas)

        self.scrollable\_frame.bind(

            "<Configure>",

            lambda e: self.canvas.configure(

                scrollregion=self.canvas.bbox("all")

            )

        )

        self.canvas.create\_window((0, 0), window=self.scrollable\_frame, anchor="nw")

        self.canvas.configure(yscrollcommand=self.scrollbar.set)

        self.canvas.pack(side="left", fill="both", expand=True)

        self.scrollbar.pack(side="right", fill="y")

    def add\_image(self, image, max\_width=300, max\_height=300):

        width, height = image.size

        aspect\_ratio = width / height

        if width > max\_width or height > max\_height:

            if aspect\_ratio > 1:

                new\_width = max\_width

                new\_height = int(max\_width / aspect\_ratio)

            else:

                new\_height = max\_height

                new\_width = int(max\_height \* aspect\_ratio)

            image = image.resize((new\_width, new\_height))

        photo = ImageTk.PhotoImage(image)

        label = tk.Label(self.scrollable\_frame, image=photo)

        label.image = photo

        label.pack(pady=5)

class ImageConverterApp:

    def \_\_init\_\_(self, root):

        self.root = root

        self.root.title("Image Converter")

        self.root.geometry("500x400")

        self.root.pack\_propagate(True)

        self.image\_bytes = None

        self.image\_label = tk.Label(root)

        self.option\_var = tk.StringVar()

        self.option\_var.set("grey filter")

        self.imgPath=None

        self.upload\_button = tk.Button(root, text="Upload Photo", command=self.upload\_image)

        self.option\_menu = tk.OptionMenu(root, self.option\_var, "grey filter", "edge detection", "color manipulation")

        self.convert\_button = tk.Button(root, text="Convert", command=self.convert\_image\_thread)

        self.upload\_button.pack()

        self.option\_menu.pack()

        self.convert\_button.pack()

        self.image\_label.pack()

        self.scrollable\_frame = ScrollableImageFrame(root)

        self.scrollable\_frame.pack(side="top", fill="both", expand=True)

    def upload\_image(self):

        file\_path = filedialog.askopenfilename()

        if file\_path:

            image = Image.open(file\_path)

            self.imgPath=file\_path

            self.scrollable\_frame.add\_image(image)

    def resize\_photo(self, photo, width, height):

        return photo.subsample(int(photo.width() / width), int(photo.height() / height))

    def convert\_to\_bytes(self, image):

        img\_byte\_array = io.BytesIO()

        image.save(img\_byte\_array, format=image.format)

        return img\_byte\_array.getvalue()

    def bytes\_to\_image(self,image\_bytes):

        image\_stream = io.BytesIO(image\_bytes)

        image = Image.open(image\_stream)

        return image

    def send\_image(self, conn, imagePath):

        with open(imagePath, 'rb') as f:

            image\_bytes = f.read()

        conn.sendall(len(image\_bytes).to\_bytes(4, byteorder='big'))

        conn.sendall(image\_bytes)

    def receive\_image(self,conn):

        length = int.from\_bytes(conn.recv(4), byteorder='big')

        if length !=0:

            image\_bytes = b''

            while len(image\_bytes) < length:

                data = conn.recv(length - len(image\_bytes))

                if not data:

                    break

                image\_bytes += data

            return image\_bytes,length

    def display\_image\_from\_bytes(self,image\_bytes):

        image\_stream = io.BytesIO(image\_bytes)

        image = Image.open(image\_stream)

        image.show()

    def convert\_image\_thread(self):

        threading.Thread(target=self.convert\_image).start()

    def convert\_image(self):

        processedImages=[]

        path=self.imgPath

        option = self.option\_var.get()

        if option=="grey filter":

            option="gr"

        elif option=="edge detection":

            option="ed"

        elif option=="color manipulation":

            option="fl"

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

        server\_public\_ip = 'localhost'

        port = 12348

        client\_socket.connect((server\_public\_ip, port))

        client\_socket.send(option.encode('utf-8'))

        self.send\_image(client\_socket,path)

        imageBytes,\_=self.receive\_image(client\_socket)

        imageBytes=self.bytes\_to\_image(imageBytes)

        processedImages.append(imageBytes)

        for x in processedImages:

            self.scrollable\_frame.add\_image(x)

if \_\_name\_\_ == "\_\_main\_\_":

    root = tk.Tk()

    app = ImageConverterApp(root)

    root.mainloop()

## **Images functions middleware:**

import io

import matplotlib.pyplot as plt

import numpy as np

from PIL import Image

def split\_image(num\_segments, image\_bytes):

    img = np.array(Image.open(io.BytesIO(image\_bytes)))

    height, width, \_ = img.shape

    segment\_height = height // num\_segments

    segments = []

    for i in range(num\_segments):

        start = i \* segment\_height

        end = start + segment\_height

        if i == num\_segments - 1:

            end = height

        segment = img[start:end, :, :]

        segment\_bytes = io.BytesIO()

        Image.fromarray(segment).save(segment\_bytes, format='JPEG')

        segment\_bytes.seek(0)

        segments.append(segment\_bytes.read())

    return segments

def combine\_segments\_to\_bytes(segments):

    first\_segment = Image.open(io.BytesIO(segments[0]))

    width, height = first\_segment.size

    combined\_image = Image.new("RGB", (width, height \* len(segments)))

    for i, segment\_bytes in enumerate(segments):

        segment = Image.open(io.BytesIO(segment\_bytes))

        combined\_image.paste(segment, (0, i \* height))

    combined\_image\_bytes = io.BytesIO()

    combined\_image.save(combined\_image\_bytes, format='JPEG')

    combined\_image\_bytes.seek(0)

    return combined\_image\_bytes.read()

def display\_image\_from\_bytes(image\_bytes):

    image\_stream = io.BytesIO(image\_bytes)

    image = Image.open(image\_stream)

    image.show()

def receive\_image(conn):

    length = int.from\_bytes(conn.recv(4), byteorder='big')

    if length !=0:

        image\_bytes = b''

        while len(image\_bytes) < length:

            data = conn.recv(length - len(image\_bytes))

            if not data:

                break

            image\_bytes += data

        return image\_bytes,length

def send\_image(conn, imagePath):

    with open(imagePath, 'rb') as f:

        image\_bytes = f.read()

    conn.sendall(len(image\_bytes).to\_bytes(4, byteorder='big'))

    conn.sendall(image\_bytes)

def send\_image\_segments(conn, image\_bytes):

    conn.sendall(len(image\_bytes).to\_bytes(4, byteorder='big'))

    conn.sendall(image\_bytes)

def send\_image\_knownbytes(conn, image):

    image\_bytes = image[1]

    conn.sendall(image\_bytes.to\_bytes(4, byteorder='big'))

    conn.sendall(image[0])

## **Image processing module:**

import io

import matplotlib.pyplot as plt

import numpy as np

from PIL import Image

def split\_image(num\_segments, image\_bytes):

    img = np.array(Image.open(io.BytesIO(image\_bytes)))

    height, width, \_ = img.shape

    segment\_height = height // num\_segments

    segments = []

    for i in range(num\_segments):

        start = i \* segment\_height

        end = start + segment\_height

        if i == num\_segments - 1:

            end = height

        segment = img[start:end, :, :]

        segment\_bytes = io.BytesIO()

        Image.fromarray(segment).save(segment\_bytes, format='JPEG')

        segment\_bytes.seek(0)

        segments.append(segment\_bytes.read())

    return segments

def combine\_segments\_to\_bytes(segments):

    first\_segment = Image.open(io.BytesIO(segments[0]))

    width, height = first\_segment.size

    combined\_image = Image.new("RGB", (width, height \* len(segments)))

    for i, segment\_bytes in enumerate(segments):

        segment = Image.open(io.BytesIO(segment\_bytes))

        combined\_image.paste(segment, (0, i \* height))

    combined\_image\_bytes = io.BytesIO()

    combined\_image.save(combined\_image\_bytes, format='JPEG')

    combined\_image\_bytes.seek(0)

    return combined\_image\_bytes.read()

def display\_image\_from\_bytes(image\_bytes):

    image\_stream = io.BytesIO(image\_bytes)

    image = Image.open(image\_stream)

    image.show()

def receive\_image(conn):

    length = int.from\_bytes(conn.recv(4), byteorder='big')

    if length !=0:

        image\_bytes = b''

        while len(image\_bytes) < length:

            data = conn.recv(length - len(image\_bytes))

            if not data:

                break

            image\_bytes += data

        return image\_bytes,length

def send\_image(conn, imagePath):

    with open(imagePath, 'rb') as f:

        image\_bytes = f.read()

    conn.sendall(len(image\_bytes).to\_bytes(4, byteorder='big'))

    conn.sendall(image\_bytes)

def send\_image\_segments(conn, image\_bytes):

    conn.sendall(len(image\_bytes).to\_bytes(4, byteorder='big'))

    conn.sendall(image\_bytes)

def send\_image\_knownbytes(conn, image):

    image\_bytes = image[1]

    conn.sendall(image\_bytes.to\_bytes(4, byteorder='big'))

    conn.sendall(image[0])

# **Analysis of the codes:**

## **Master node:**

This code sets up a basic server-client architecture for image processing using sockets and threading.

* **Imports**:
  + **socket**: This module provides access to the BSD socket interface.
  + **threading**: This module provides high-level threading interface.
* **Function Definitions**:
  + **recieveAndSendClient()**: This function sets up a server socket, binds it to localhost on port 12348, and listens for incoming connections. When a client connects, it receives an operation code (2 bytes), and an image. Then, it spawns a new thread (**sendImageToWorker**) to handle the processing of the image.
  + **sendImageToWorker(server\_public\_ip, port, clientsockloggedonmaster, image\_bytes, operation, addr)**: This function handles the processing of the image. It connects to another server (a worker node) using the provided IP address and port. It then splits the image into segments, sends each segment along with the operation code to the worker server, receives the processed segments back, combines them, and sends the combined image back to the client.
* **Main Block**:
  + **if \_\_name\_\_ == "\_\_main\_\_":**: This block is the entry point of the script. It calls the **recieveAndSendClient()** function to start the server.
* **Analysis**:
  + The code utilizes threading to handle multiple client connections simultaneously.
  + It follows a simple protocol where the client sends an operation code (like "rotate", "resize", etc.) followed by an image to the server.
  + The server then forwards the image segments to a worker node for processing.
  + Once processed, the server receives the processed image segments from the worker, combines them, and sends the final image back to the client.

## **Worker node:**

This code establishes a server that listens for incoming client connections and processes image-related requests.

* **Imports**:
  + **socket**: Provides access to socket interface functionalities.
  + **threading**: Enables high-level threading capabilities.
  + **imageFunctionsMiddleware**: contains functions for image manipulation.
  + **imageProcessingModule**: includes functions for image processing tasks like grey filtering, edge detection, etc.
* **Function Definitions**:
  + **handle\_client(client\_socket, addr)**: This function is responsible for handling individual client connections. It continuously listens for messages from the client. Upon receiving a message, it checks for specific commands ('gr' for grey filter, 'ed' for edge detection, 'fl' for image filtering). Based on the command received, it processes the image accordingly using functions from the imported modules and sends back the processed image to the client. If the message is 'q', indicating the client wants to disconnect, the loop breaks, and the client socket is closed.
* **Main Functionality**:
  + **main()**: This function initializes the server socket, binds it to localhost on port 12345, and starts listening for incoming connections. Upon accepting a connection, it spawns a new thread to handle the client connection using the **handle\_client** function.
* **Main Block**:
  + **if \_\_name\_\_ == "\_\_main\_\_":**: This block is the entry point of the script. It calls the **main()** function to start the server.
* **Analysis**:
  + The server listens for connections on port 12345.
  + It processes incoming messages from clients to perform specific image processing tasks.
  + The server's functionality seems well-structured, with threading used to handle multiple client connections concurrently.

## **Client GUI:**

This code defines a simple GUI application using Tkinter for uploading images, converting them using image processing operations, and displaying the results.

* **Tkinter GUI Setup**:
  + **ImageConverterApp**: This class initializes the main application window (**root**) and sets up various GUI elements like buttons, labels, and an option menu using Tkinter widgets.
  + **ScrollableImageFrame**: This class extends **tk.Frame** and creates a scrollable frame to display images.
* **Image Conversion and Display**:
  + **upload\_image**: Opens a file dialog to allow the user to select an image file. Once an image is selected, it is displayed in the scrollable frame using the **add\_image** method of **ScrollableImageFrame**.
  + **convert\_to\_bytes**: Converts an image object to bytes.
  + **bytes\_to\_image**: Converts bytes back to an image object.
  + **send\_image** and **receive\_image**: These methods handle sending and receiving images over sockets.
  + **display\_image\_from\_bytes**: Displays an image from its bytes representation.
* **Image Processing**:
  + **convert\_image\_thread** and **convert\_image**: These methods handle the image processing operation selected by the user. They create a client socket, connect to a server, send the selected operation and image bytes, receive the processed image bytes, convert them back to an image object, and display them in the GUI.
* **Main Functionality**:
  + The **if \_\_name\_\_ == "\_\_main\_\_":** block initializes the Tkinter application (**root**) and starts the main event loop.
* **Analysis**:
  + The code provides a basic interface for users to upload images, select an image processing operation, and view the processed images.
  + It uses threading to avoid blocking the GUI while processing images.
  + The image processing operations (**grey filter**, **edge detection**, **color manipulation**) are selected via an option menu.

## **Images functions middleware:**

This code provides functions for splitting images into segments, combining segments back into a single image, displaying images from bytes, and sending/receiving images over a socket connection.

* **split\_image(num\_segments, image\_bytes)**:
  + This function takes the number of segments desired and an image in bytes format.
  + It first converts the image bytes into a NumPy array using PIL's **Image.open** and **io.BytesIO**.
  + Then, it calculates the segment height based on the number of segments and splits the image vertically into segments.
  + Each segment is converted back to image bytes using PIL's **Image.fromarray** and **io.BytesIO**.
  + Finally, it returns a list of image bytes representing each segment.
* **combine\_segments\_to\_bytes(segments)**:
  + This function takes a list of image segments in bytes format.
  + It combines the segments into a single image vertically using PIL's **Image.new**, **Image.paste**, and **io.BytesIO**.
  + The combined image is saved as JPEG format bytes and returned.
* **display\_image\_from\_bytes(image\_bytes)**:
  + This function displays an image from its bytes representation using PIL's **Image.open** and **Image.show**.
* **receive\_image(conn)**:
  + This function receives an image over a socket connection.
  + It first receives the length of the image bytes and then receives the image bytes themselves.
  + The image bytes and length are returned.
* **send\_image(conn, imagePath)**:
  + This function sends an image over a socket connection.
  + It opens the image file, reads its bytes, sends the length of the image bytes, and then sends the image bytes over the connection.
* **send\_image\_segments(conn, image\_bytes)**:
  + This function sends image segments over a socket connection.
  + It sends the length of each image segment followed by the segment bytes over the connection.
* **send\_image\_knownbytes(conn, image)**:
  + This function sends an image with known bytes over a socket connection.
  + It sends the length of the image bytes followed by the image bytes itself over the connection.

## **Image processing module:**

This code defines three image processing functions using the OpenCV library (cv2):

* **greyFilter(image\_bytes):**
  + This function takes image bytes as input.
  + It decodes the image bytes into an OpenCV image using cv2.imdecode.
  + Then, it converts the image to grayscale using cv2.cvtColor.
  + The processed grayscale image is encoded back to JPEG format bytes using cv2.imencode.
  + Finally, it returns the processed image bytes.
* **edgeDetection(image\_bytes):**
  + This function takes image bytes as input.
  + It decodes the image bytes into an OpenCV image.
  + Converts the image to grayscale.
  + Applies Canny edge detection using cv2.Canny.
  + Encodes the detected edges image to JPEG format bytes.
  + Returns the bytes of the edge-detected image.
* **imageFiltering(image\_bytes):**
  + This function takes image bytes as input.
  + It decodes the image bytes into an OpenCV image.
  + Applies Gaussian blurring to the image to reduce noise using cv2.GaussianBlur.
  + Defines a sharpening kernel and applies it to the blurred image using cv2.filter2D.
  + Encodes the sharpened image to JPEG format bytes.
  + Returns the bytes of the filtered image.

# **Testing:**

We will use manual testing for this phase

We will upload the master node on virtual machine and the worker node on another machine and will modify the hosts to the public IP addresses and ports of the machine we want to connect.

Then we will run the worker node on vm

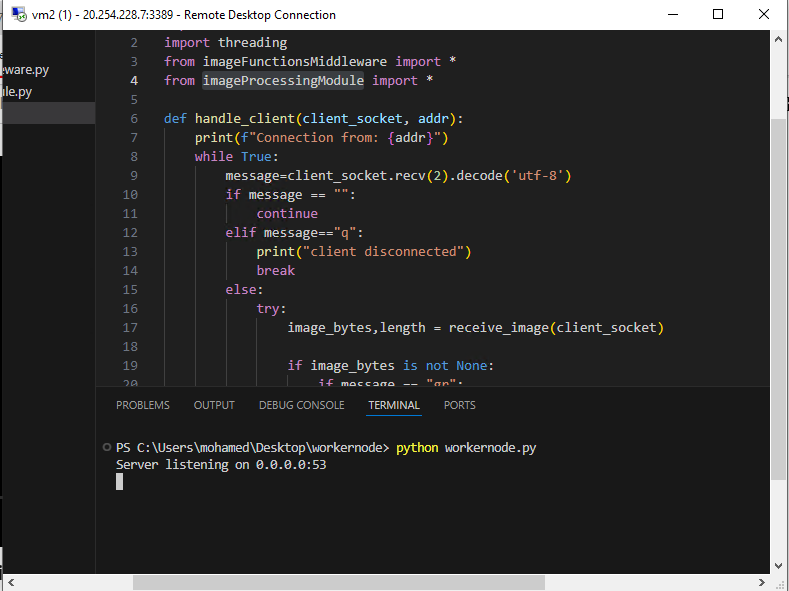


Figure running worker node on vm

And we will run the master node on another vm

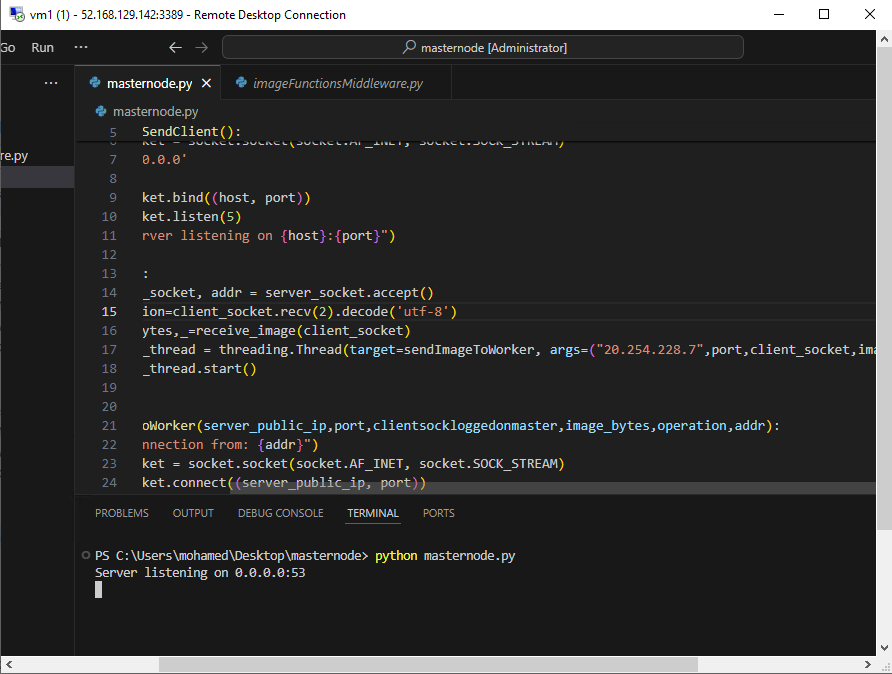


Figure running master node on another vm

Then we will run our app on our local laptop and choose the photo and click convert and wait for processing

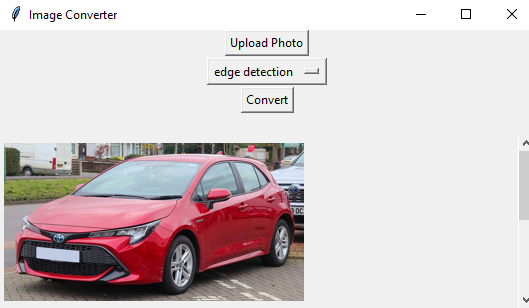


Figure running client gui on local machine

After the image is processed it is sent back from the master node and appears on our app

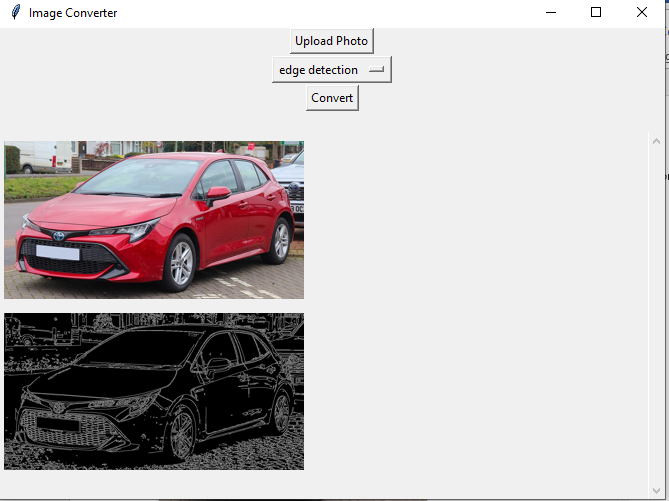


Figure image processed then sent to the client

Here is master node connection from client

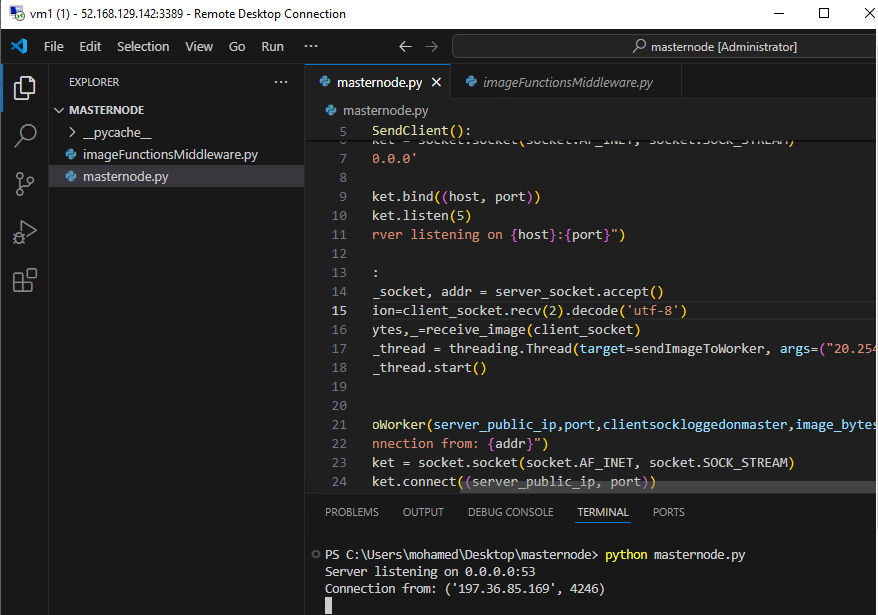


Figure master node connection from client

And this is worker node connection form master node

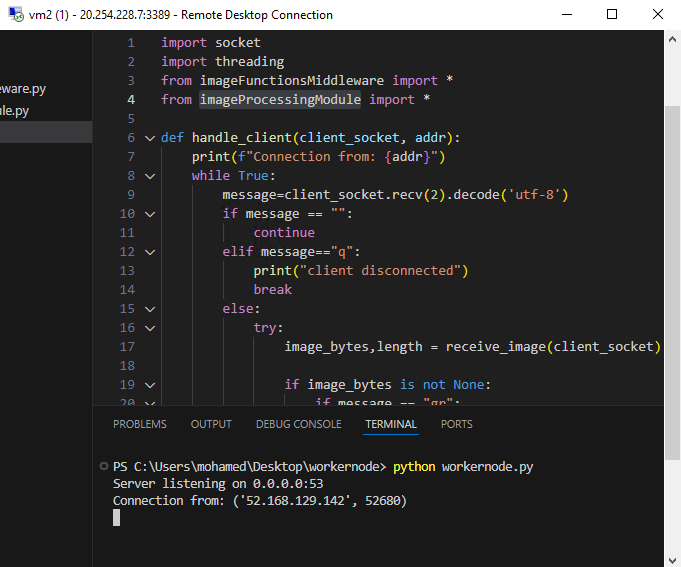
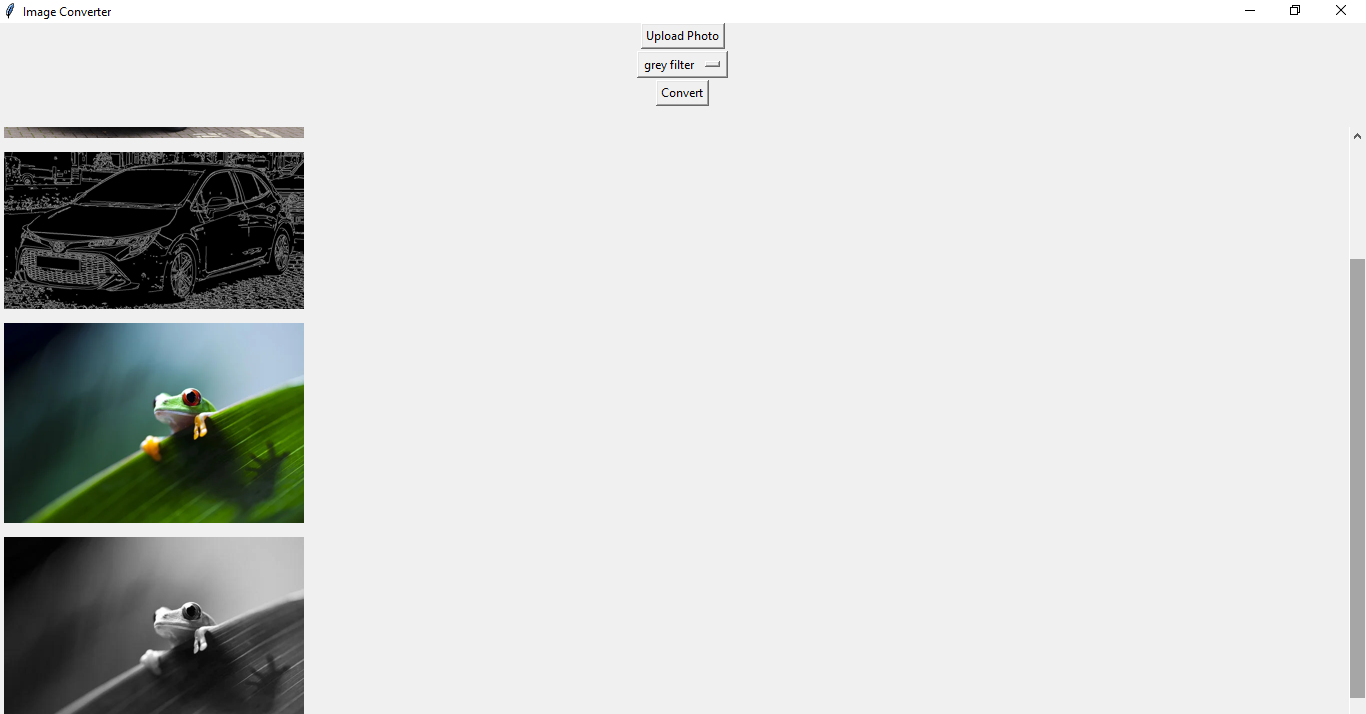


Figure worker node connection from master node

Trying many photos from the same client

Figure converting many photos from the same client



And this is trying many clients on the same time

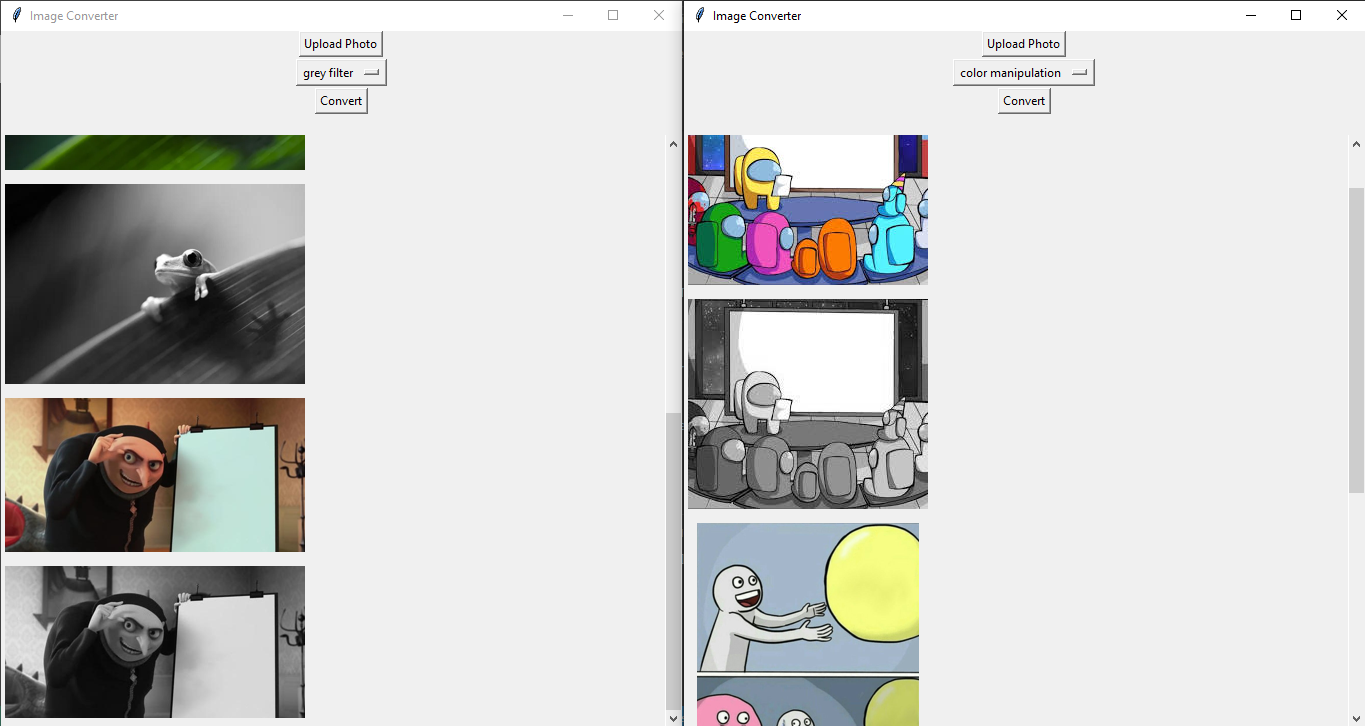


Figure converting many photos from different clients