

Distributed Image Processing System using Cloud Computing

Table of contents:

Introduction:.....	4
Project Scope:.....	4
Objectives:.....	4
Requirements:	5
Beneficiaries of the project:.....	5
User Stories:	6
System Architecture:.....	7
1. User Interface (UI):.....	7
2. Master Node:.....	7
3. Worker Nodes:	7
4. Communication Layer:	7
5. Monitoring:.....	7
Selected Technologies:.....	8
1. Cloud Platform:	8
2. Programming Language:	8
3. Parallel Computing Framework:	8
4. Communication:	8
Considerations:.....	8
Why we used these technologies:.....	8
1. Cloud Platform - Microsoft Azure:.....	8
2. Programming Language - Python:.....	9
3. Parallel Computing Framework - MPI (Message Passing Interface):.....	9
Communication:	10
Cost analysis:	11
Development Costs:	11
Infrastructure Costs:.....	11
Maintenance Costs:.....	12

Additional Costs:	12
Cost Estimation:	12
Cost Optimization Strategies:	12
Project plan:	13
Diagrams:	16
1. Sequence diagram:	16
2. Components diagram:	16
3. Infrastructure diagram:	17
4. Network diagram:	17
End-user guide: Distributed Image Processing System	18
1. Introduction	18
2. Getting Started	18
3. Uploading an Image	18
4. Selecting Image Processing Operation	18
5. Processing the Image	18
6. Conclusion	19
Additional Tips:	19
Phase 2 introduction:	19
Setting up the environment:	20
Codes:	22
1. Master node:	22
2. Worker node:	23
3. Client GUI:	25
4. Images functions middleware:	28
5. Image processing module:	30
Analysis of the codes:	32
1. Master node:	32
2. Worker node:	33
3. Client GUI:	34
4. Images functions middleware:	35
5. Image processing module:	36
Monitoring:	37
Client gui:	37

Master node:	38
Worker node:	39
Testing:	40
Conclusion:	46
References:	47

Table of figures:

Figure 1 client-master node protocol	10
Figure 2 master node-worker node protocol	10
Figure 3 sequence diagram	16
Figure 4 components diagram	16
Figure 5 infrastructure diagram	17
Figure 6 network diagram	17
Figure 7 azure resources	20
Figure 8 RDP download	20
Figure 9 inbound rules	21
Figure 10 pinging machines	21
Figure 11 testing the TCP port	22
Figure 12 running worker node on vm	40
Figure 13 running master node on another vm	41
Figure 14 running client GUI on local machine	41
Figure 15 image processed then sent to the client	42
Figure 16 master node connection from client	42
Figure 17 worker node connection from master node	43
Figure 18 converting many photos from the same client	43
Figure 19 converting many photos from different clients	44
Figure 20 worker node status active	44
Figure 21 worker node closed	45
Figure 22 master node closed	45
Figure 23 running nodes again and check server status	46

Introduction:

In today's digital era, the demand for image processing applications continues to rise, driven by various fields such as healthcare, entertainment, surveillance, and more. However, the computational complexity of image processing tasks often poses challenges in terms of processing time and resource utilization. To address these challenges, the integration of cloud computing and distributed systems has emerged as a powerful solution, enabling efficient parallel processing of image data.

The "Distributed Image Processing System using Cloud Computing" project aims to leverage the scalability and computational resources offered by cloud environments to implement a robust and efficient image processing system. By distributing processing tasks across multiple virtual machines in the cloud, the system can handle large volumes of image data effectively while ensuring scalability and fault tolerance.

This project focuses on developing a distributed system using Python programming language and cloud-based virtual machines. The system will utilize either OpenCL or MPI (Message Passing Interface) for parallel processing of image data, enabling the implementation of various image processing algorithms such as filtering, edge detection, and color manipulation.

Project Scope:

The project aims to develop a distributed image processing system utilizing cloud computing technologies. It involves designing and implementing a system capable of distributing image processing tasks across multiple virtual machines in the cloud. The system will support various image processing algorithms such as filtering, edge detection, and color manipulation. It should be scalable to accommodate an increasing workload by adding more virtual machines and should maintain fault tolerance to handle node failures gracefully.

Objectives:

- Design and implement a distributed image processing system using Python.
- Utilize cloud-based virtual machines for distributed computing.

- Use image processing algorithms including filtering, edge detection, and color manipulation.
- Ensure scalability to accommodate increased workload by adding virtual machines dynamically.
- Ensure fault tolerance by reassigning tasks from failed nodes to operational ones.

Requirements:

- **Distributed Processing:** The system should distribute image processing tasks across multiple virtual machines in the cloud.
- **Image Processing Algorithms:** Use filtering, edge detection, and colour manipulation algorithms.
- **Scalability:** The system should dynamically scale by adding virtual machines as the workload increases.
- **Fault Tolerance:** The system should handle node failures gracefully by reassigning tasks from failed nodes to operational ones.
- **User Interface:** Develop a user-friendly interface for users to upload images, select processing operations, monitor task progress, and download processed images.
- **Cloud Computing Platform:** Select a suitable cloud computing platform (e.g., AWS, Azure, Google Cloud) for hosting virtual machines.
- **Parallel Processing Framework:** Choose either OpenCL or MPI for parallel processing of image data.
- **Monitoring System:** Implement a monitoring system to track the progress of image processing tasks.
- **Documentation:** Provide comprehensive documentation including system architecture, setup instructions, and user guide.

Beneficiaries of the project:

1. **Researchers and Academia:** Researchers and academics involved in fields such as computer vision, image processing, and distributed systems can benefit from the project's advancements. The system provides a platform for exploring and experimenting with different image processing algorithms in a distributed computing environment, enabling them to conduct research and develop new techniques more efficiently.

2. **Healthcare Professionals:** In the healthcare industry, medical imaging plays a crucial role in diagnosis, treatment planning, and monitoring of patients. Healthcare professionals can benefit from the project by utilizing the distributed image processing system to enhance the speed and accuracy of medical image analysis. This can lead to faster diagnoses, improved treatment outcomes, and ultimately better patient care.
3. **Entertainment and Media Industry:** The entertainment and media industry often deals with large volumes of image and video data for tasks such as video editing, special effects, and content creation. The distributed image processing system can streamline these workflows by providing efficient parallel processing capabilities, enabling content creators to produce high-quality media content more effectively.
4. **Surveillance and Security Agencies:** Surveillance systems rely heavily on image processing technologies for tasks such as object detection, tracking, and facial recognition. By leveraging the distributed image processing system, surveillance and security agencies can enhance the capabilities of their surveillance systems, improving situational awareness and response times in critical situations.
5. **E-commerce and Retail:** E-commerce platforms and retail businesses can benefit from the project by integrating image processing capabilities for tasks such as product recognition, image-based search, and visual recommendation systems. The distributed system enables real-time processing of images, enhancing the user experience and driving sales through personalized product recommendations.
6. **Government and Public Sector:** Government agencies and public sector organizations can leverage the distributed image processing system for various applications, including satellite image analysis, urban planning, disaster management, and environmental monitoring. By processing large-scale image data efficiently, these organizations can make data-driven decisions and address societal challenges more effectively.

User Stories:

- As a user, I want to upload an image to the system for processing.
- As a user, I want to select the type of image processing operation to be performed.
- As a user, I want to download the processed image once the operation is complete.
- As a user, I want to monitor the progress of the image processing task.

System Architecture:

1. User Interface (UI):

- Provides an interface for users to interact with the system.
- Allows users to upload images, select processing operations, monitor task progress and view the processed images.

2. Master Node:

- Virtual machine in the cloud responsible for the application backend before the image processing.
- Handles user requests and generates messages to the worker nodes.
- Divide the image into many segments using image segmentation methods.
- Distributes image processing tasks to worker nodes.
- Manages scalability and fault tolerance.
- Sends back the processed image to the client UI.

3. Worker Nodes:

- Virtual machines in the cloud responsible for actual image processing using rpc architecture.
- Receive tasks from the backend (master node), perform processing using parallel computing, and return results.

4. Communication Layer:

- Facilitates communication between different components of the system using TCP and web sockets to RPC communication and we will define it below.
- Utilizes messaging protocols or frameworks for task distribution and result retrieval.

5. Monitoring:

- Monitors system performance, resource utilization, and task progress.

Selected Technologies:

1. Cloud Platform:

- **Microsoft Azure:** Cloud provider with virtual machines (Azure VMs), storage (Azure Blob Storage), and messaging services (Azure Service Bus).

2. Programming Language:

- **Python:** Selected for its ease of development, rich ecosystem of libraries (e.g., CV for image processing), and suitability for parallel computing.

3. Parallel Computing Framework:

- **MPI (Message Passing Interface):** Enables distributed computing and communication between worker nodes.

4. Communication:

- RPC with TCP and WebSockets

Considerations:

- **Scalability:** Ensure the architecture can scale horizontally by adding more worker nodes dynamically.
- **Fault Tolerance:** Implement mechanisms to handle node failures, such as task reassignment and redundancy.
- **Cost Optimization:** Optimize resource usage to minimize operational costs, especially in cloud environments where costs can scale with usage.

Why we used these technologies:

1. Cloud Platform - Microsoft Azure:

- **Azure VMs:** These provide scalable and customizable virtual machines, allowing the deployment of various applications without worrying about hardware infrastructure.
- **Azure Blob Storage:** Offers scalable object storage for documents, images, videos, and other unstructured data, enabling efficient data management and access.

Reason for Selection: Microsoft Azure was chosen for its robust infrastructure, extensive services, and reliable performance. It offers a wide range of scalable

solutions that fit the project's requirements, ensuring flexibility and efficiency in deployment and management.

2. Programming Language - Python:

- **Ease of Development:** Python's simple and readable syntax makes it easy to write and maintain code, accelerating development cycles.
- **Rich Ecosystem of Libraries:** Python boasts a vast collection of libraries for various purposes, such as computer vision (CV) for image processing, machine learning, data analysis, and more. This wealth of resources enhances productivity and facilitates the implementation of complex functionalities.
- **Suitability for Parallel Computing:** Python supports parallel computing through frameworks like MPI, enabling efficient utilization of resources and faster processing of tasks.

Reason for Selection: Python was chosen for its combination of simplicity, versatility, and powerful libraries. Its suitability for parallel computing aligns well with the project's requirements for efficient data processing and analysis.

3. Parallel Computing Framework - MPI (Message Passing Interface):

- **Distributed Computing:** MPI enables efficient communication and coordination between multiple processes running on distributed computing systems, allowing for parallel execution of tasks.
- **Scalability:** It supports scaling across multiple nodes, enabling the system to handle large datasets and compute-intensive workloads effectively.
- **Performance:** MPI is known for its high performance and low overhead, making it well-suited for demanding parallel computing tasks.

Reason for Selection: MPI was chosen for its proven track record in parallel computing, especially in high-performance computing (HPC) environments. It provides the necessary tools and mechanisms for efficient parallelization of algorithms and processing of large datasets, essential for the project's objectives.

Communication:

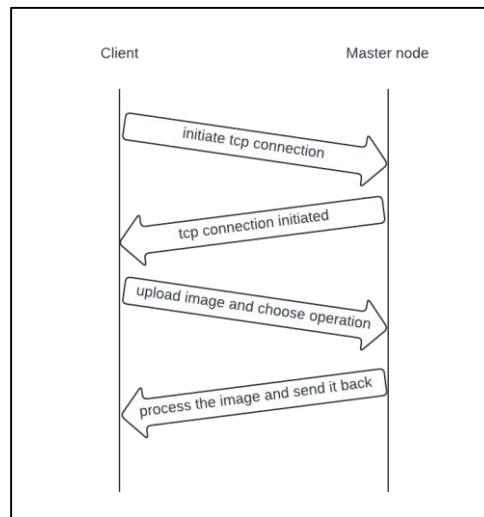


Figure 1 client-master node protocol

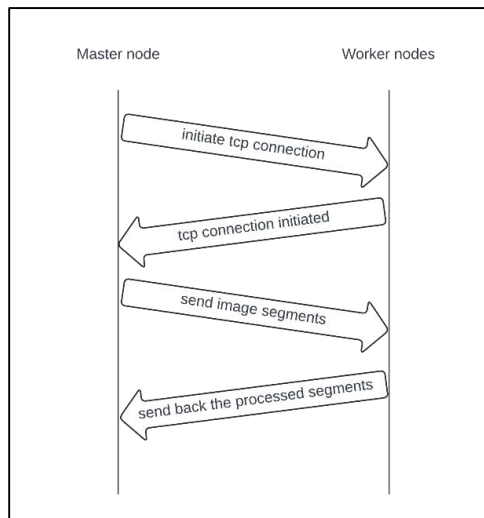


Figure 2 master node-worker node protocol

We will use TCP and WebSockets for RPC Communication

Cost analysis:

Developing a distributed image processing system using cloud computing Application involves various costs, including:

Development Costs:

- **Software Development:**
 - **Resource Time:**
 - Programming (Python) - Analyzing, designing, coding, and testing the application.
 - Network Engineering - Designing and implementing the network architecture.
 - **Tools and Frameworks:**
 - Python development environment (IDE)
 - Specific libraries for networking, parallel computing, and UI/UX
 - Cloud-based development platform
 - **Testing and Quality Assurance:**
 - Unit testing, integration testing, and user acceptance testing
 - Automated testing tools
- **Documentation:**
 - Creating user manuals, API documentation, and internal technical documentation

Infrastructure Costs:

- **Deployment:**
 - Cloud-based server
 - Domain name and SSL certificate
 - Load balancer
- **Hosting:**

- Monthly or annual fees for cloud server or other hosting services
- Bandwidth costs depending on user activity

Maintenance Costs:

- Bug Fixes: Addressing issues reported by users
- Feature Enhancements: Implementing new features and functionality
- Security Updates: Maintaining security patches and updates for libraries and frameworks
- Version Control: Managing code changes and releases

Additional Costs:

- Project Management: Planning, scheduling, and coordinating development activities
- Legal and Regulatory Compliance: Ensuring compliance with data privacy regulations
- Third-Party Services: APIs, libraries, or other paid services
- Marketing and Promotion: Advertising and promoting the application to attract users

Cost Estimation:

Due to the project's scope and varying factors, providing a definitive cost estimate is difficult. However, here's a rough breakdown:

- Development: \$5,000 - \$20,000+
- Infrastructure: \$500 - \$2,000+ per month
- Maintenance: \$1,000 - \$5,000+ per month

Cost Optimization Strategies:

- **Open-source libraries and frameworks:**
 - Utilize freely available libraries and frameworks for various functionalities, reducing licensing costs.
- **Cloud-based development and hosting:**
 - Leverage cloud platforms for development and deployment to reduce infrastructure costs and maintenance overhead.

- **Agile development methodology:**
 - Focus on rapid prototyping and iterative development to ensure resource efficiency and early feedback.
- **Community-driven development:**
 - Encourage contributions from open-source communities to leverage shared resources and expertise.

Overall, the cost of developing and maintaining a distributed image processing system using cloud computing Application will depend on various factors like project complexity, team size, and chosen technologies. Implementing cost-optimization strategies can significantly reduce expenses and ensure project viability.

Project plan:

Phase 1: Project Planning and Design (2-3 weeks)

Tasks:

1. Define project scope, objectives, and requirements.
2. Research cloud computing technologies and select the appropriate platform (AWS, Google Cloud, Azure, etc.).
3. Design system architecture, including components, interactions, and data flows.
4. Determine technologies for parallel processing (MPI, OpenCL).
5. Create a detailed project plan with tasks, responsibilities, and timelines.
6. Draft user stories based on gathered requirements.
7. Document project plan and design decisions.

Responsibilities:

- Mohamed Amr, Youssef Emad, Salma Nasreldin: System design, technology selection.
- Mohamed Ibrahim: Diagrams, user stories.

Timelines:

- Weeks 1-2: Define scope, objectives, and requirements; research and select technologies; design system architecture.
- Week 3: Finalize project plan, document design decisions, and user stories.

Phase 2: Development of Basic Functionality (2-3 weeks)

Tasks:

1. Implement basic image processing operations (filtering, edge detection, color manipulation).
2. Set up cloud environment and provision virtual machines.
3. Develop worker threads for processing tasks.
4. Implement image upload functionality.
5. Develop user interface for basic operations.
6. Manual testing of implemented functionality.

Responsibilities:

- All team: Implementation of basic functionalities, GUI coding.
- Mohamed Amr, Youssef Emad: Cloud setup, guidance on system integration, manual testing.
- Salma Nasreldin: Progress tracking, issue resolution, testing connection between vms and local machines.
- Mohamed Ibrahim: diagrams updating, GUI designing.

Timelines:

- Weeks 4: Implement basic image processing operations and cloud setup.
- Weeks 5-6: Develop worker threads, image upload functionality, and user interface.

Phase 3: Development of Advanced Functionality (2-3 weeks)

Tasks:

1. Implement advanced image processing operations (e.g., feature extraction, object recognition).
2. Develop distributed processing functionality using MPI or OpenCL.
3. Implement scalability features to add more virtual machines dynamically.

4. Incorporate fault tolerance mechanisms to handle node failures.
5. Conduct integration testing of advanced functionality.

Responsibilities:

- To be discussed

Timelines:

- Weeks 7-8: Implement advanced image processing operations and distributed processing.
- Weeks 8-9: Incorporate scalability and fault tolerance features, conduct integration testing.

Phase 4: Testing, Documentation, and Deployment (2-3 weeks)

Tasks:

1. Conduct thorough testing of the entire system, including unit, integration, and system testing.
2. Document system design, codebase, and user instructions.
3. Prepare deployment scripts and configurations.
4. Deploy the system to the cloud environment.
5. Perform final system testing and validation.

Responsibilities:

- To be discussed

Timelines:

- Weeks 10-11: Testing and documentation.
- Weeks 12: Deployment, final testing, and validation.

Diagrams:

1. Sequence diagram:

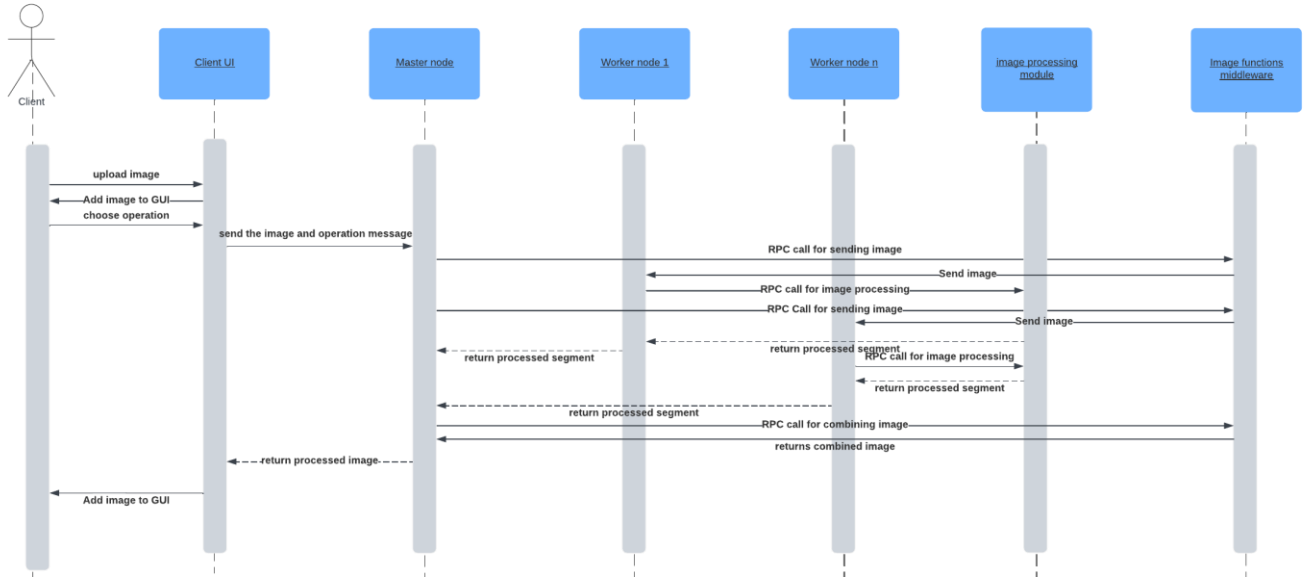


Figure 3 sequence diagram

2. Components diagram:

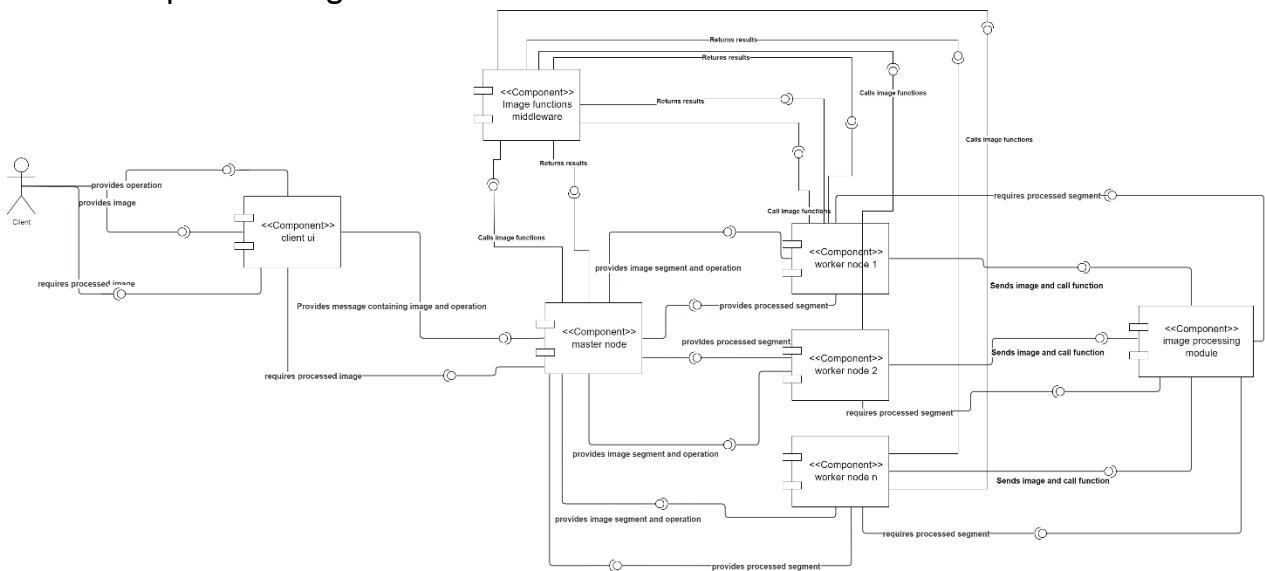


Figure 4 components diagram

3. Infrastructure diagram:

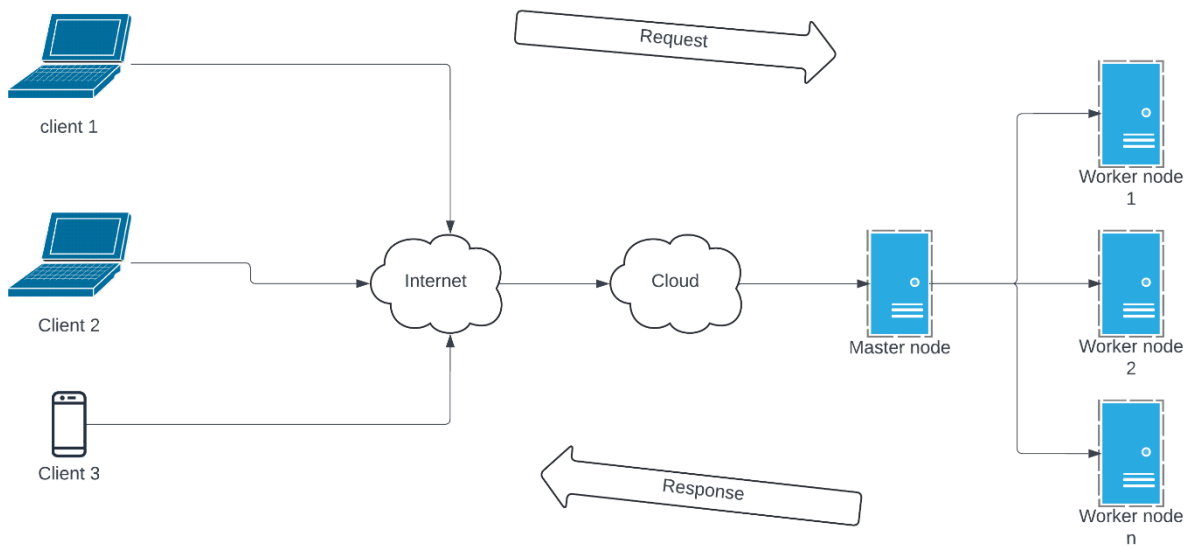


Figure 5 infrastructure diagram

4. Network diagram:

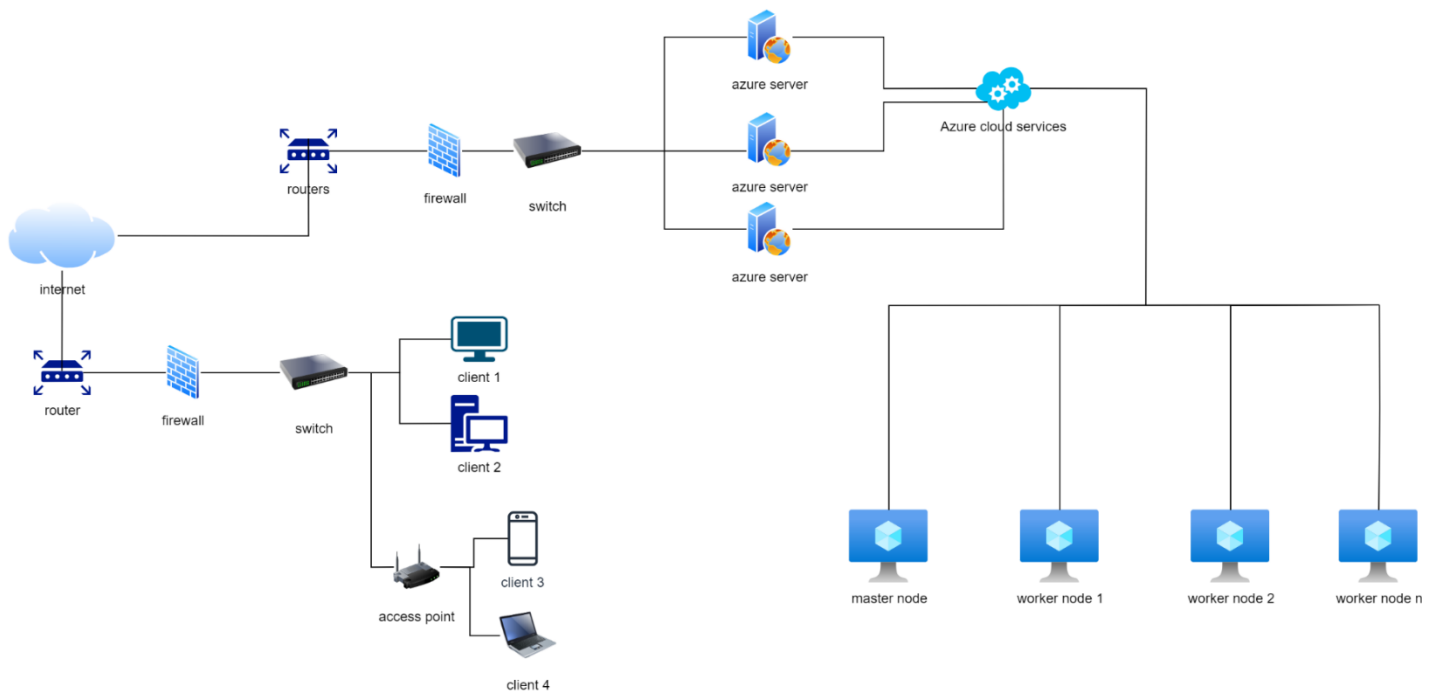


Figure 6 network diagram

End-user guide: Distributed Image Processing System

1. Introduction

Welcome to the Distributed Image Processing System! This user guide will walk you through the steps to upload an image, choose an image processing operation, and process the image using the system's graphical user interface (GUI).

2. Getting Started

- **System Requirements:** Ensure that you have a stable internet connection.
- **Accessing the System:** Open the GUI application.

3. Uploading an Image

- **Step 1:** Click on the "Upload" button to select an image file from your device.
- **Step 2:** Choose the image file you want to process from your local storage and click "Open" to upload it to the system.
- **Step 3:** Once the upload is complete, the selected image will be displayed on the GUI.

4. Selecting Image Processing Operation

- **Step 1:** Choose the type of image processing operation you want to perform from the dropdown menu.
- **Step 2:** Available operations may include:
 - Basic operations such as filtering, color manipulation, etc.
 - Advanced operations like edge detection.
- **Step 3:** After selecting the desired operation, the system will display a preview of the processed image on the GUI.

5. Processing the Image

- **Step 1:** Once you have selected the image processing operation, click on the "Convert" button to initiate the processing.
- **Step 2:** The system will distribute the processing task across multiple virtual machines in the cloud for parallel execution.
- **Step 3:** Once the processing is complete, the processed image will be displayed on the GUI, and you can download it to your device.

6. Conclusion

Congratulations! You have successfully processed an image using the Distributed Image Processing System. Feel free to explore other image processing operations and functionalities offered by the system.

Additional Tips:

- If you encounter any issues or have questions about the system's functionality, refer to the system documentation or contact your system administrator for assistance.
- Ensure that you have the necessary permissions to access and use the system features effectively.

Thank you for using the Distributed Image Processing System. We hope you find it useful for your image processing needs!

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:

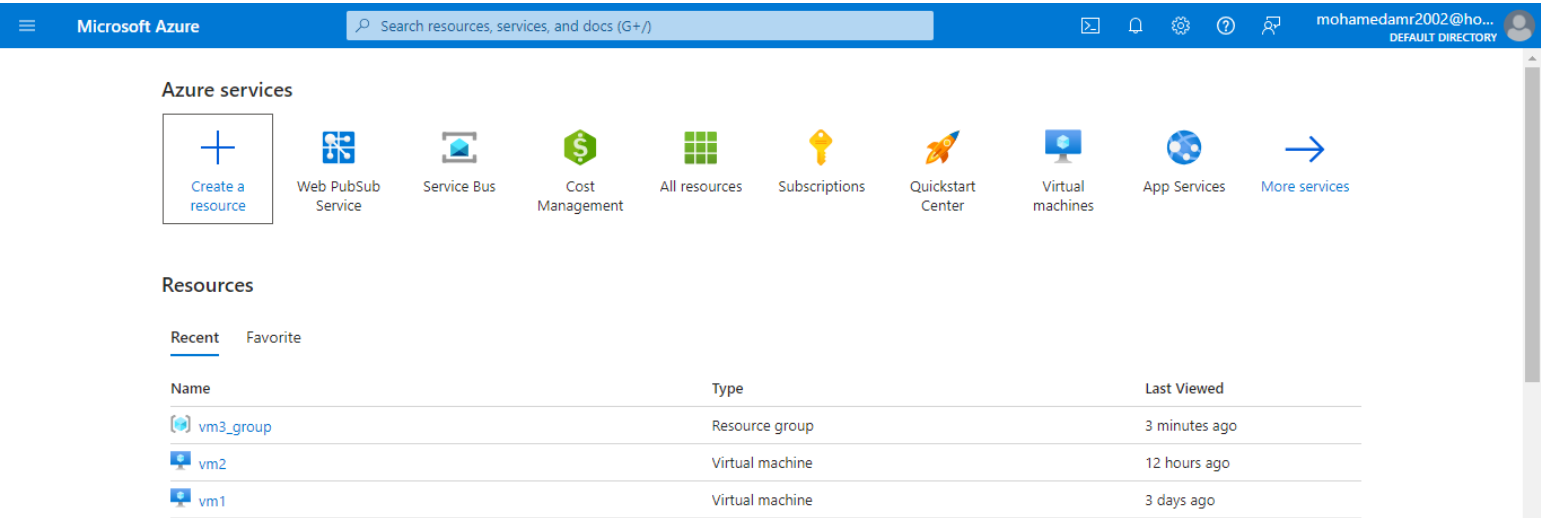


Figure 7 azure resources

Then we downloaded RDP files for both machine to start them

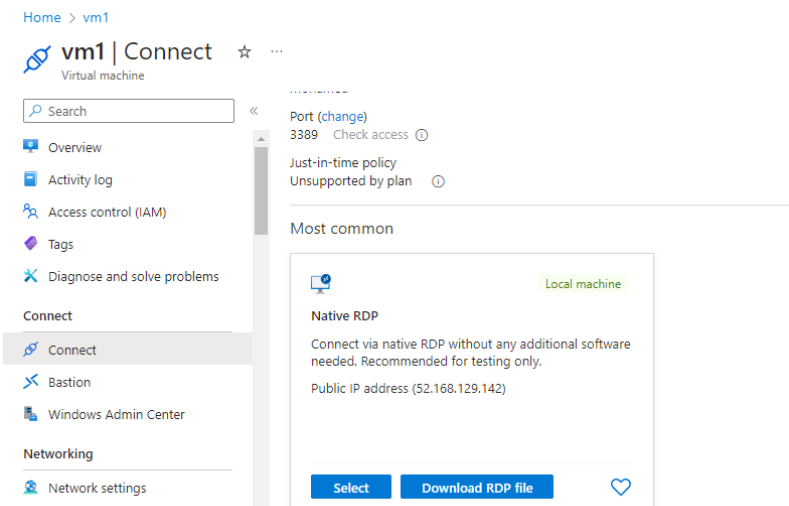


Figure 8 RDP download

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:

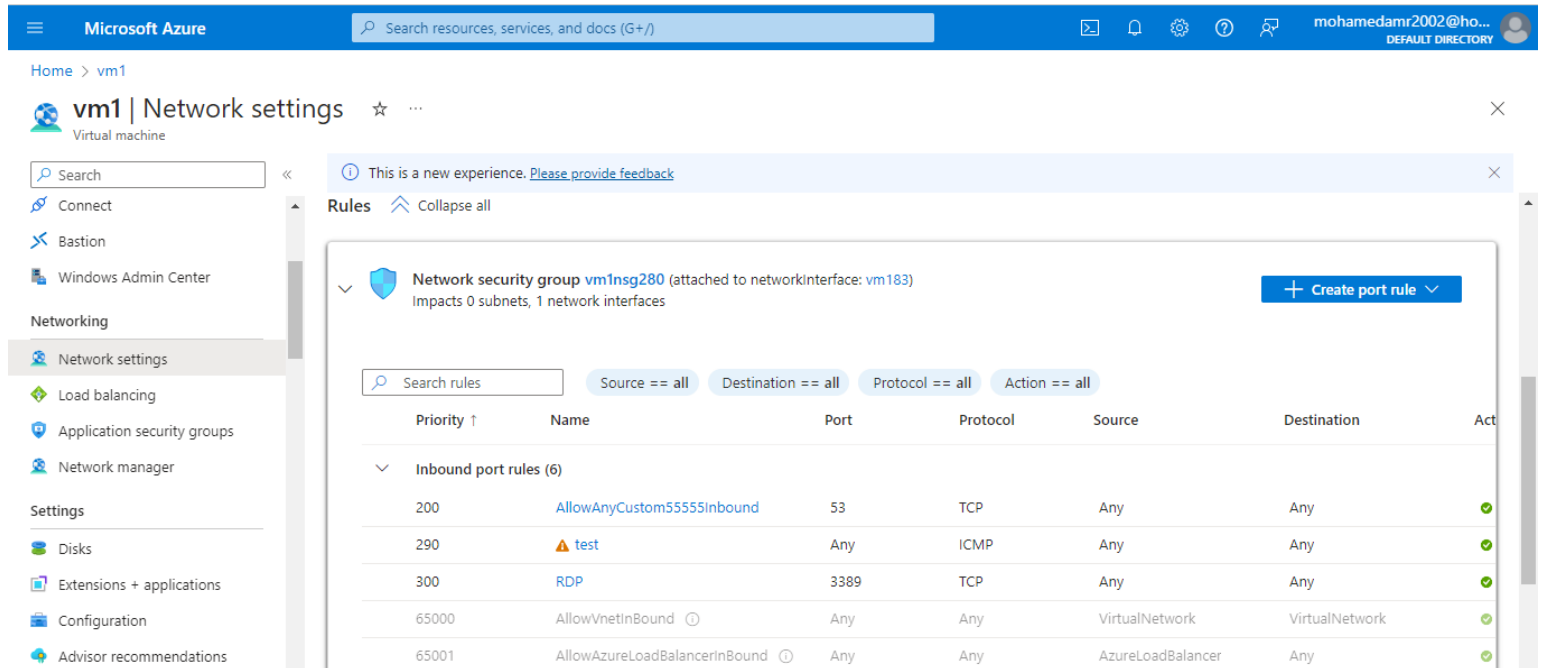


Figure 9 inbound rules

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

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Mohamed Amr> ping 52.168.129.142

Pinging 52.168.129.142 with 32 bytes of data:
Reply from 52.168.129.142: bytes=32 time=165ms TTL=110
Reply from 52.168.129.142: bytes=32 time=136ms TTL=110
Reply from 52.168.129.142: bytes=32 time=134ms TTL=110
Reply from 52.168.129.142: bytes=32 time=216ms TTL=110

Ping statistics for 52.168.129.142:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 134ms, Maximum = 216ms, Average = 162ms
PS C:\Users\Mohamed Amr>

```

Figure 10 pinging machines

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

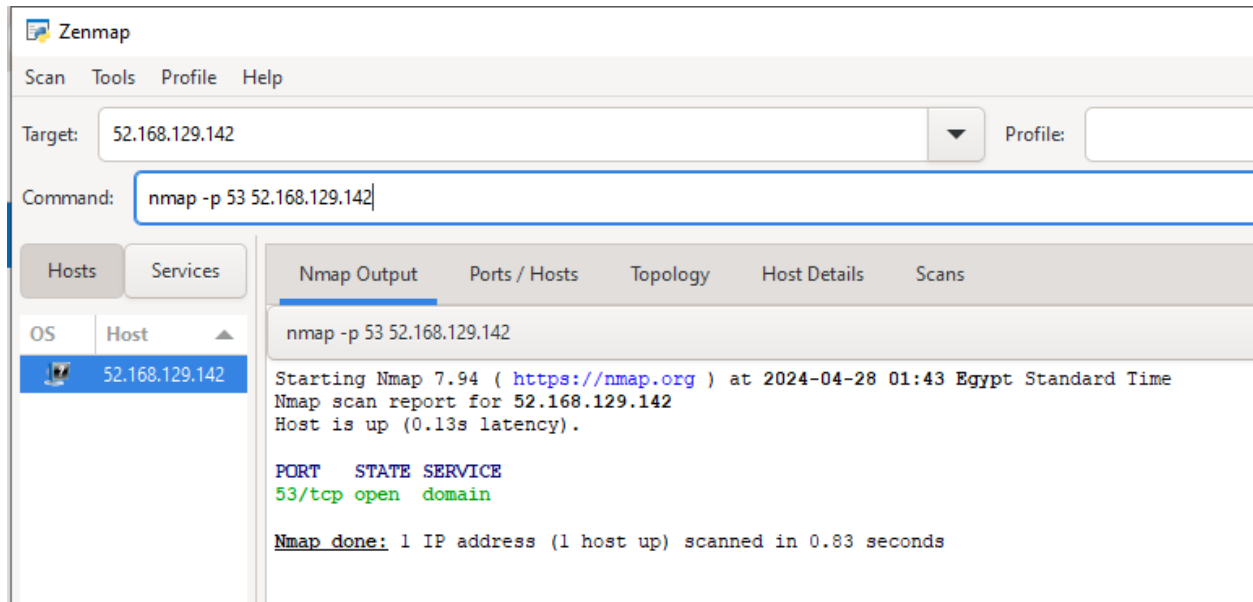


Figure 11 testing the TCP port

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

Codes:

1. 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,oper
ation,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()

```

2. 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,
(processessed_image_bytes, len(processessed_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()
```

3. 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="f1"
    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()

```

4. 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])

```

5. 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:

1. 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 "gr", "ed", 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.

2. 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.

3. 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.

4. 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.

5. 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.

Monitoring:

Client gui:

We added fixed label at the bottom to get the server status

```
self.server_status_label = tk.Label(root, text="Server Status: Unknown")
self.server_status_label.pack()
```

we added also functions and a thread for sending and receiving to and from master node

```
def receive_server_status(self):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_public_ip = 'localhost' # '52.168.129.142'
    port = 12348 # Port for receiving server status
    try:
        client_socket.connect((server_public_ip, port))
        client_socket.send("st".encode('utf-8'))
        message = client_socket.recv(2).decode('utf-8')
        if message=="ok":
            return "active"
        elif message=="no":
            return "error"
        return message
    except ConnectionRefusedError:
        print("Connection to server failed.")
        return "Error in Master Node"

def monitor_server_status_thread(self):
    threading.Thread(target=self.monitor_server_status).start()

def monitor_server_status(self):
    while True:
        status = self.receive_server_status()
        self.server_status_label.config(text=f"Server Status: {status}")
        if status == "active":
            self.server_status_label.config(fg="green")
```

```

        else:
            self.server_status_label.config(fg="red")
            # Update label with server status

            time.sleep(1) # Adjust the sleep time as needed

```

Then we called the thread at the main loop

```

if __name__ == "__main__":
    root = tk.Tk()
    app = ImageConverterApp(root)
    app.monitor_server_status_thread()
    root.mainloop()

```

Master node:

We added if condition in the main loop to differentiate between regular message and server status message

```

while True:
    client_socket, addr = server_socket.accept()
    operation=client_socket.recv(2).decode('utf-8')
    if operation == "st":
        monitorWorker('localhost',12345,client_socket)
    else:
        imageBytes,_=receive_image(client_socket)
        client_thread = threading.Thread(target=sendImageToWorker,
args=("localhost",12345,client_socket,imageBytes,operation,addr))
        client_thread.start()

```

Then we added monitorWorker function to test the worker node

```

def monitorWorker(server_public_ip, port, clientsockloggedonmaster):
# while True:
    try:
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client_socket.connect((server_public_ip, port))
        client_socket.send("st".encode('utf-8'))
        message = client_socket.recv(2).decode('utf-8')
        if message == "ok":
            clientsockloggedonmaster.send("ok".encode('utf-8'))
            print("Worker is still alive")
        else:
            print("Worker is dead")
    except ConnectionRefusedError:

```

```
clientsockloggedonmaster.send("no".encode('utf-8'))  
print("Connection to worker failed. Worker might be down.")
```

Worker node:

Then we added if condition also in the main loop of handle_client function in the worker node to send ok if server status is good

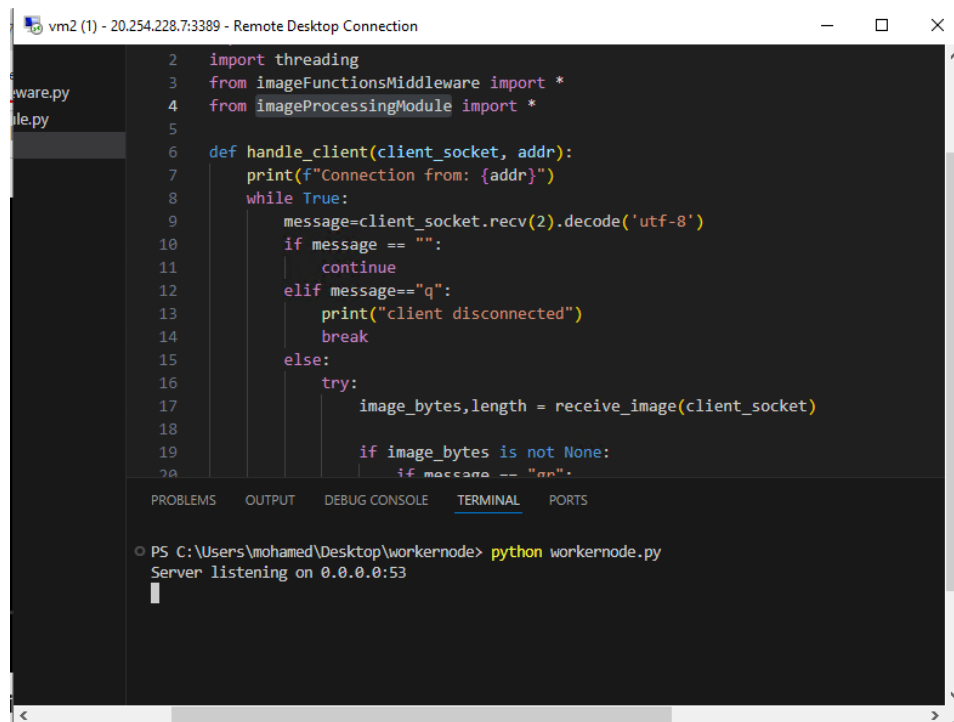
```
def handle_client(client_socket, addr):  
    print(f"Connection from: {addr}")  
    while True:  
        message=client_socket.recv(2).decode('utf-8')  
        if message == "":  
            continue  
  
        elif message=="st":  
            client_socket.send("ok".encode('utf-8'))
```

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



```
vm2 (1) - 20.254.228.7:3389 - Remote Desktop Connection

2  import threading
3  from imageFunctionsMiddleware import *
4  from imageProcessingModule import *
5
6  def handle_client(client_socket, addr):
7      print(f"Connection from: {addr}")
8      while True:
9          message=client_socket.recv(2).decode('utf-8')
10         if message == "":
11             continue
12         elif message=="q":
13             print("client disconnected")
14             break
15         else:
16             try:
17                 image_bytes,length = receive_image(client_socket)
18
19                 if image_bytes is not None:
20                     if message == "on":
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\mohamed\Desktop\workernode> python workernode.py
Server listening on 0.0.0.0:53
```

Figure 12 running worker node on vm

And we will run the master node on another vm

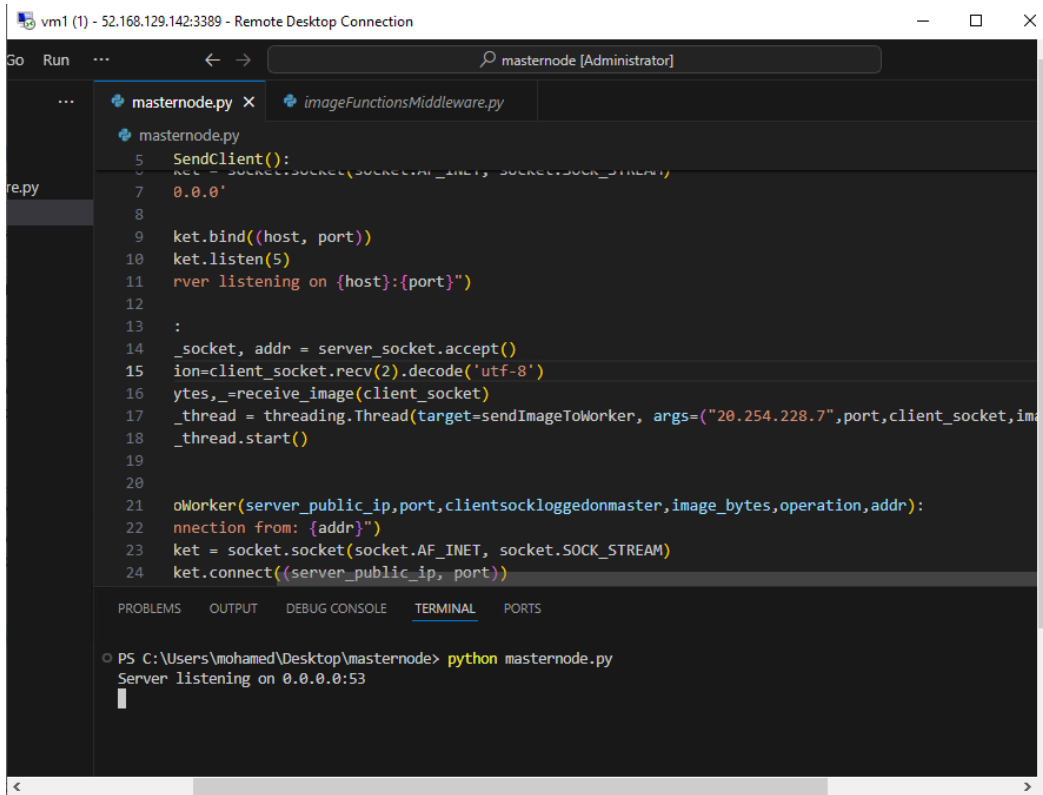


Figure 13 running master node on another vm

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

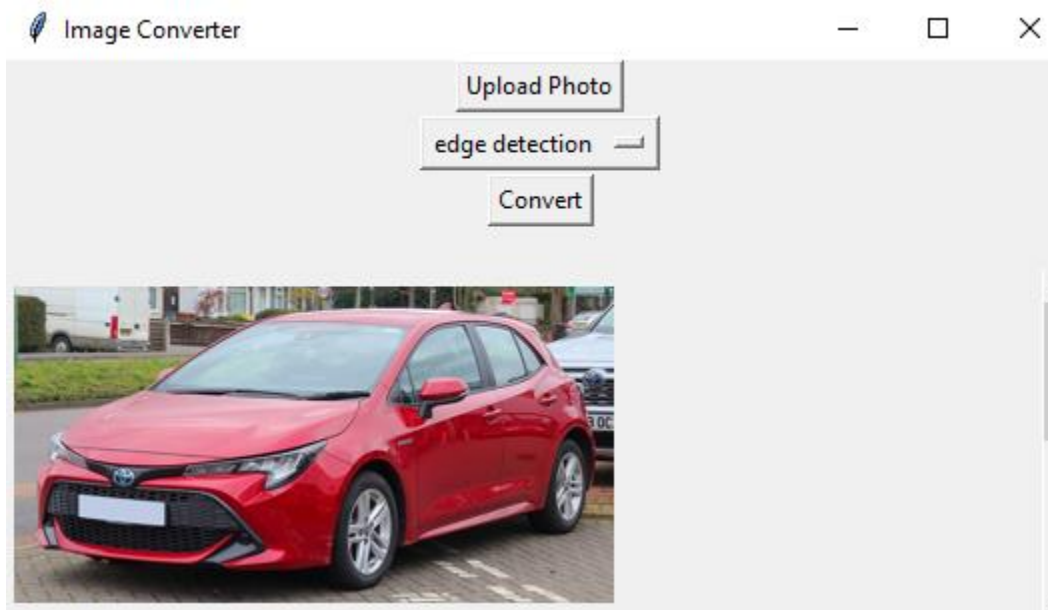


Figure 14 running client GUI on local machine

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

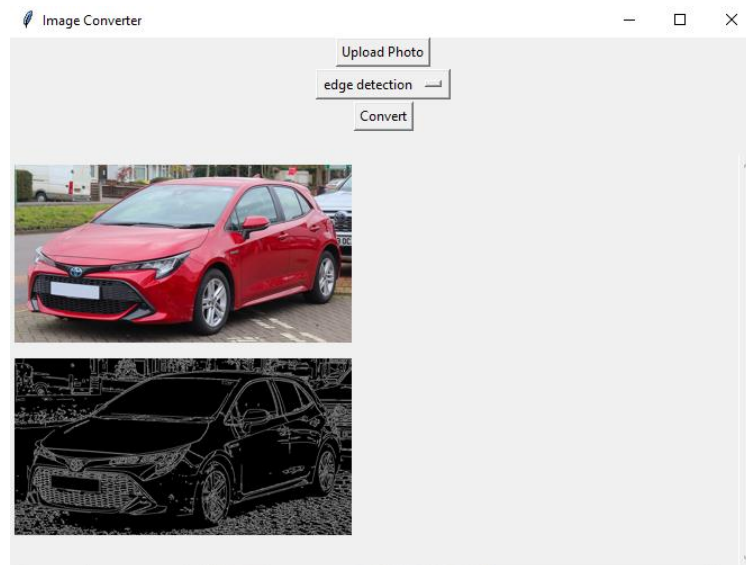


Figure 15 image processed then sent to the client

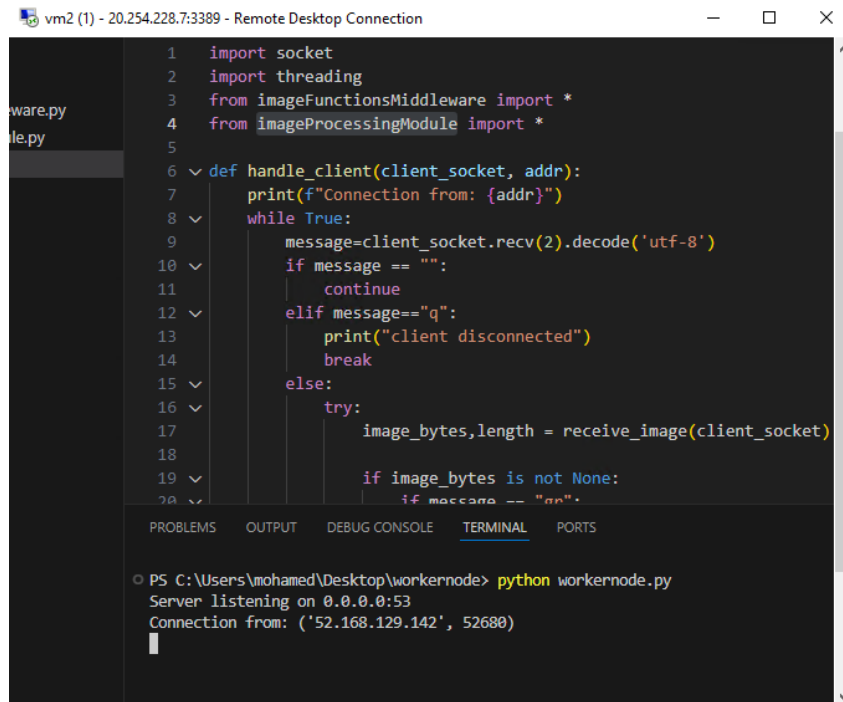
Here is master node connection from client

A screenshot of a Remote Desktop Connection window titled "vm1 (1) - 52.168.129.142:3389 - Remote Desktop Connection". The window shows a Windows File Explorer on the left with the "MASTERNODE" folder selected, containing files like "_pycache_", "imageFunctionsMiddleware.py", and "masternode.py". The main area shows the "masternode.py" script in a code editor. The script defines a "SendClient()" function that listens on 0.0.0.0:53, accepts a connection, receives an image, and sends it to a worker. The terminal at the bottom shows the command "python masternode.py" and the output "Server listening on 0.0.0.0:53" and "Connection from: ('197.36.85.169', 4246)".

```
5 def SendClient():
6     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7     sock.bind((0.0.0.0, 53))
8
9     sock.listen(5)
10    while True:
11        client_socket, addr = sock.accept()
12        data = client_socket.recv(2048).decode('utf-8')
13        bytes_data = data.encode('utf-8')
14        _thread = threading.Thread(target=sendImageToWorker, args=(20.254.1.1, bytes_data))
15        _thread.start()
16
17    def sendImageToWorker(server_public_ip, port, client_socket, image_bytes):
18        connection = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
19        connection.connect((server_public_ip, port))
20        connection.send(image_bytes)
```

Figure 16 master node connection from client

And this is worker node connection form master node



vm2 (1) - 20.254.228.7:3389 - Remote Desktop Connection

```
1 import socket
2 import threading
3 from imageFunctionsMiddleware import *
4 from imageProcessingModule import *
5
6 def handle_client(client_socket, addr):
7     print(f"Connection from: {addr}")
8     while True:
9         message=client_socket.recv(2).decode('utf-8')
10        if message == "":
11            continue
12        elif message=="q":
13            print("client disconnected")
14            break
15        else:
16            try:
17                image_bytes,length = receive_image(client_socket)
18
19                if image_bytes is not None:
20                    if message == "qq":
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

PS C:\Users\mohamed\Desktop\workernode> python workernode.py
Server listening on 0.0.0.0:53
Connection from: ('52.168.129.142', 52680)

Figure 17 worker node connection from master node

Trying many photos from the same client

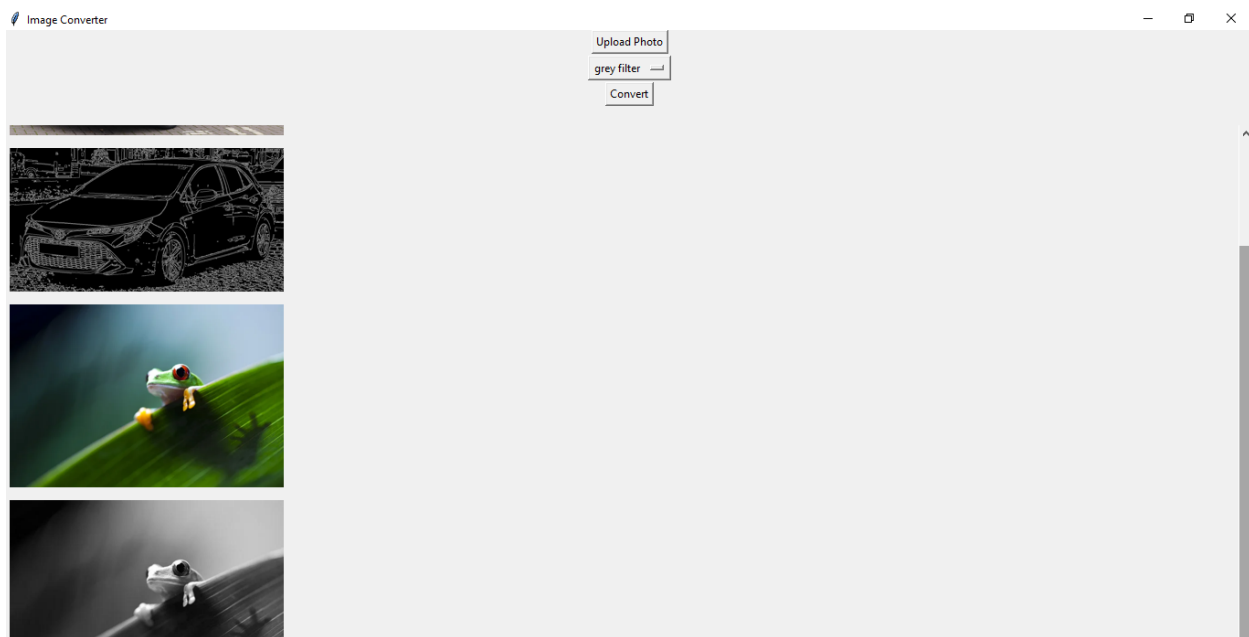


Figure 18 converting many photos from the same client

And this is trying many clients on the same time

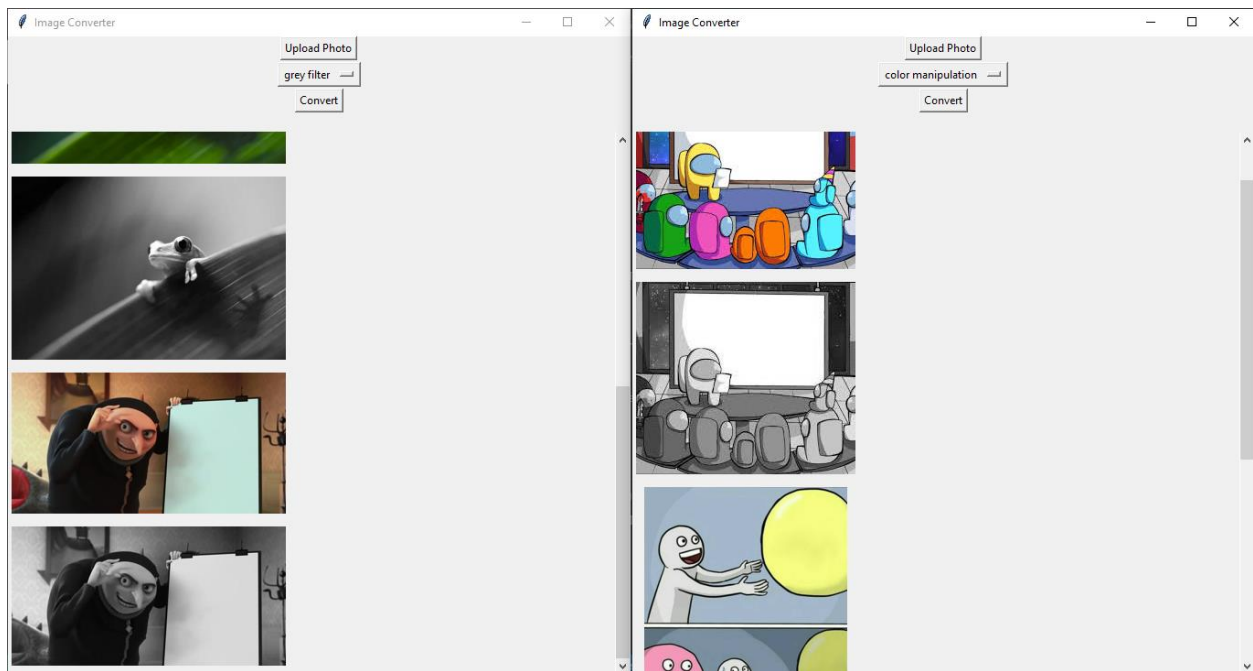


Figure 19 converting many photos from different clients

Then we added server monitoring feature

The worker is running correctly

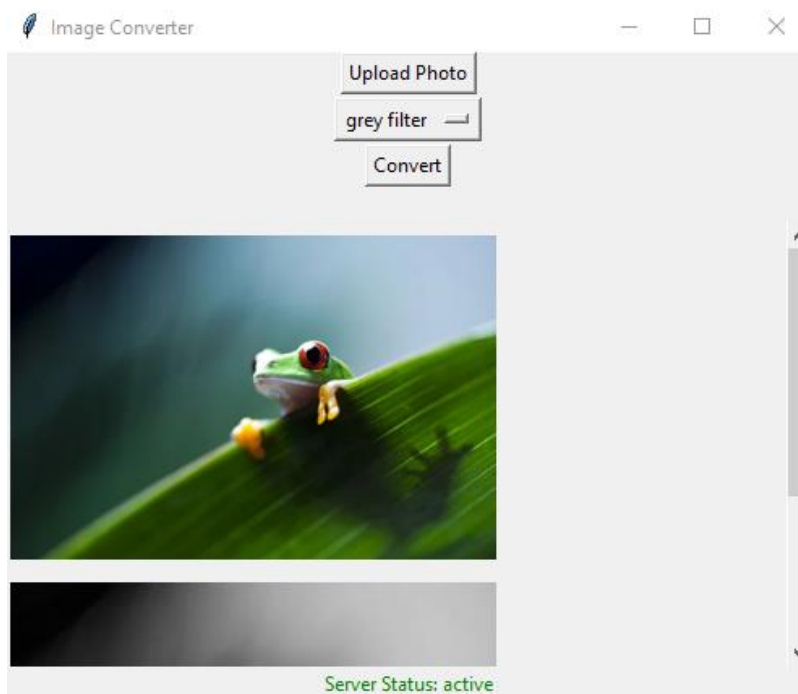


Figure 20 worker node status active

If the worker node is closed

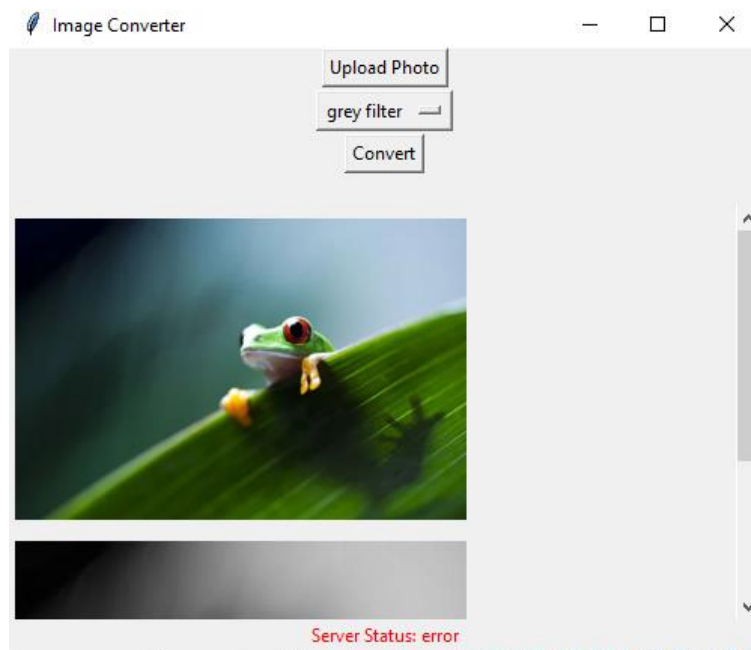


Figure 21 worker node closed

If the master node is closed

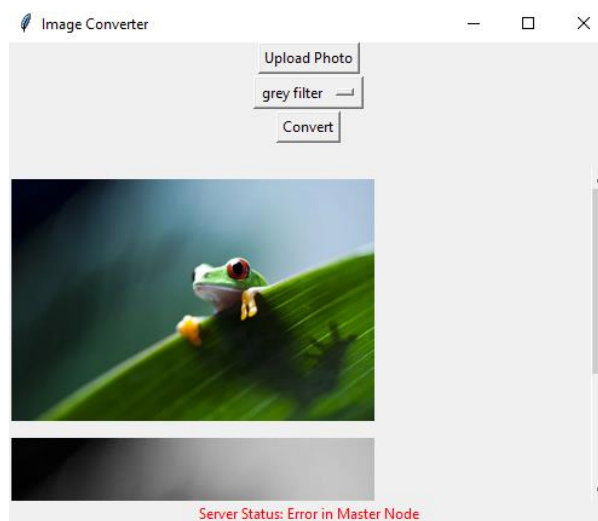


Figure 22 master node closed

After we ran them again

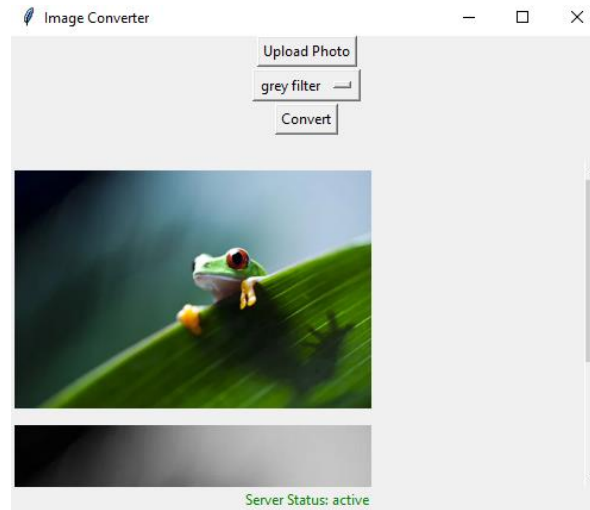


Figure 23 running nodes again and check server status

Conclusion:

In conclusion, the "Distributed Image Processing System using Cloud Computing" project represents a powerful solution to the growing demand for efficient image processing capabilities across various industries and domains. By harnessing the power of cloud computing and distributed systems, the system enables faster, more scalable, and fault-tolerant image processing workflows, benefiting researchers, healthcare professionals, media creators, security agencies, e-commerce platforms, government organizations, and more.

Moving forward, the project lays a solid foundation for future enhancements and extensions, paving the way for further innovation in the field of distributed image processing. With ongoing development and refinement, the system has the potential to continue making significant contributions to advancing image processing technologies and addressing real-world challenges effectively.

References:

<https://learn.microsoft.com/en-us/azure/?product=popular>

<https://learn.microsoft.com/en-us/azure/azure-portal/>

<https://docs.python.org/3/library/socket.html>

<https://realpython.com/python-sockets/>

<https://docs.python.org/3/howto/sockets.html>

<https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC>

<https://miro.com/diagramming/what-is-a-uml-diagram/>

<https://www.geeksforgeeks.org/python-network-programming/>

<https://konfuzio.com/en/cv2/#:~:text=The%20cv2%20module%20is%20the,commonly%20used%20functions%20in%20cv2.>

<https://nmap.org/docs.html>

<https://docs.python.org/3/library/tk.html>