

# Distributed Image Processing System using Cloud



**Program: Computer Engineering and  
Software Systems**

**Course Code: CSE354**

**Course Name: Distributed Computing**

**Submitted to**

**Prof. Ayman M. Bahaa-Eldin**

**Ain Shams University**

**Faculty of Engineering**

**Spring Semester – 2024**

**Team 19**

**Phase 4**

## **Personal Information**

**20P3485 Mohamed Amr EL Mallah**

**20P8449 Mohamed Ibrahim Elsayed Barakat**

**20P7105 Salma Nasrelden Aboelela Hendawy**

**20P3844 Youssef Emad Eldin Elshahat Barakat**

# 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:.....	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 – TCP sockets:.....	9
Communication: .....	11
Cost analysis: .....	12
Development Costs: .....	12
Infrastructure Costs:.....	12
Maintenance Costs:.....	13

<b>Additional Costs:</b>	13
<b>Cost Estimation:</b>	13
<b>Cost Optimization Strategies:</b>	13
<b>Project plan:</b>	14
<b>Diagrams:</b>	17
1. Sequence diagrams:	17
2. Components diagram:	18
3. Infrastructure diagram:	19
4. Network diagram:	19
<b>End-user guide: Distributed Image Processing System</b>	20
1. Introduction	20
2. Getting Started	20
3. Uploading an Image	20
4. Selecting Image Processing Operation	20
5. Processing the Image	20
6. Conclusion	21
<b>Additional Tips:</b>	21
<b>Setting up the environment:</b>	21
<b>Codes:</b>	24
1. Master node:	24
2. Worker node:	27
3. Client GUI:	30
4. Images functions middleware:	37
5. Database:	39
6. Get Logs:	40
<b>Analysis of the codes:</b>	40
1. Master node:	40
2. Worker node:	43
3. Client GUI:	45
4. Images functions middleware:	47
5. Image processing module:	49
6. DB:	50
7. View logs:	51

<b>Image processing:</b> .....	51
<b>Monitoring:</b> .....	51
<b>Client gui:</b> .....	51
<b>Master node:</b> .....	54
<b>Worker node:</b> .....	55
<b>Fault tolerance:</b> .....	55
<b>Scalability:</b> .....	56
<b>Parallelizing:</b> .....	57
<b>Database and logging:</b> .....	57
<b>Testing:</b> .....	58
<b>Conclusion:</b> .....	63
<b>GitHub link:</b> .....	64
<b>References:</b> .....	64

## Table of figures:

Figure 1 client-master node protocol .....	11
Figure 2 master node-worker node protocol .....	11
Figure 3 master node monitoring sequence diagram.....	17
Figure 4 sequence diagram .....	17
Figure 5 monitoring workernodes sequence diagram.....	18
Figure 6 components diagram .....	18
Figure 7 infrastructure diagram .....	19
Figure 8 network diagram.....	19
Figure 9 azure resources .....	21
Figure 10 RDP download.....	22
Figure 11 inbound rules .....	22
Figure 12 pinging machines .....	23
Figure 13 testing the TCP port .....	23
Figure 14 database logs .....	58
Figure 15 running worker node on vm .....	59
Figure 16 running master node on another vm.....	60
Figure 17 running client GUI on local machine.....	60
Figure 18 image processed then sent to the client.....	61
Figure 19 master node connection from client .....	61
Figure 20 worker node connection from master node.....	62
Figure 21 converting many photos from the same client.....	62
Figure 22 converting many photos from different clients.....	63

## 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 distribution 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 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:

- **TCP sockets:** we implemented our parallel programming functions with TCP sockets without needing for MPI.

### 4. Communication:

- 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, 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 – TCP sockets:

Parallel programming with Python TCP sockets offers several benefits, particularly when dealing with network-bound or I/O-bound tasks. Here are some of the key advantages:

### 1. Improved Performance and Throughput

- **Concurrency:** Parallel programming allows handling multiple connections simultaneously, which can significantly improve the performance of network applications. This is particularly beneficial in a server environment where numerous clients are making requests concurrently.
- **Efficient Resource Utilization:** By distributing tasks across multiple threads or processes, you can better utilize the available CPU cores and network bandwidth, leading to improved overall throughput.

### 2. Reduced Latency

- **Faster Response Times:** Handling multiple client requests in parallel can reduce the latency experienced by each client, as the server can process multiple requests concurrently rather than sequentially.
- **Responsive Servers:** Parallel processing helps in maintaining a responsive server even under heavy load, as incoming connections are handled promptly without significant delays.

### 3. Scalability

- **Handling Large Number of Connections:** Parallel programming allows the server to scale and handle a large number of simultaneous connections. This is crucial for applications like web servers, chat applications, and multiplayer games.
- **Dynamic Resource Allocation:** With parallel programming, resources can be dynamically allocated based on the workload, providing better scalability for varying loads.

#### 4. Enhanced Reliability and Fault Tolerance

- **Isolation of Tasks:** Using separate threads or processes can isolate tasks, so a failure in one task does not necessarily impact others. This enhances the reliability of the application.
- **Graceful Degradation:** In case of overload or partial failure, parallel programming can help in maintaining the operation of critical components, ensuring that the system degrades gracefully rather than failing completely.

#### 5. Simplified Code Structure for Asynchronous Tasks

- **Easier to Manage:** For many I/O-bound tasks, using parallel programming techniques like threading or asynchronous I/O can simplify the code structure. This can make it easier to write and maintain compared to complex state machine-based asynchronous code.
- **Modular Design:** Tasks can be encapsulated into independent modules or functions, leading to cleaner and more maintainable code.

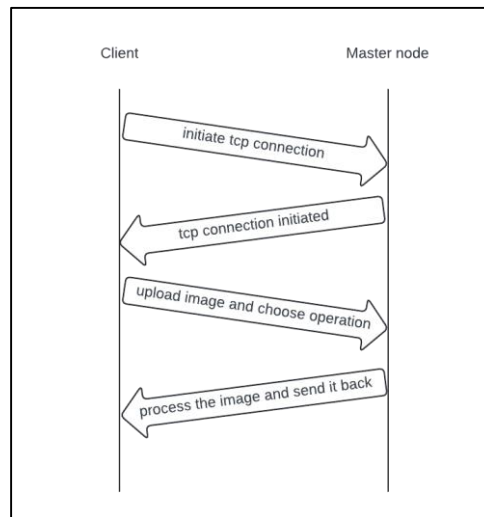
#### 6. Efficient Use of Multiprocessing

- **CPU-Bound Tasks:** For tasks that are CPU-bound, using multiprocessing can take full advantage of multiple CPU cores. Python's Global Interpreter Lock (GIL) can limit the performance of multi-threaded programs, but multiprocessing can bypass this limitation by running separate Python interpreter instances.
- **Concurrent Data Processing:** This is especially useful for applications that need to perform intensive data processing in addition to handling network communication.

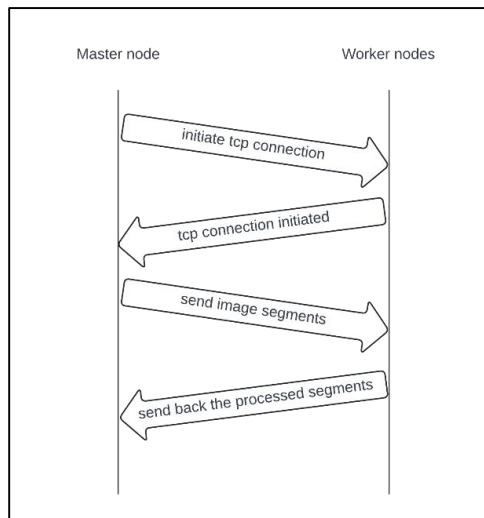
#### 7. Asynchronous I/O Operations

- **Event-Driven Architecture:** Libraries like **asyncio** enable an event-driven programming style, which is highly efficient for I/O-bound and high-level structured network code.
- **Non-Blocking Operations:** Asynchronous programming allows for non-blocking operations, meaning a single thread can manage many network connections without being blocked by any single operation.

## Communication:



*Figure 1 client-master node protocol*



*Figure 2 master node-worker node protocol*

We will use TCP and WebSockets for 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) or others.
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:**

- Salma Nasreldin, Youssef Emad: Implement advanced image processing operations and distributed processing.
- Mohamed Amr, Mohamed Ibrahim: Incorporate scalability and fault tolerance features, conduct integration testing.

**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:**

- Mohamed Amr, Salma Nasreldin, Youssef Emad: Testing and documentation.
- Mohamed Ibrahim: Deployment, final testing, and validation.

**Timelines:**

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

## Diagrams:

### 1. Sequence diagrams:

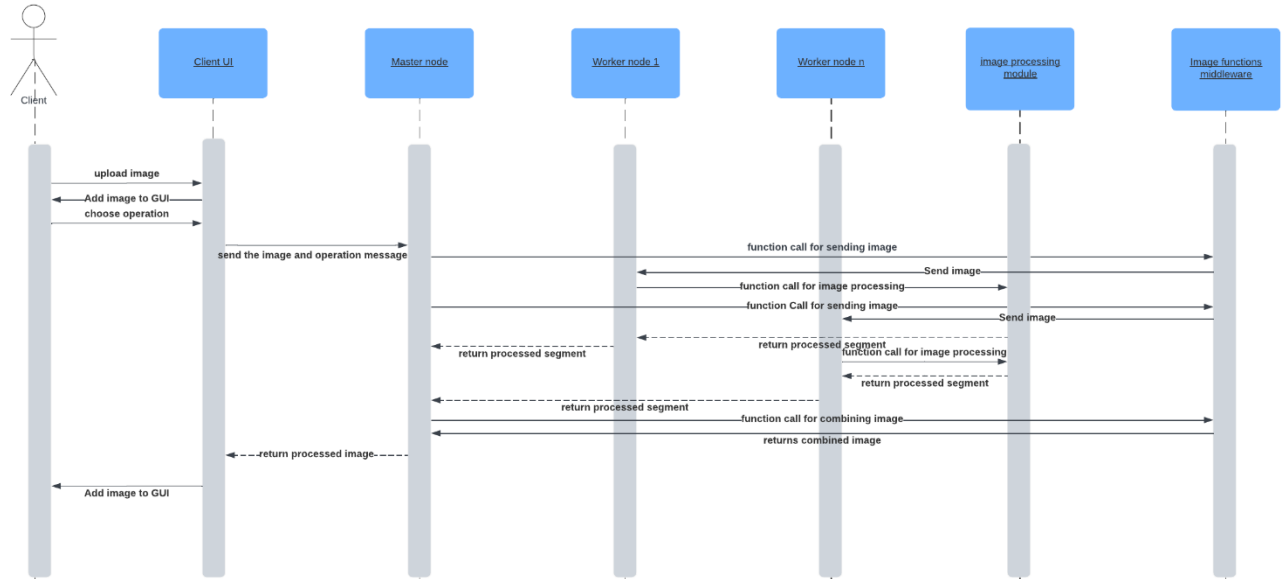


Figure 4 sequence diagram

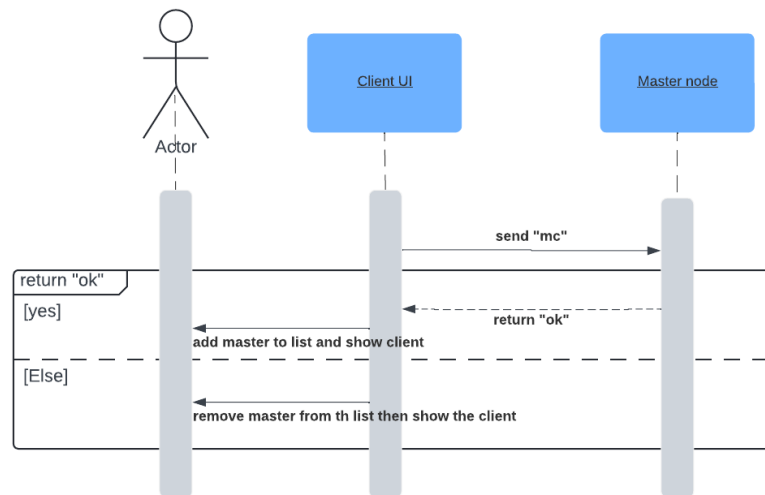


Figure 3 master node monitoring sequence diagram

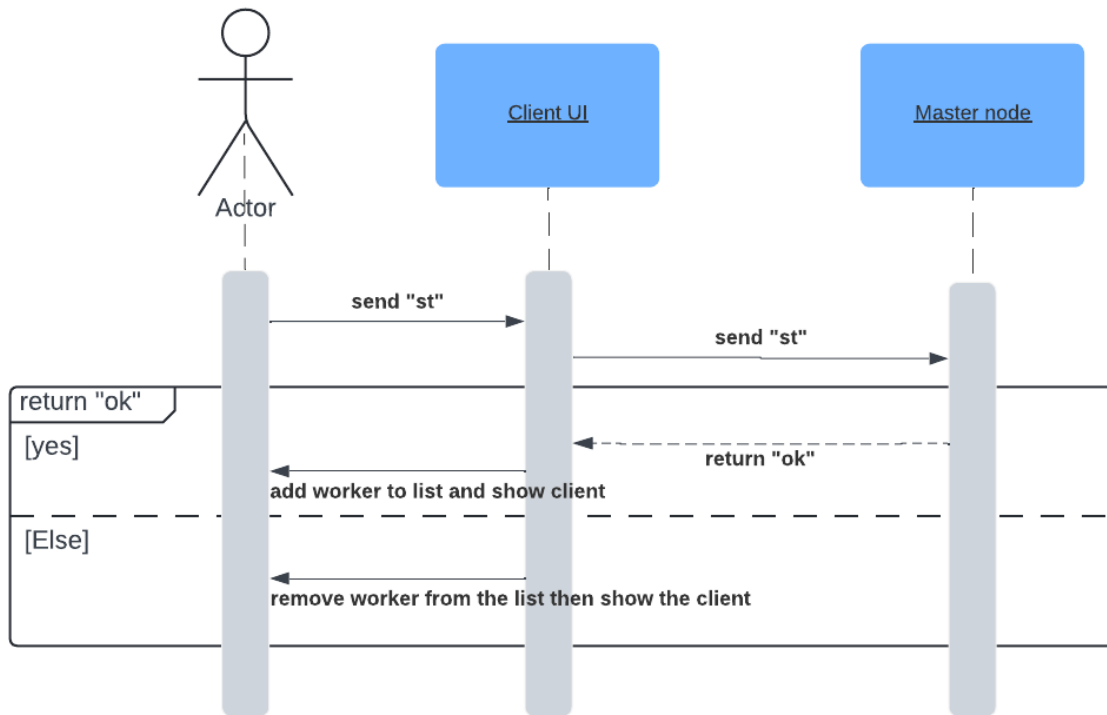


Figure 5 monitoring workernodes sequence diagram

## 2. Components diagram:

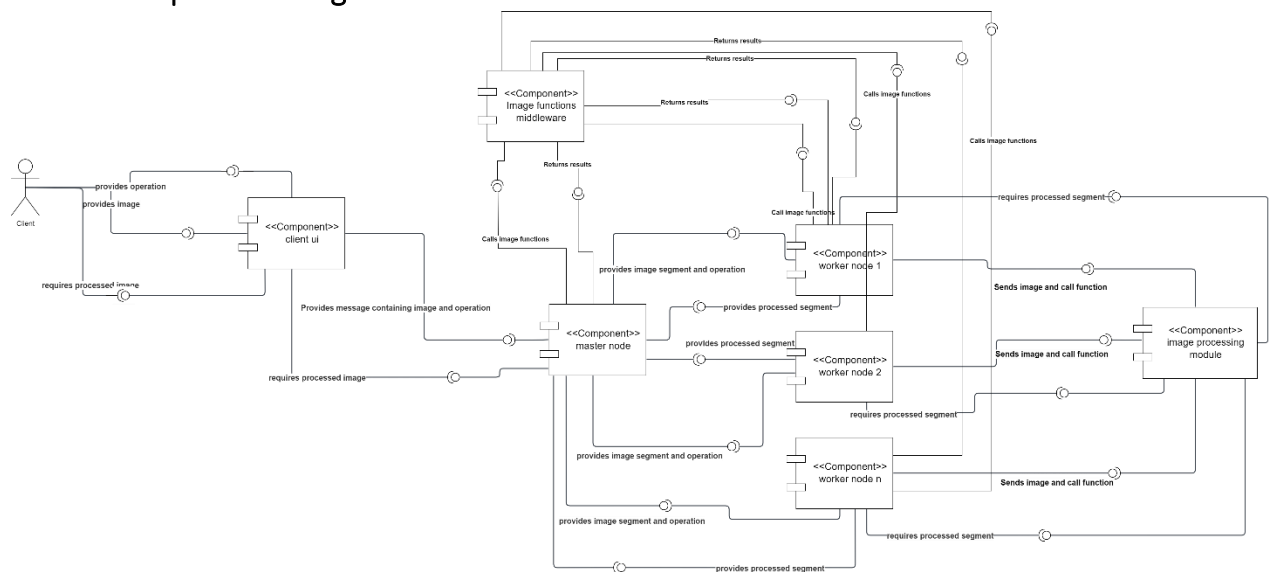


Figure 6 components diagram

### 3. Infrastructure diagram:

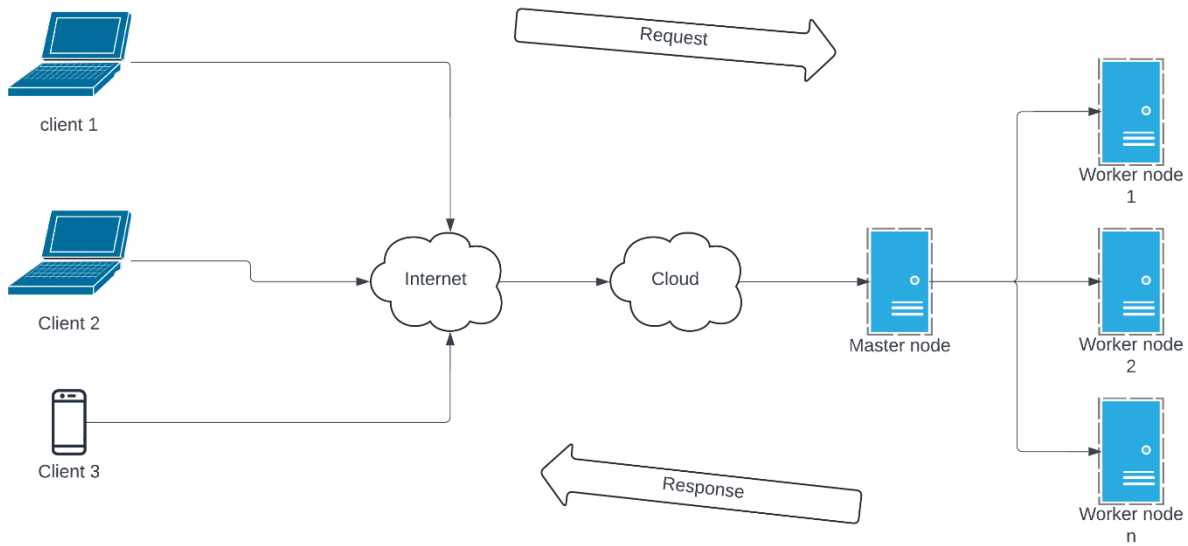


Figure 7 infrastructure diagram

### 4. Network diagram:

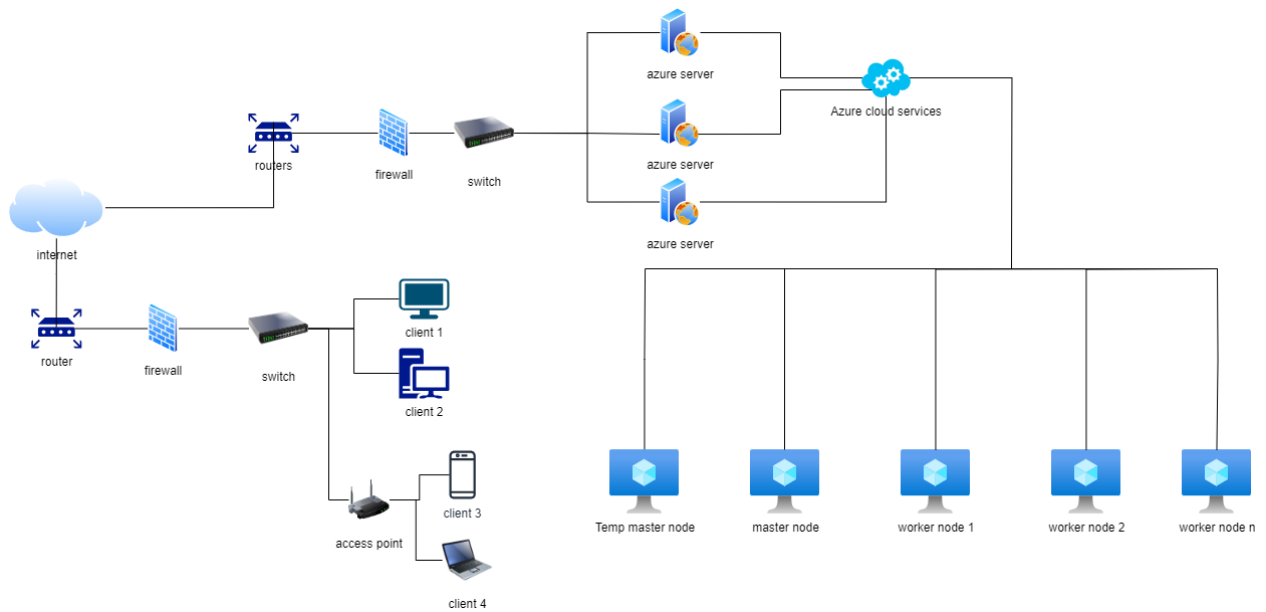


Figure 8 network diagram

# 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 the image files from your device.
- **Step 2:** Choose the image files 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, sketch, etc.
- **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:

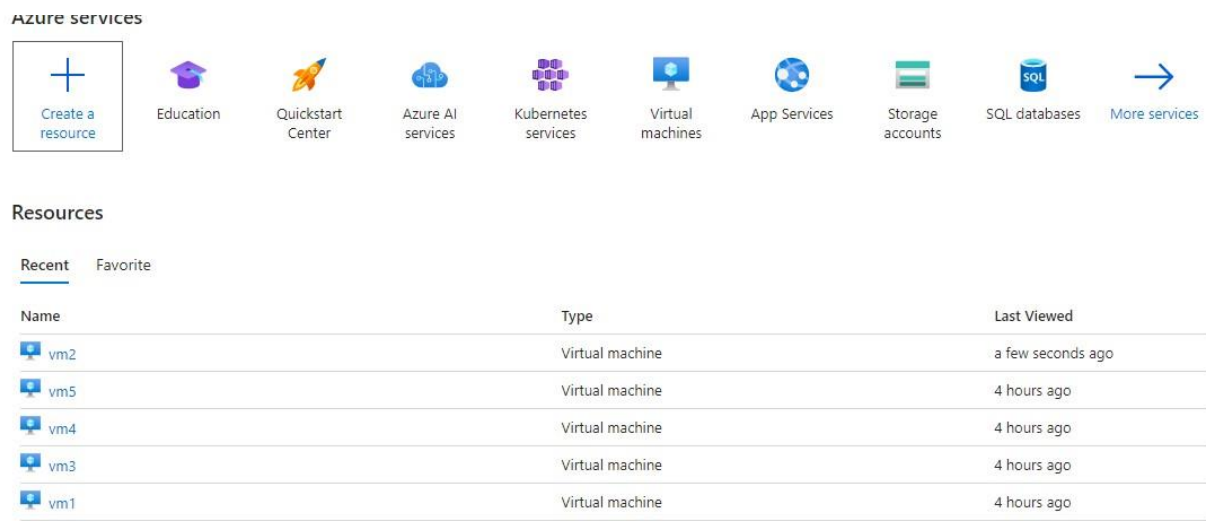
- If you encounter any issues or have questions about the system's functionality, refer to the system documentation or contact your system administrator for assistance.
- Ensure that you have the necessary permissions to access and use the system features effectively.

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

### Setting up the environment:

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

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



The screenshot displays the Azure portal interface. At the top, under 'Azure services', there is a row of icons for various services: 'Create a resource', 'Education', 'Quickstart Center', 'Azure AI services', 'Kubernetes services', 'Virtual machines', 'App Services', 'Storage accounts', 'SQL databases', and 'More services'. Below this, the 'Resources' section is visible, with tabs for 'Recent' and 'Favorite'. The 'Recent' tab is active, showing a table of resources.

Name	Type	Last Viewed
vm2	Virtual machine	a few seconds ago
vm5	Virtual machine	4 hours ago
vm4	Virtual machine	4 hours ago
vm3	Virtual machine	4 hours ago
vm1	Virtual machine	4 hours ago

Figure 9 azure resources

Then we downloaded RDP files for both machine to start them

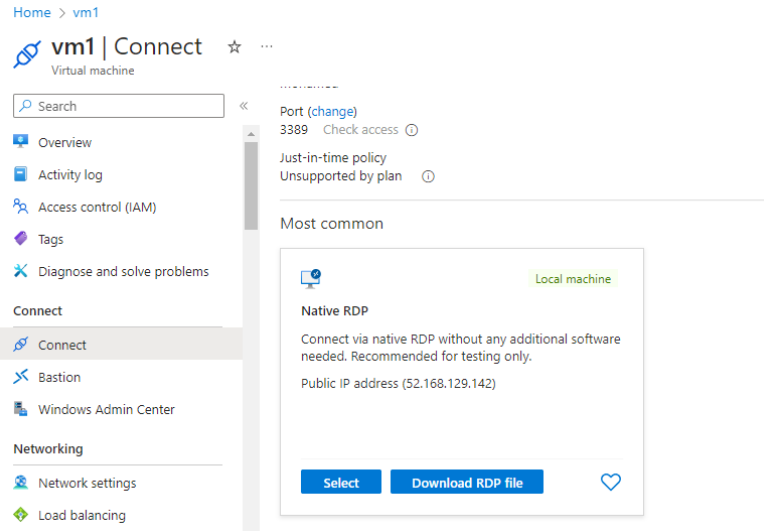


Figure 10 RDP download

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:

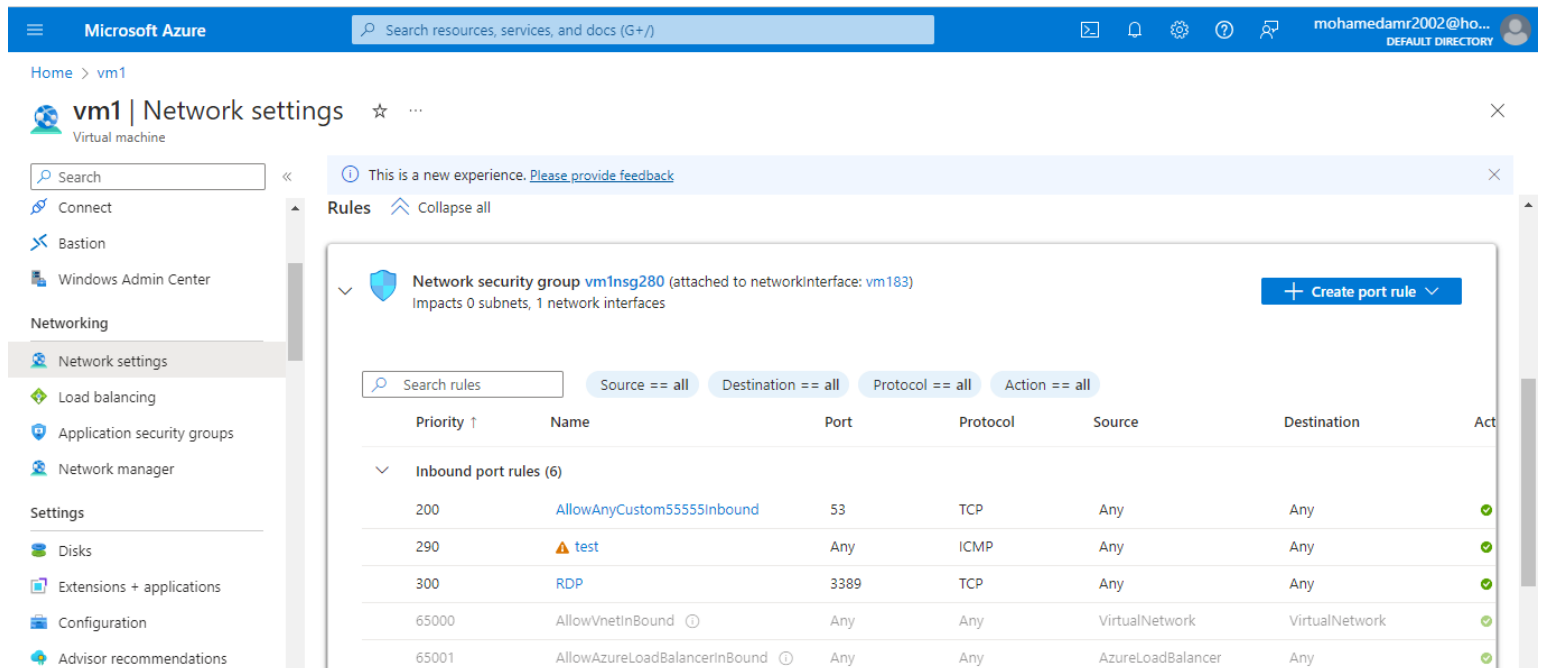


Figure 11 inbound rules

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 12 pinging machines

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

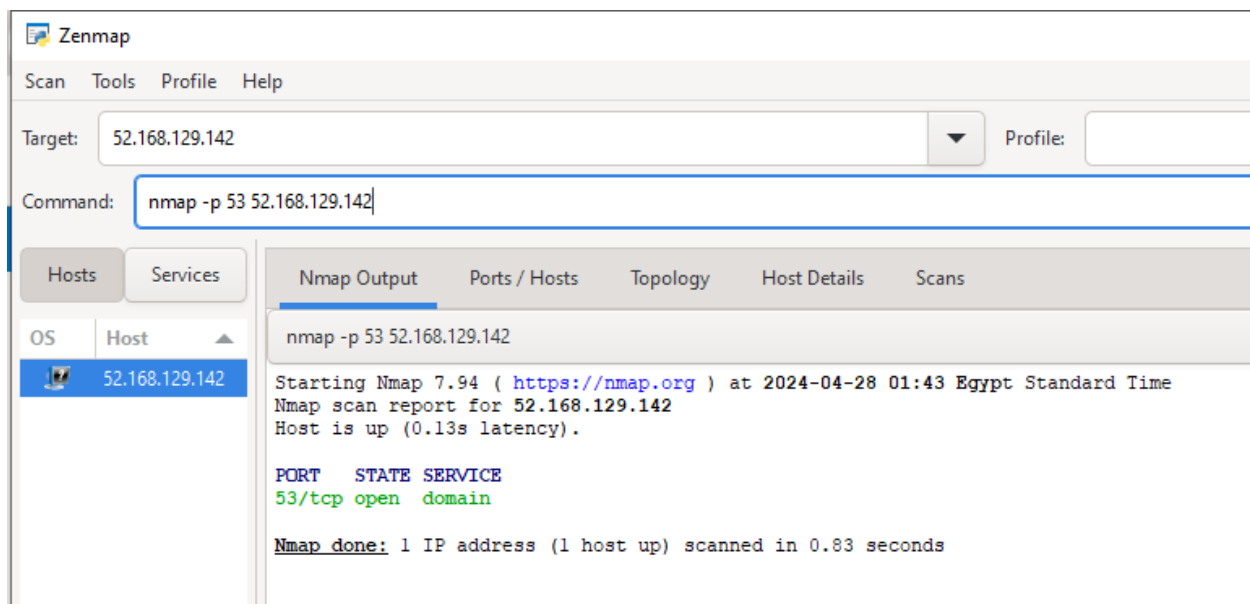


Figure 13 testing the TCP port

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



Codes:

1. Master node:

```
import socket
import threading
import json
from imageFunctionsMiddleware import *
# from db import *
from datetime import datetime
import urllib.request
workerslist=[('40.82.152.147',53),('20.215.232.32',53),('20.28.42.106',53)]#[('localhost',55555),('localhost',55554),('localhost',55553)]#
print(len(workerslist))

def send_list_over_socket(client_socket, data):
    try:
        serialized_data = json.dumps(data)
        buffer_size = len(serialized_data)
        client_socket.send(str(buffer_size).encode('utf-8'))
        client_socket.recv(2)
        client_socket.sendall(serialized_data.encode('utf-8'))
    except Exception as e:
        print(e)

def monitorWorker(server_public_ip, port, clientsockloggedonmaster,i):
    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":
            print("Worker is still alive")
        else:
            print("Worker is dead")
    except ConnectionRefusedError:
        print("Connection to worker failed. Worker might be down.")

def monitorWorker2(server_public_ip, port):
    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":

            print("Worker is still alive")
            return message
        else:
            print("Worker is dead")
    except ConnectionRefusedError:
        return "no"

def chechWorkinworkers(workerslist):
    workingWorkerlists=[]
    for i,worker in enumerate(workerslist):
        ip,ports=worker
        try:
            message=monitorWorker2(ip,ports)
            if message == "ok":
                # insert_log(f"{worker} is still alive {datetime.now()}") # Log
message to database
                if worker not in workingWorkerlists:
                    workingWorkerlists.append(worker)
            else:
                # insert_log(f"{worker} is dead {datetime.now()} ")
                if worker in workingWorkerlists:
                    workingWorkerlists.remove(worker)
        except Exception as e:
            print(e)
    return workingWorkerlists

def recieveAndSendClient():
    # insert_log(f"{get_public_ip()} worker started {datetime.now()}")
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = 'localhost' #'0.0.0.0' #'localhost'
    port = 55551#55551
    server_socket.bind((host, port))
    server_socket.listen(5)
    print(f"Server listening on {host}:{port}")
    workingWorkerlistscheckst=chechWorkinworkers(workerslist)
    while True:
        client_socket, addr = server_socket.accept()
        try:
            operation=client_socket.recv(2).decode('utf-8')
        except Exception as e:

```

```

        print(e)
    if operation == "st":
        workingWorkerlistscheckst=chechWorkinworkers(workerslist)
        try:
            send_list_over_socket(client_socket,workingWorkerlistscheckst)
        except Exception as E:
            print(E)
    elif operation=="mc":#master check
        client_socket.send("ok".encode('utf-8'))
    else:
        imageBytes,_=receive_image(client_socket)
        client_thread = threading.Thread(target=sendImageToWorker,
args=(client_socket,imageBytes,operation,addr))
        client_thread.start()

def sendImageToWorker(clientsockloggedonmaster,image_bytes,operation,addr):
    print(f"Connection from: {addr}")
    workingworkers=chechWorkinworkers(workerslist)
    segments = split_image(len(workingworkers), image_bytes)
    clientsockets=[]
    processed_segments_bytes = []
    for i,worker in enumerate(workingworkers):
        ip,ports=worker
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        try:
            client_socket.connect((ip, ports))
            client_socket.send(operation.encode('utf-8'))
            send_image_segments(client_socket, segments[i])
            clientsockets.append(client_socket)
        except:
            sendImageToWorker(clientsockloggedonmaster,image_bytes,operation,addr)
            #recursion to check if segment is failed its processes the image again

    for socketi in clientsockets:
        processed_segment_bytes, _ = receive_image(socketi)
        # display_image_from_bytes(processed_segment_bytes)
        processed_segments_bytes.append(processed_segment_bytes)
    if len(processed_segments_bytes)==len(segments):
        combined_image_path =
combine_segments_to_bytes(processed_segments_bytes)
        # display_image_from_bytes(combined_image_path)
        send_image_segments(clientsockloggedonmaster,combined_image_path)
    else:

```

```

        sendImageToWorker(clientsockloggedonmaster,image_bytes,operation,addr)#re
cursion to check if segment is failed its processes the image again
        client_socket.close()

def get_public_ip():
    try:
        response = urllib.request.urlopen('http://httpbin.org/ip')
        data = json.loads(response.read().decode())
        ip_address = data['origin']
        return ip_address
    except Exception as e:
        print("Error getting public IP:", e)
        return None

if __name__ == "__main__":
    recieveAndSendClient()

```

## 2. Worker node:

```

import socket
import threading
from imageFunctionsMiddleware import *
from imageProcessingModule import *
# from db import *
import urllib.request
import json
from datetime import datetime

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

            elif message=="st":
                client_socket.send("ok".encode('utf-8'))
                client_socket.close()

            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)))
            client_socket.close()
        elif message == "ed":
            edges_bytes = edgeDetection(image_bytes)
            send_image_knownbytes(client_socket, (edges_bytes,
len(edges_bytes)))
            client_socket.close()
        elif message == "fl":
            filtered_image_bytes = imageFiltering(image_bytes)
            send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
            client_socket.close()
        elif message == "bl":
            filtered_image_bytes = gaussian_blur(image_bytes)
            send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
            client_socket.close()
        elif message == "sk":
            filtered_image_bytes = laplacian(image_bytes)
            send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
            client_socket.close()
        elif message == "iv":
            filtered_image_bytes = invert_colors(image_bytes)
            send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
            client_socket.close()
        elif message == "bc":
            filtered_image_bytes =
adjust_brightness_contrast(image_bytes)
            send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
            client_socket.close()
        elif message == "rf":
            filtered_image_bytes = apply_red_filter(image_bytes)
            send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
            client_socket.close()
        elif message == "bf":
            filtered_image_bytes = apply_blue_filter(image_bytes)
            send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))

```

```

        client_socket.close()
    elif message == "gf":
        filtered_image_bytes =
apply_green_filter(image_bytes)
        send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
        client_socket.close()
    elif message == "cc":
        filtered_image_bytes =
convert_color_space(image_bytes)
        send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
        client_socket.close()
    elif message == "hm":
        filtered_image_bytes = apply_heat_filter(image_bytes)
        send_image_knownbytes(client_socket,
(filtered_image_bytes, len(filtered_image_bytes)))
        client_socket.close()
    else:
        print(f"Unknown message: {message} enter right
choice")

        continue
except Exception as e:
    print(f"Error receiving image: {e}")
    # insert_log(f"worker {get_public_ip()} closed
{e} {datetime.now()}")
    break
except Exception as error:
    # insert_log(f"worker {get_public_ip()} closed
{error} {datetime.now()}")
    client_socket.close()

def get_public_ip():
    try:
        response = urllib.request.urlopen('http://httpbin.org/ip')
        data = json.loads(response.read().decode())
        ip_address = data['origin']
        return ip_address
    except Exception as e:
        print("Error getting public IP:", e)
        return None

```

```

def main():
    # insert_log(f"{get_public_ip()} worker started {datetime.now()}")
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = '0.0.0.0' #'0.0.0.0' #'localhost'
    port = 53 #55555
    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.daemon=True
        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
import time
import json
import sys

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_frame = tk.Frame(self.scrollable_frame)
    label = tk.Label(label_frame, image=photo)
    label.image = photo
    label.pack(pady=5, side="top")
    download_button = tk.Button(label_frame, text="Download", command=lambda:
self.save_image(image))
    download_button.pack(side="bottom")
    label_frame.pack(pady=5, padx=5)

def save_image(self, image):
    default_filename="image.png"
    file_path =
filedialog.asksaveasfilename(defaultextension=".png",initialfile=default_filename
, filetypes=[("PNG files", "*.png"), ("JPEG files", "*.jpg"), ("All files",
"*.*)"])
    if file_path:
        image.save(file_path)

class ImageConverterApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Image Converter")
        self.root.geometry("550x550")
        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.uploaded_images = []
        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 sharpening","blur","sketch","invert colors","brightness
contrast","red filter","blue filter","green filter","convert color","heat map")
        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)
        self.success_fail_label = tk.Label(root, text="", fg="green")
        self.success_fail_label.pack()
        self.masters_label = tk.Label(root, text="Masters Status: Unknown")
        self.masters_label.pack()
        self.server_status_label = tk.Label(root, text="workers Status: Unknown")
        self.server_status_label.pack()
        self.masters=[("51.120.112.111",53),("4.232.128.42",53)]#[("localhost",55
550),("localhost",55551)]#
        self.workingmasterslists=[]
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)

    def on_closing(self):
        self.root.destroy()
        sys.exit(0)

    def upload_image(self):
        self.uploaded_images = []
        file_types = [("Image Files", "*.jpg; *.jpeg; *.png; *.gif; *.bmp")]
        file_paths = filedialog.askopenfilenames(filetypes=file_types)

        if file_paths:
            for file_path in file_paths:
                image = Image.open(file_path)
                self.uploaded_images.append(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()

    if imagePath.lower().endswith('.png'):
        image = Image.open(io.BytesIO(image_bytes)).convert('RGB')
        output = io.BytesIO()
        image.save(output, format='JPEG')
        image_bytes = output.getvalue()
    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):
    thread=threading.Thread(target=self.convert_image)
    thread.daemon = True
    thread.start()

def receive_list_from_socket(self,client_socket):
    buffer_size_str = client_socket.recv(1024).decode('utf-8')
    buffer_size = int(buffer_size_str)
    client_socket.send(b'OK') # Send acknowledgment
    received_data = b''
    while len(received_data) < buffer_size:
        received_data += client_socket.recv(min(buffer_size -
len(received_data), 1024))
    return json.loads(received_data.decode('utf-8'))

def receive_server_status(self):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ip,port=self.workingmasterslists[0]
    try:
        client_socket.connect((ip, port))
        client_socket.send("st".encode('utf-8'))
        message = self.receive_list_from_socket(client_socket)
        print(message)
        if message=="ok":
            return "active"
        elif message=="no":
            return "error"
        return message
    except Exception as E:
        print("Connection to server failed.")
        return "Error in Master Node"

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

def monitor_server_status(self):
    while True:
        try:
            status = self.receive_server_status()

```

```

        self.server_status_label.config(text=f"Available workers
({len(status)}): {status}")

        time.sleep(1)
    except Exception as E:
        print(E)
        continue

def monitormaster(self,server_public_ip, port):
    try:
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client_socket.connect((server_public_ip, port))
        client_socket.send("mc".encode('utf-8'))
        message = client_socket.recv(2).decode('utf-8')
        if message == "ok":
            print("master is still alive")
            return message
        else:
            print("master is dead")
    except ConnectionRefusedError:
        return "no"

def chechWorkingmasters(self):

    while True:
        try:
            for i, master in enumerate(self.masters):
                ip, ports = master
                message = self.monitormaster(ip, ports)
                if message == "ok":
                    if master not in self.workingmasterslists:
                        self.workingmasterslists.append(master)
                else:
                    if master in self.workingmasterslists:
                        self.workingmasterslists.remove(master)
            self.masters_label.config(text=f"Available Masters
({len(self.workingmasterslists)}): {self.workingmasterslists}")

            time.sleep(1)
        except Exception as e:
            print(e)
            continue

def monitor_masters_thread(self):

```

```

thread=threading.Thread(target=self.checkWorkingmasters)
thread.daemon = True # Make the thread a daemon thread
thread.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 sharpening":
        option="fl"
    elif option=="blur":
        option="bl"
    elif option=="sketch":
        option="sk"
    elif option=="invert colors":
        option="iv"
    elif option=="brightness contrast":
        option="bc"
    elif option=="red filter":
        option="rf"
    elif option=="blue filter":
        option="bf"
    elif option=="green filter":
        option="gf"
    elif option=="convert color":
        option="cc"
    elif option=="heat map":
        option="hm"

    sockets=[]

    print(self.uploaded_images)
    for path in self.uploaded_images:

        ip,port=self.workingmasterslists[0]
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client_socket.connect((ip, port))
        client_socket.send(option.encode('utf-8'))
        self.send_image(client_socket,path)
        sockets.append(client_socket)

```

```

        try:
            for client_socket in sockets:
                imageBytes, _ = self.receive_image(client_socket)
                imageBytes = self.bytes_to_image(imageBytes)
                processedImages.append(imageBytes)
                self.success_fail_label.config(text="Conversion successful",
fg="green")
            except Exception as E:
                self.success_fail_label.config(text="Conversion failed, Try again",
fg="red")
            for x in processedImages:
                self.scrollable_frame.add_image(x)

if __name__ == "__main__":
    root = tk.Tk()
    app = ImageConverterApp(root)
    app.monitor_masters_thread()
    time.sleep(4) #this delay is to wait for the system to get the working
masters
    app.monitor_server_status_thread()
    root.mainloop()

```

#### 4. Images functions middleware:

```

import cv2
import numpy as np

def greyFilter(image_bytes):
    image = cv2.imdecode(np.frombuffer(image_bytes, np.uint8), cv2.IMREAD_COLOR)
    processed_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    processed_image_bytes = cv2.imencode('.jpg', processed_image)[1].tobytes()
    return processed_image_bytes

def edgeDetection(image_bytes):
    image = cv2.imdecode(np.frombuffer(image_bytes, np.uint8), cv2.IMREAD_COLOR)
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    edges = cv2.Canny(gray_image, 100, 200)
    edges_bytes = cv2.imencode('.jpg', edges)[1].tobytes()

```

```

    return edges_bytes

def imageFiltering(image_bytes):
    image = cv2.imdecode(np.frombuffer(image_bytes, np.uint8), cv2.IMREAD_COLOR)
    blurred_image = cv2.GaussianBlur(image, (5, 5), 0)
    kernel_sharpening = np.array([[-1, -1, -1],
                                   [-1, 9, -1],
                                   [-1, -1, -1]])
    sharpened_image = cv2.filter2D(blurred_image, -1, kernel_sharpening)
    filtered_image_bytes = cv2.imencode('.jpg', sharpened_image)[1].tobytes()
    return filtered_image_bytes

def gaussian_blur(byte_image, kernel_size=(31, 31)):
    """Apply Gaussian blur to the byte image"""
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_COLOR)
    blurred_image = cv2.GaussianBlur(image, kernel_size, 0)
    _, img_encoded = cv2.imencode('.jpg', blurred_image)
    return img_encoded.tobytes()

def laplacian(byte_image):
    """Apply Laplacian filter to the byte image"""
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_GRAYSCALE)
    laplacian_image = cv2.Laplacian(image, cv2.CV_64F)
    laplacian_image = np.uint8(np.absolute(laplacian_image))
    _, img_encoded = cv2.imencode('.jpg', laplacian_image)
    return img_encoded.tobytes()

def invert_colors(byte_image):
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_COLOR)
    inverted_image = cv2.bitwise_not(image)
    _, img_encoded = cv2.imencode('.jpg', inverted_image)
    return img_encoded.tobytes()

def apply_red_filter(byte_image):
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_COLOR)
    filtered_image = np.zeros_like(image)
    filtered_image[:, :, 2] = image[:, :, 2]
    _, img_encoded = cv2.imencode('.jpg', filtered_image)
    return img_encoded.tobytes()

```

```

def apply_green_filter(byte_image):
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_COLOR)
    filtered_image = np.zeros_like(image)
    filtered_image[:, :, 1] = image[:, :, 1]
    _, img_encoded = cv2.imencode('.jpg', filtered_image)
    return img_encoded.tobytes()

def apply_blue_filter(byte_image):
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_COLOR)
    filtered_image = np.zeros_like(image)
    filtered_image[:, :, 0] = image[:, :, 0]
    _, img_encoded = cv2.imencode('.jpg', filtered_image)
    return img_encoded.tobytes()

def adjust_brightness_contrast(byte_image, alpha=1.5, beta=40):
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_COLOR)
    adjusted_image = cv2.convertScaleAbs(image, alpha=alpha, beta=beta)
    _, img_encoded = cv2.imencode('.jpg', adjusted_image)
    return img_encoded.tobytes()

def convert_color_space(byte_image, target_color_space=cv2.COLOR_BGR2HSV):
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_COLOR)
    converted_image = cv2.cvtColor(image, target_color_space)
    _, img_encoded = cv2.imencode('.jpg', converted_image)
    return img_encoded.tobytes()

def apply_heat_filter(byte_image):
    image = np.frombuffer(byte_image, dtype=np.uint8)
    image = cv2.imdecode(image, cv2.IMREAD_GRAYSCALE)
    heatmap = cv2.applyColorMap(image, cv2.COLORMAP_JET)

    _, img_encoded = cv2.imencode('.jpg', heatmap)
    return img_encoded.tobytes()

```

## 5. Database:

```

from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi
import certifi

```



```

uri =
"mongodb+srv://mohamedamr2002a:9IfsisX1Tg861u5C@cluster0.mlzj01e.mongodb.net/?ret
ryWrites=true&w=majority&appName=Cluster0"

# Create a new client and connect to the server
client = MongoClient(uri,tlsCAFile=certifi.where())

db=client.get_database("distributed_db")

logs_collection = db.get_collection("logs")

def insert_log(log_message):
    log_document = {'log': log_message}
    logs_collection.insert_one(log_document)

def view_logs():
    logs = logs_collection.find()
    for i, log in enumerate(logs):
        print(f"{i}:{log['log']}")

```

## 6. Get Logs:

```

from db import *
view_logs()

```

Analysis of the codes:

### 1. Master node:

#### Functions and Their Features

##### 1. send\_list\_over\_socket(client\_socket, data):

- **Purpose:** Sends a list of data over a socket connection.
- **Features:**
  - Serializes the data to JSON format.
  - Sends the size of the serialized data to the client.
  - Waits for an acknowledgment from the client before sending the actual data.
  - Uses **try-except** for basic error handling.

2. **monitorWorker(server\_public\_ip, port, clientsockloggedonmaster, i):**

- **Purpose:** Checks if a worker node is alive by attempting to connect and communicate with it.
- **Features:**
  - Creates a socket connection to the worker.
  - Sends a status request ("st").
  - Receives a response and prints whether the worker is alive or dead.
  - Handles **ConnectionRefusedError** to indicate the worker might be down.

3. **monitorWorker2(server\_public\_ip, port):**

- **Purpose:** Similar to **monitorWorker**, but returns the status directly.
- **Features:**
  - Sends a status request ("st").
  - Returns "ok" if the worker responds correctly, otherwise returns "no".
  - Handles **ConnectionRefusedError** to indicate the worker might be down.

4. **checkWorkinworkers(workerslist):**

- **Purpose:** Checks which workers from the **workerslist** are alive and returns a list of them.
- **Features:**
  - Iterates through the list of workers.
  - Uses **monitorWorker2** to check each worker's status.
  - Maintains and returns a list of workers that are alive.

5. **recieveAndSendClient():**

- **Purpose:** Main server function to receive client connections and dispatch tasks.
- **Features:**
  - Sets up a server socket to listen for incoming connections.
  - Handles different operations based on client requests:
    - **"st"**: Checks and sends back the list of working workers.
    - **"mc"**: Sends an acknowledgment ("ok").

- Other operations: Receives an image from the client and processes it by dispatching to workers.
  - Uses threading to handle image processing in parallel.
6. **sendImageToWorker(clientsockloggedonmaster, image\_bytes, operation, addr):**
- **Purpose:** Distributes image processing tasks to the workers and sends the results back to the client.
  - **Features:**
    - Checks which workers are alive.
    - Splits the image into segments based on the number of alive workers.
    - Sends each segment to a corresponding worker.
    - Collects the processed segments from the workers.
    - Combines the processed segments into a complete image.
    - Sends the combined image back to the client.
    - Uses recursion to retry if any segment fails to be processed.
7. **get\_public\_ip():**
- **Purpose:** Retrieves the public IP address of the server.
  - **Features:**
    - Uses **urllib.request** to make an HTTP request to an external service.
    - Parses the response to extract the public IP address.
    - Handles exceptions to return **None** if the request fails.

## Summary

Each function in this code plays a specific role in setting up a distributed image processing system, where:

- **send\_list\_over\_socket:** Handles the sending of data over a socket.
- **monitorWorker and monitorWorker2:** Check the status of worker nodes.
- **checkWorkinworkers:** Returns a list of currently active workers.
- **recieveAndSendClient:** Acts as the main server, handling client requests and delegating tasks.

- **sendImageToWorker:** Manages the distribution of image processing tasks among workers and combines the results.
- **get\_public\_ip:** Retrieves the server's public IP address for logging or other purposes.

These functions together facilitate the distributed processing of images, ensuring that tasks are correctly assigned and results are accurately combined and returned to the client.

## 2. Worker node:

### Functions and Their Features

#### 1. **handle\_client(client\_socket, addr):**

- **Purpose:** Manages communication with a connected client, processes image operations based on client requests, and closes the connection after processing each request.
- **Features:**
  - **Connection Handling:**
    - Prints the address of the connected client.
    - Enters an infinite loop to continuously handle client requests.
  - **Message Handling:**
    - Receives a 2-byte message from the client.
    - If the message is "st", it sends an "ok" response to confirm the worker is alive and closes the connection.
  - **Image Processing:**
    - Receives an image from the client.
    - Applies the appropriate image processing function from the **imageProcessingModule** based on the client's request.
    - Sends the processed image back to the client.
    - Closes the connection after processing each request.
  - **Error Handling:**
    - Catches exceptions during image reception and processing, printing error messages.

- Closes the client socket in case of errors or after processing each request.

## 2. `get_public_ip()`:

- **Purpose:** Retrieves the public IP address of the server.
- **Features:**
  - Makes an HTTP request to an external service to get the public IP.
  - Parses the JSON response to extract the IP address.
  - Handles exceptions and prints an error message if the request fails, returning **None**.

## 3. `main()`:

- **Purpose:** Initializes the server, listens for incoming client connections, and spawns threads to handle each connection.
- **Features:**
  - **Server Initialization:**
    - Creates a server socket.
    - Binds the server to all available interfaces ('0.0.0.0') on port **53**.
    - Starts listening for incoming connections.
  - **Connection Handling:**
    - Enters an infinite loop to accept client connections.
    - Spawns a new thread for each client connection to handle them concurrently.
    - Starts each thread to handle client communication.

## Summary

The provided code sets up a server that handles client connections, processes image operations, and responds to clients. Here's a summary of its functionality:

- **handle\_client:**
  - Handles communication with clients, processes image operations, and closes connections after processing each request.
- **get\_public\_ip:**

- Retrieves and returns the public IP address of the server.
- **main:**
  - Initializes the server, listens for incoming connections, and spawns threads to handle client connections concurrently.

## Potential Issues and Improvements

### 1. Resource Management:

- Ensure that all sockets are properly closed after use to avoid resource leaks.
- The code currently closes the client socket after processing each request, which is good practice.

### 2. Error Handling:

- Ensure that exceptions are properly handled and logged for debugging purposes.
- Consider implementing retries or error recovery mechanisms for robustness.

### 3. Concurrency:

- Thread management and concurrency seem appropriate for handling multiple clients simultaneously.

### 4. Security:

- Validate user inputs and sanitize them to prevent potential security vulnerabilities like injection attacks.

### 5. Performance Optimization:

- Consider optimizing image processing functions for performance, especially if dealing with large images or high concurrency.

### 6. Port Selection:

- Port 53 is typically reserved for DNS services. Consider using a different port for your application to avoid conflicts.

### 3. Client GUI:

This code is a GUI application built using Tkinter for converting and displaying images. Let's break down its functionality and features:

## Classes and Their Methods

## 1. ScrollableImageFrame:

- **Purpose:** Provides a frame with a scrollable area for displaying images.
- **Features:**
  - Utilizes Tkinter's Canvas and Scrollbar widgets to create a scrollable frame.
  - Provides a method `add_image` to add images to the scrollable frame along with a download button for each image.

## 2. ImageConverterApp:

- **Purpose:** Main application class for the image converter GUI.
- **Features:**
  - Initializes the GUI window and various widgets such as buttons, labels, and option menus.
  - Provides methods for uploading images, converting images, monitoring server and master status, and handling image-related operations.
  - Utilizes threading to perform tasks such as monitoring server and master status concurrently.
  - Allows users to select image processing options from an option menu.
  - Displays success or failure messages for image conversion.
  - Monitors the status of workers and masters through separate threads.
  - Uses sockets for communication with master and worker nodes.

### Key Methods and Their Functionality

- **upload\_image:** Allows users to upload one or more images, which are then displayed in the scrollable frame.
- **convert\_image\_thread:** Starts a thread to convert uploaded images based on the selected processing option.
- **receive\_list\_from\_socket:** Receives a list from a socket connection.
- **monitor\_server\_status\_thread and chechWorkingmasters:** Start threads to monitor the status of worker nodes and master nodes, respectively. They update labels in the GUI to display the status.

- **convert\_image:** Initiates the image conversion process by sending images to worker nodes for processing based on the selected option. It then receives and displays the processed images.

### Additional Features

- **Error Handling:** Some error handling is implemented, such as displaying failure messages for image conversion and handling exceptions in socket communication.
- **Dynamic GUI Updates:** The GUI dynamically updates labels to display the status of worker nodes and master nodes.
- **Multi-Threading:** Utilizes threading to perform tasks concurrently, such as monitoring server and master status while allowing the user to interact with the GUI.

## 4. Images functions middleware:

This code provides functions for handling images in byte format, splitting images into segments, combining segments back into a single image, displaying images, and sending/receiving images over a network connection. Let's break down each function:

### 1. split\_image(num\_segments, image\_bytes):

- Splits an image into multiple segments based on the number of segments specified.
- **Parameters:**
  - **num\_segments:** Number of segments to split the image into.
  - **image\_bytes:** Bytes representation of the image.
- Returns a list of byte representations of image segments.

### 2. combine\_segments\_to\_bytes(segments):

- Combines multiple image segments into a single image.
- **Parameters:**
  - **segments:** List of byte representations of image segments.
- Returns byte representation of the combined image.

### 3. display\_image\_from\_bytes(image\_bytes):

- Displays an image from its byte representation using PIL's Image.show() method.



- **Parameters:**
  - **image\_bytes:** Byte representation of the image.

#### 4. **receive\_image(conn):**

- Receives an image over a network connection.
- **Parameters:**
  - **conn:** Network connection object.
- Returns byte representation of the received image and its length.

#### 5. **send\_image(conn, imagePath):**

- Sends an image over a network connection.
- **Parameters:**
  - **conn:** Network connection object.
  - **imagePath:** Path to the image file to be sent.

#### 6. **send\_image\_segments(conn, image\_bytes):**

- Sends image segments over a network connection.
- **Parameters:**
  - **conn:** Network connection object.
  - **image\_bytes:** Bytes representation of the image.

#### 7. **send\_image\_knownbytes(conn, image):**

- Sends an image with its known byte size over a network connection.
- **Parameters:**
  - **conn:** Network connection object.
  - **image:** Tuple containing the image bytes and its size.

These functions provide a comprehensive set of tools for handling image data in byte format, whether for local processing or transmission over a network. They allow for tasks such as splitting, combining, displaying, and sending/receiving images efficiently.

## 5. Image processing module:

This code provides several image processing functions using the OpenCV library, each accepting image bytes as input and returning processed image bytes. Let's break down each function:

### 1. **greyFilter(image\_bytes):**

- Converts the image to grayscale.

### 2. **edgeDetection(image\_bytes):**

- Performs edge detection on the image using the Canny edge detector.

### 3. **imageFiltering(image\_bytes):**

- Applies a Gaussian blur to the image followed by sharpening using a custom kernel.

### 4. **gaussian\_blur(byte\_image, kernel\_size=(31, 31)):**

- Applies Gaussian blur to the image with a specified kernel size.

### 5. **laplacian(byte\_image):**

- Applies Laplacian edge detection to the image.

### 6. **invert\_colors(byte\_image):**

- Inverts the colors of the image.

### 7. **apply\_red\_filter(byte\_image), apply\_green\_filter(byte\_image), apply\_blue\_filter(byte\_image):**

- Retains only the red, green, or blue channels of the image, respectively.

### 8. **adjust\_brightness\_contrast(byte\_image, alpha=1.5, beta=40):**

- Adjusts the brightness and contrast of the image.

### 9. **convert\_color\_space(byte\_image, target\_color\_space=cv2.COLOR\_BGR2HSV):**

- Converts the color space of the image to the specified target color space.

### 10. **apply\_heat\_filter(byte\_image):**

- Applies a heatmap filter to the grayscale image.

These functions utilize the powerful image processing capabilities of OpenCV to perform a variety of tasks such as filtering, edge detection, color space conversion, and more. They're useful for tasks ranging from basic image enhancement to more advanced computer vision applications.

## 6. DB:

This script interacts with a MongoDB database using the PyMongo library to perform logging operations. Let's break down the code:

### 1. Import Statements:

- **from pymongo.mongo\_client import MongoClient:** Imports the **MongoClient** class from the **pymongo.mongo\_client** module, which is used to connect to MongoDB.
- **from pymongo.server\_api import ServerApi:** Imports the **ServerApi** class, which allows configuring server API version.

### 2. URI Definition:

- **uri:** Defines the connection URI for MongoDB. It contains credentials, cluster details, and other parameters necessary for establishing a connection. The **appName** parameter is set to **"Cluster0"**.

### 3. Client Connection:

- **client = MongoClient(uri,tlsCAFile=certifi.where()):** Creates a new **MongoClient** instance and connects to the MongoDB server specified by the URI. It uses the **certifi** library to locate the CA certificate file for TLS/SSL encryption.

### 4. Database and Collection:

- **db = client.get\_database("distributed\_db"):** Retrieves the database named **"distributed\_db"** from the MongoDB server.
- **logs\_collection = db.get\_collection("logs"):** Retrieves the collection named **"logs"** from the database. This collection is used for storing log documents.

### 5. Log Insertion Function:

- **insert\_log(log\_message):** Inserts a new log document into the **"logs"** collection. It takes a **log\_message** parameter and creates a document with a single field **"log"** containing the log message.

### 6. View Logs Function:

- **view\_logs():** Retrieves all log documents from the **"logs"** collection and prints them. It iterates over the cursor returned by **find()** and prints each log message along with its index.

Overall, this script provides functions for logging messages into a MongoDB database and viewing the logged messages. It's a simple yet effective way to track application activities and errors in a distributed environment.

## 7. View logs:

**view\_logs()** is used to retrieve and view these logs, possibly for monitoring or debugging purposes.

## Image processing:

1. **greyFilter**: Converts a color image into grayscale.
2. **edgeDetection**: Detects edges in an image using the Canny edge detection algorithm.
3. **imageFiltering**: Applies a Gaussian blur followed by a sharpening filter to enhance details in the image.
4. **gaussian\_blur**: Blurs the input image using a Gaussian blur filter.
5. **laplacian**: Applies Laplacian edge detection to the input grayscale image.
6. **invert\_colors**: Inverts the colors of the input image.
7. **apply\_red\_filter**: Filters the input image to show only the red channel.
8. **apply\_green\_filter**: Filters the input image to show only the green channel.
9. **apply\_blue\_filter**: Filters the input image to show only the blue channel.
10. **adjust\_brightness\_contrast**: Adjusts the brightness and contrast of the input image.
11. **convert\_color\_space**: Converts the color space of the input image to the specified target color space.
12. **apply\_heat\_filter**: Applies a heat map filter to the input grayscale image.

## Monitoring:

### Client gui:

We added fixed label at the bottom to get the server status and another one for master

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

```
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 to check worker 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
```

also we added another thread to check for masters

```
def monitormaster(self,server_public_ip, port):
```

```

try:
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((server_public_ip, port))
    client_socket.send("mc".encode('utf-8'))
    message = client_socket.recv(2).decode('utf-8')
    if message == "ok":
        print("master is still alive")
        return message
    else:
        print("master is dead")
except ConnectionRefusedError:
    return "no"

def chechWorkingmasters(self):

    while True:
        try:
            for i, master in enumerate(self.masters):
                ip, ports = master
                message = self.monitormaster(ip, ports)
                if message == "ok":
                    if master not in self.workingmasterslists:
                        self.workingmasterslists.append(master)
                else:
                    if master in self.workingmasterslists:
                        self.workingmasterslists.remove(master)
                self.masters_label.config(text=f"Available Masters
({len(self.workingmasterslists)}): {self.workingmasterslists}")

            time.sleep(1)
        except Exception as e:
            print(e)
            continue

def monitor_masters_thread(self):
    thread = threading.Thread(target=self.chechWorkingmasters)
    thread.daemon = True # Make the thread a daemon thread
    thread.start()

```

Then we called the thread at the main loop

```

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

```

```

app.monitor_masters_thread()
time.sleep(4) #this delay is to wait for the system to get the working
masters
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.")

```

and to check for working workers and send this list to client

```

def chechWorkinworkers(workerslist):

```

```

workingWorkerlists=[]
for i,worker in enumerate(workerslist):
    ip,ports=worker
    try:
        message=monitorWorker2(ip,ports)
        if message == "ok":
            # insert_log(f"{worker} is still alive {datetime.now()}") # Log
message to database
            if worker not in workingWorkerlists:
                workingWorkerlists.append(worker)
        else:
            # insert_log(f"{worker} is dead {datetime.now()} ")
            if worker in workingWorkerlists:
                workingWorkerlists.remove(worker)
    except Exception as e:
        print(e)
return workingWorkerlists

```

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'))

```

And in the master we added check to send its status to the client to add it in the working masters list

```

elif operation=="mc":#master check
    client_socket.send("ok".encode('utf-8'))

```

## Fault tolerance

For fault tolerance we created another secondary master node to switch to in case of any error in our primary master node to be able to make the user to convert images again



And in worker nodes in case of failure during the processing the image is segmented and sent again in the number of the working worker nodes by recursion as we check if the number of processed segment is equal to the number of sent segments:

```
def sendImageToWorker(clientsockloggedonmaster,image_bytes,operation,addr):
    print(f"Connection from: {addr}")
    workingworkers=chechWorkinworkers(workerslist)
    segments = split_image(len(workingworkers), image_bytes)
    clientsockets=[]
    processed_segments_bytes = []
    for i,worker in enumerate(workingworkers):
        ip,ports=worker
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        try:
            client_socket.connect((ip, ports))
            client_socket.send(operation.encode('utf-8'))
            send_image_segments(client_socket, segments[i])
            clientsockets.append(client_socket)
        except:
            sendImageToWorker(clientsockloggedonmaster,image_bytes,operation,addr)
    )#recursion to check if segment is failed its processes the image again

    for socketi in clientsockets:
        processed_segment_bytes, _ = receive_image(socketi)
        # display_image_from_bytes(processed_segment_bytes)
        processed_segments_bytes.append(processed_segment_bytes)
    if len(processed_segments_bytes)==len(segments):
        combined_image_path =
combine_segments_to_bytes(processed_segments_bytes)
        # display_image_from_bytes(combined_image_path)
        send_image_segments(clientsockloggedonmaster,combined_image_path)
    else:
        sendImageToWorker(clientsockloggedonmaster,image_bytes,operation,addr)#re
cursion to check if segment is failed its processes the image again
        client_socket.close()
```

## Scalability:

For scalability we segmented the photo in the number of working worker nodes:

```
workingworkers=chechWorkinworkers(workerslist)
    segments = split_image(len(workingworkers), image_bytes)
    clientsockets=[]
```

```

processed_segments_bytes = []
for i,worker in enumerate(workingworkers):
    ip,ports=worker
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        client_socket.connect((ip, ports))
        client_socket.send(operation.encode('utf-8'))
        send_image_segments(client_socket, segments[i])
        clientsockets.append(client_socket)

```

## Parallelizing:

For parallel part we send the segment to all the worker nodes first and we append the socket that we sent with to sockets lists to make another for loop iterating on this sockets list to receive all the processed segment from all the sockets:

```

clientsockets=[]
processed_segments_bytes = []
for i,worker in enumerate(workingworkers):
    ip,ports=worker
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        client_socket.connect((ip, ports))
        client_socket.send(operation.encode('utf-8'))
        send_image_segments(client_socket, segments[i])
        clientsockets.append(client_socket)
    except:
        sendImageToWorker(clientsockloggedonmaster,image_bytes,operation,addr)
        #recursion to check if segment is failed its processes the image again

    for socketi in clientsockets:
        processed_segment_bytes, _ = receive_image(socketi)
        # display_image_from_bytes(processed_segment_bytes)
        processed_segments_bytes.append(processed_segment_bytes)

```

so all the workers process the segments at the same time then we receive it by order of the sent sockets then we combine it and send the full image to the client.

## Database and logging:

For database and logging we used pymongo online cluster to be able to submit from all the machines but the connection to it takes a large time for logging and submitting the log in the database so we

commented it in the code to use the code faster on the cloud virtual machines with low computing performance.

And we create python file to view logs from the database when we ran it in the code:

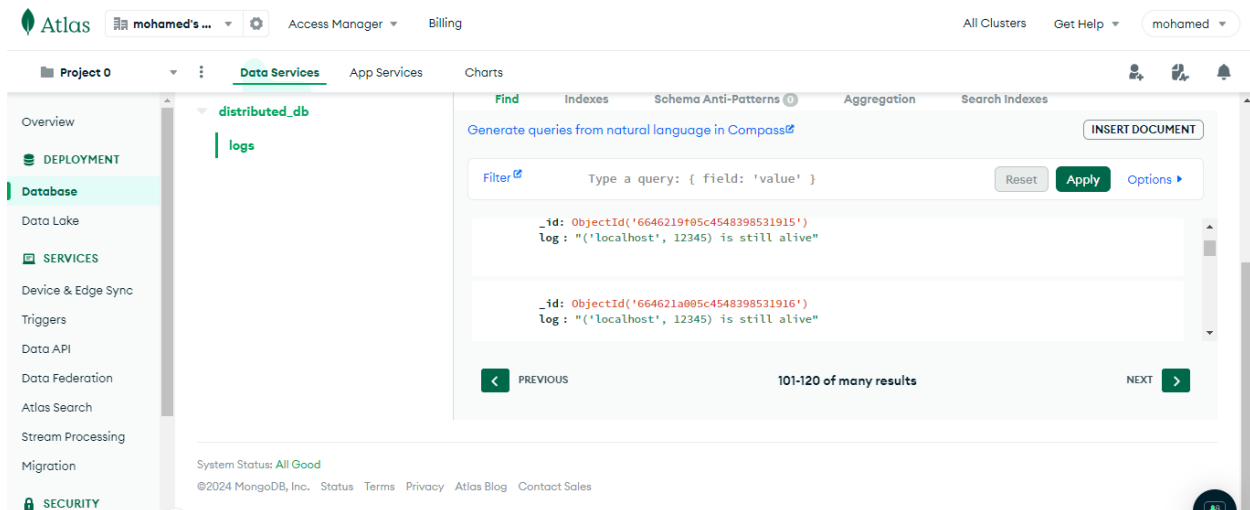


Figure 14 database logs

## Testing:

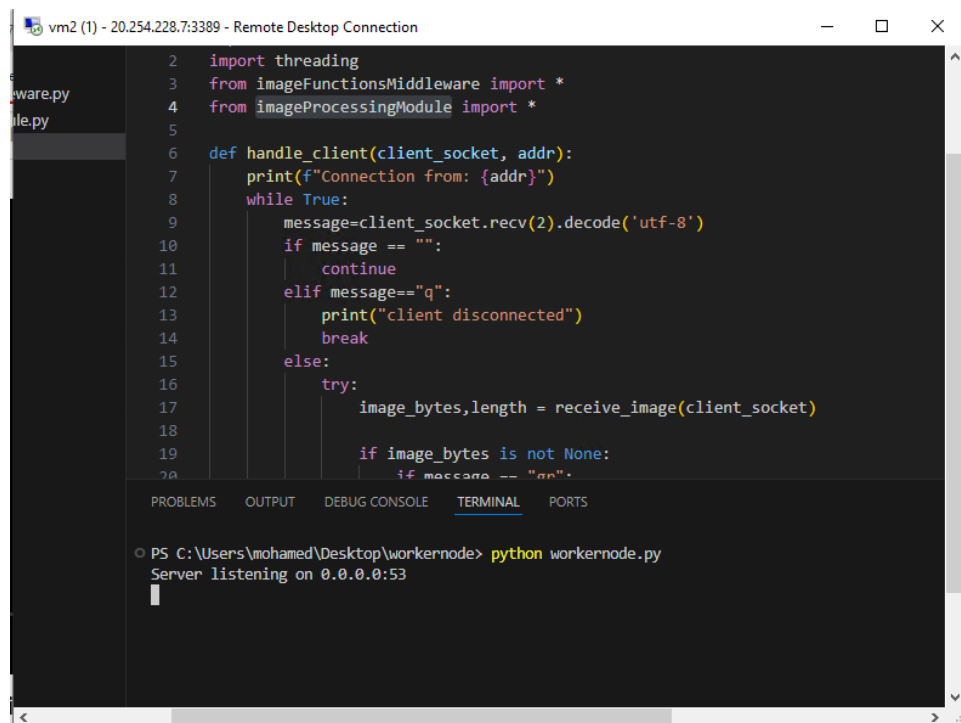
We used integration testing at first for each module by testing each function alone to get its behavior and check its outputs then we integrated the code part by part incrementally.

Then we used manual testing to test the application gui alone first then we connected it to the modules.

Here is the application test after the system integration:

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 nodes on worker vms



The screenshot shows a Remote Desktop Connection window titled "vm2 (1) - 20.254.228.7:3389 - Remote Desktop Connection". The window is divided into two main sections. The top section is a code editor displaying a Python script. The script imports the 'threading' module and imports everything from 'imageFunctionsMiddleware' and 'imageProcessingModule'. It defines a function 'handle\_client' that takes 'client\_socket' and 'addr' as arguments. Inside this function, it prints the connection address and enters a 'while True' loop. In the loop, it receives a message from the client socket, decodes it, and checks for empty messages or a 'q' character to break the loop. Otherwise, it attempts to receive an image from the client socket. The bottom section of the window is a terminal with tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL', and 'PORTS'. The 'TERMINAL' tab is active, showing the command 'python workernode.py' being executed in a PowerShell prompt. The output of the command is 'Server listening on 0.0.0.0:53'.

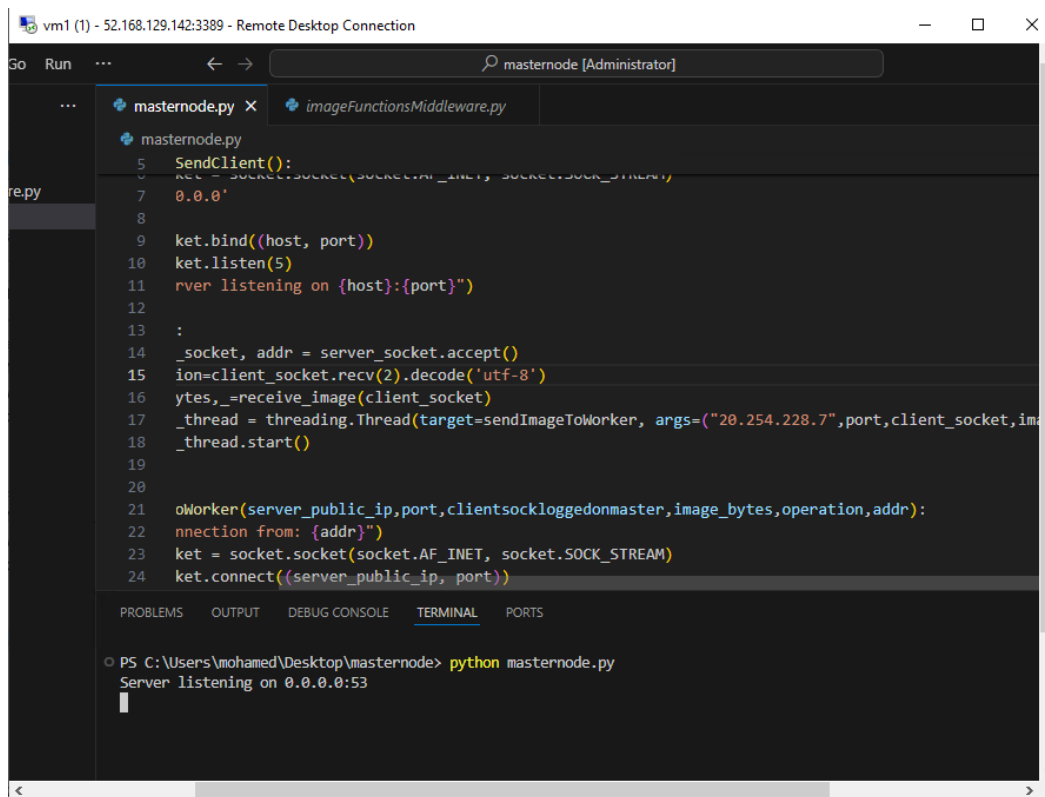
```
2 import threading
3 from imageFunctionsMiddleware import *
4 from imageProcessingModule import *
5
6 def handle_client(client_socket, addr):
7     print(f"Connection from: {addr}")
8     while True:
9         message=client_socket.recv(2).decode('utf-8')
10        if message == "":
11            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 == "q":
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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

Figure 15 running worker node on vm

And we will run the master nodes on another vms



```
vm1 (1) - 52.168.129.142:3389 - Remote Desktop Connection
Go Run ... masternode [Administrator]
masternode.py x imageFunctionsMiddleware.py
masternode.py
5 SendClient():
6 ket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7 0.0.0.0'
8
9 ket.bind((host, port))
10 ket.listen(5)
11 rver listening on {host}:{port}")
12
13 :
14 _socket, addr = server_socket.accept()
15 ion=client_socket.recv(2).decode('utf-8')
16 ytes,_=receive_image(client_socket)
17 _thread = threading.Thread(target=sendImageToWorker, args=("20.254.228.7",port,client_socket,im
18 _thread.start()
19
20
21 oWorker(server_public_ip,port,clientsockloggedonmaster,image_bytes,operation,addr):
22 nnection from: {addr}")
23 ket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
24 ket.connect((server_public_ip, port))
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
```

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

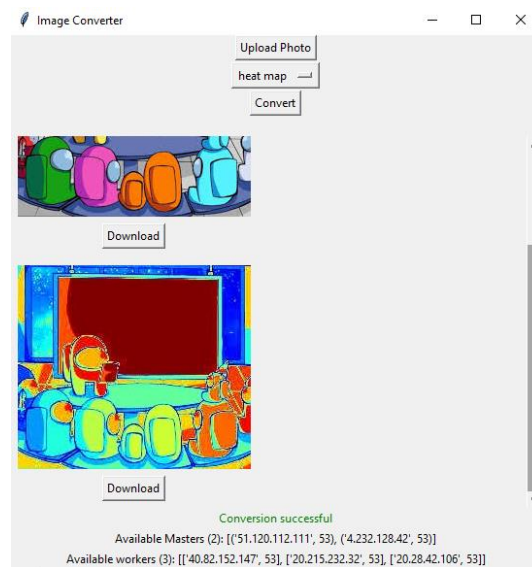


Figure 18 image processed then sent to the client

Here is master node connection from client

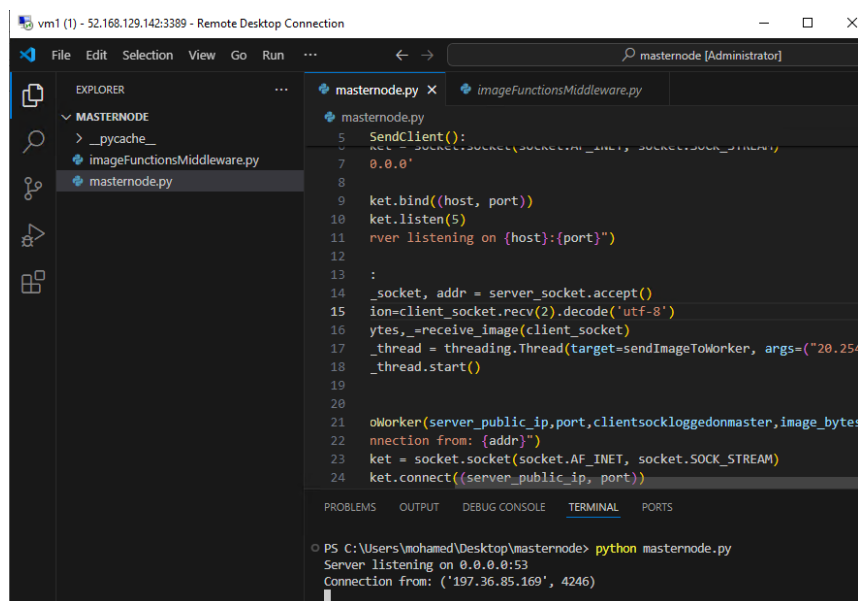


Figure 19 master node connection from client

And this is worker node connection form master node

```

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 20 worker node connection from master node

Trying many photos from the same client

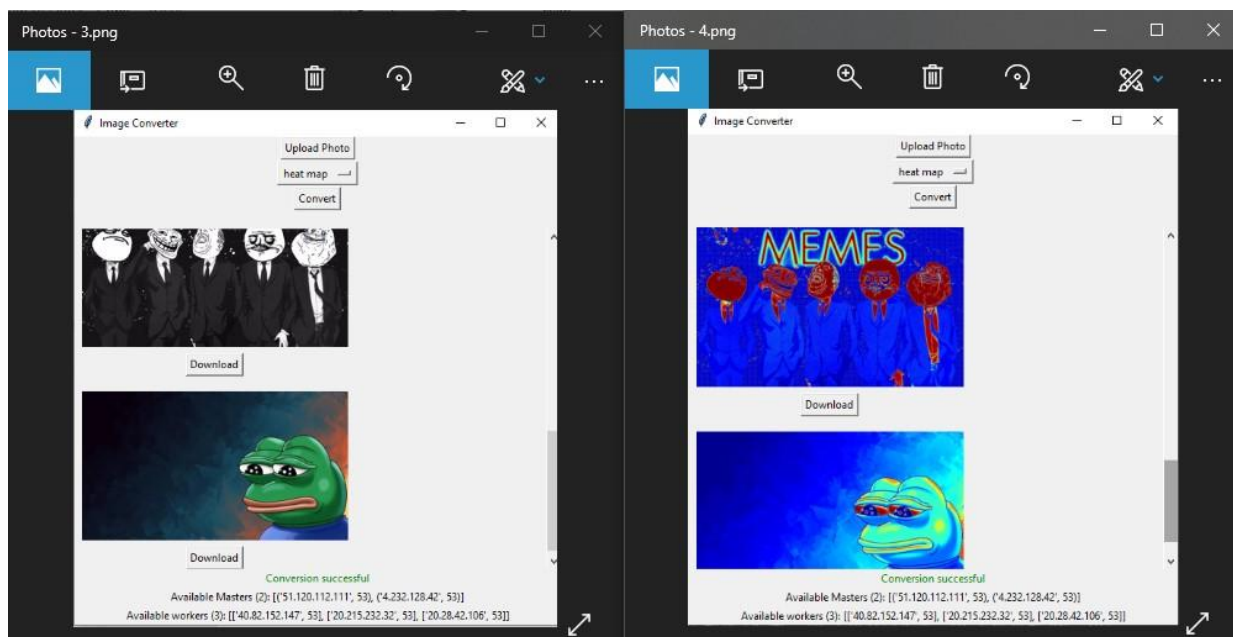


Figure 21 converting many photos from the same client

And this is trying many clients on the same time

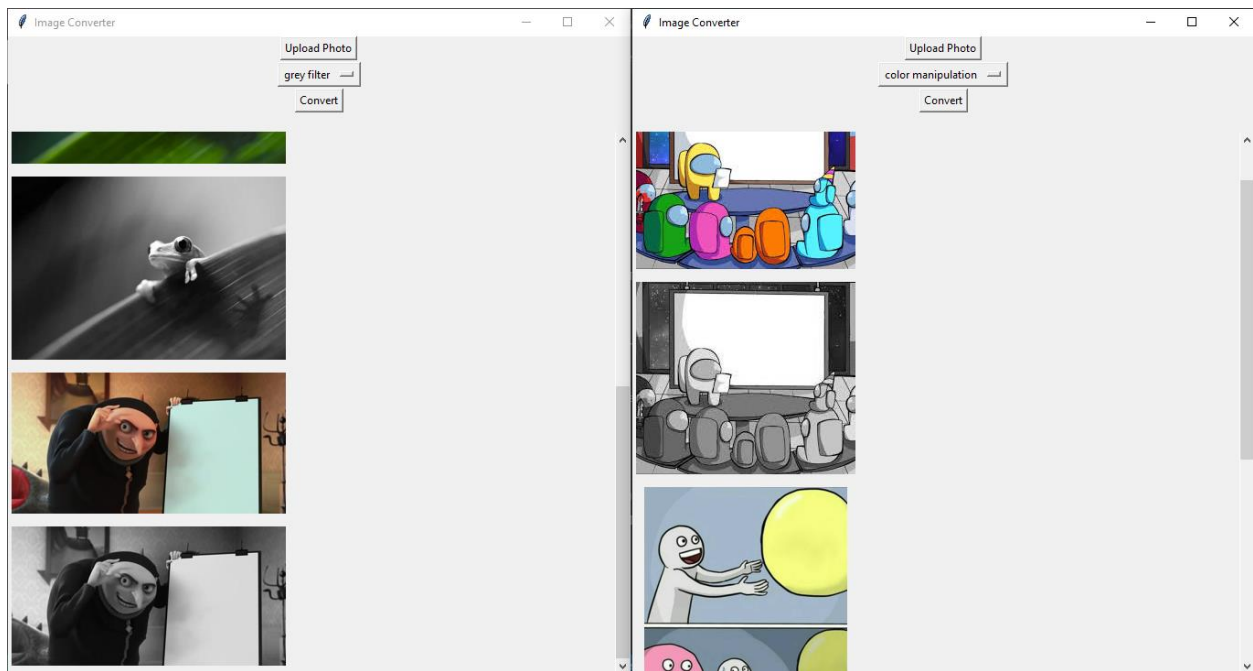


Figure 22 converting many photos from different clients

## 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.



GitHub link:

<https://github.com/Mohamedamr3737/Distributed-image-processing-with-cloud-computing->

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>

<https://www.mongodb.com/docs/>