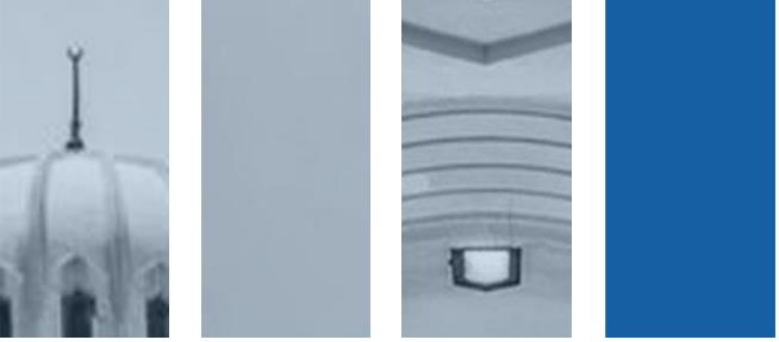

Distributed Image Processing System using Cloud

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:

Table of figures:

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:

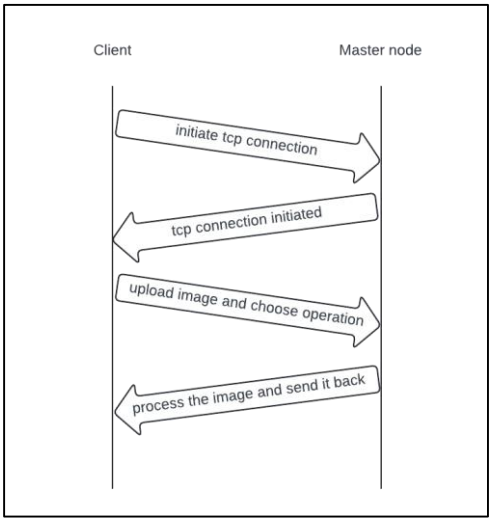

Figure 1 client-master node protocol

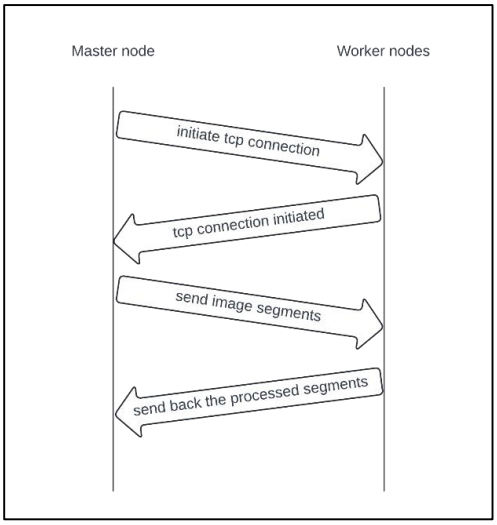

Figure 2 master node-worker node protocol

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:

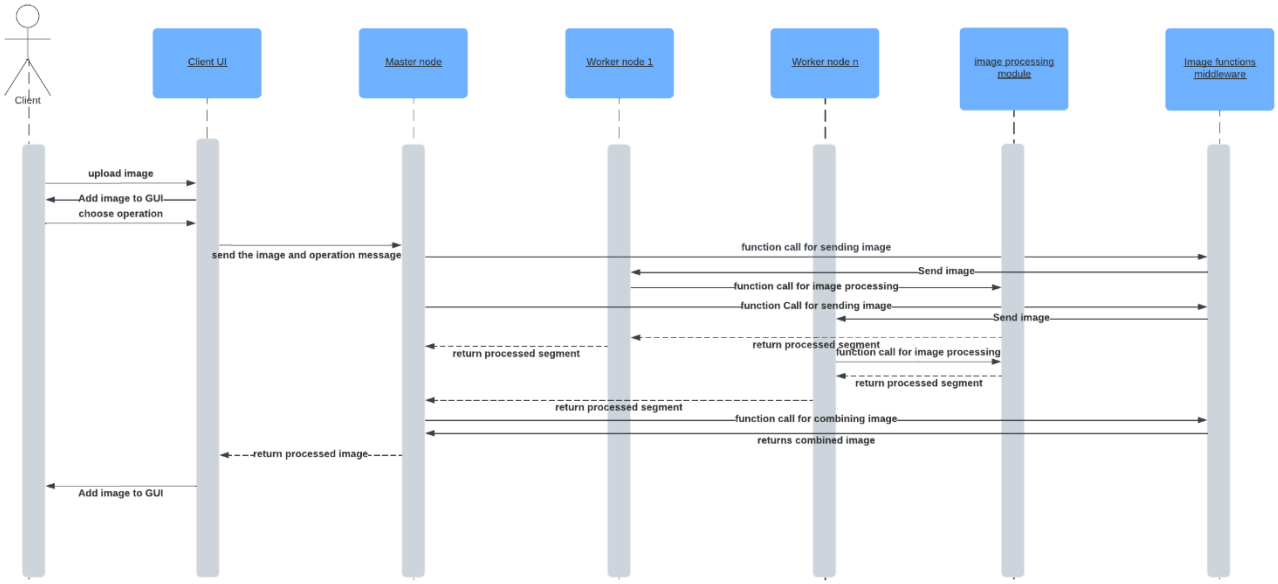

Figure 4 sequence diagram

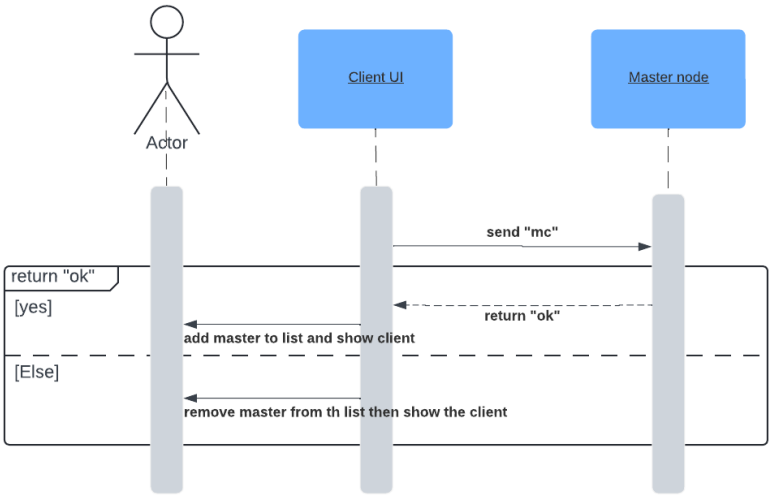

Figure 3 master node monitoring sequence diagram

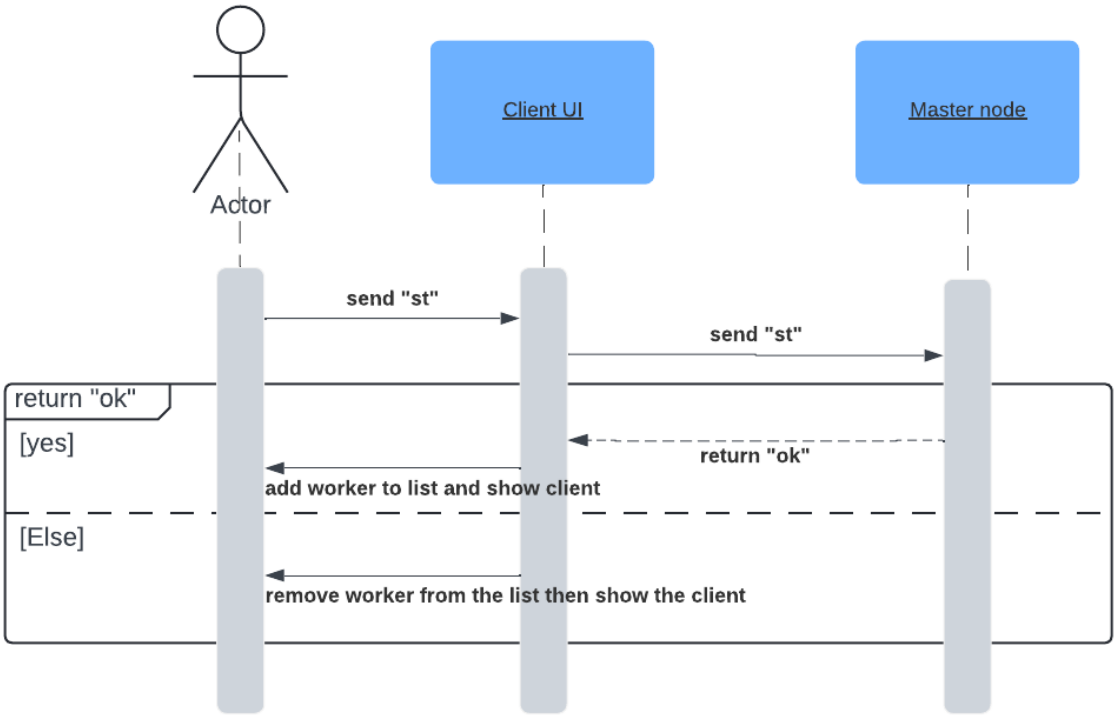

Figure 5 monitoring workernodes sequence diagram

2. Components diagram:

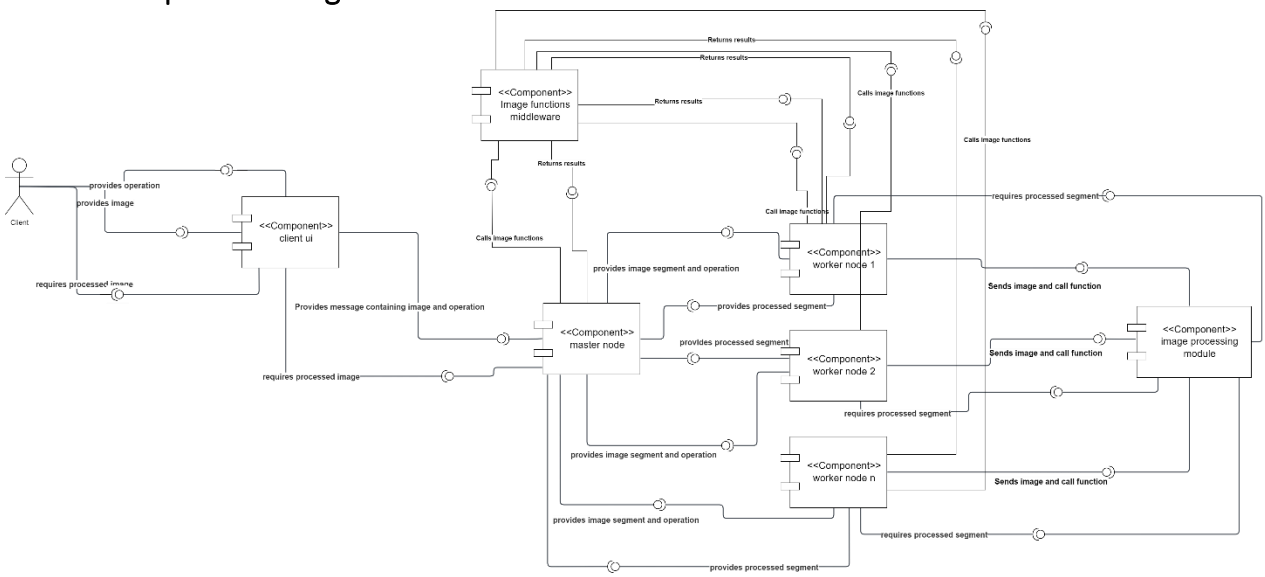

Figure 6 components diagram

3. Infrastructure diagram:

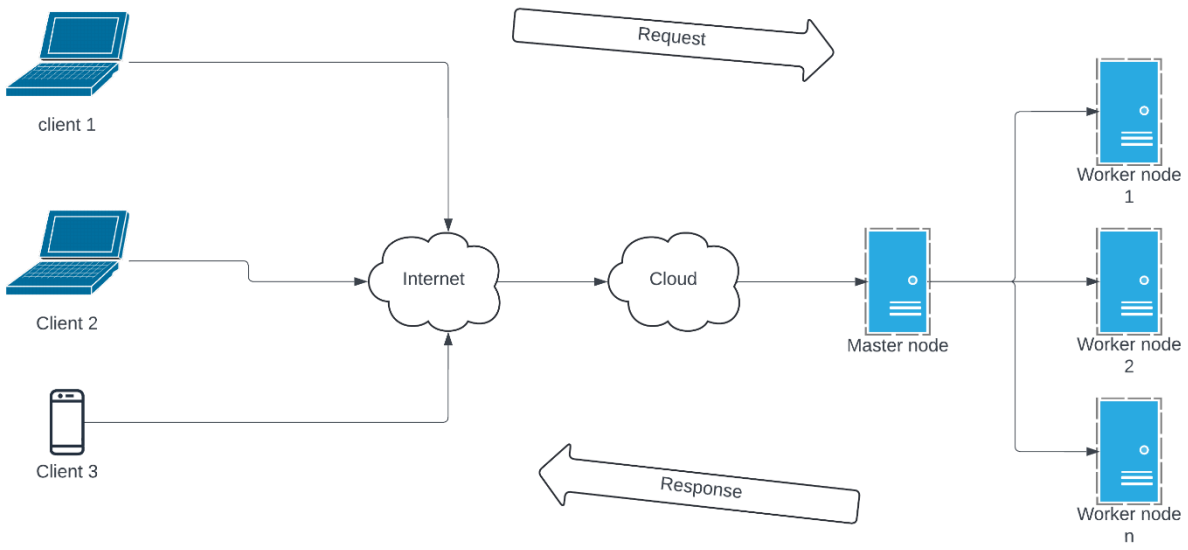

Figure 7 infrastructure diagram

4. Network diagram:

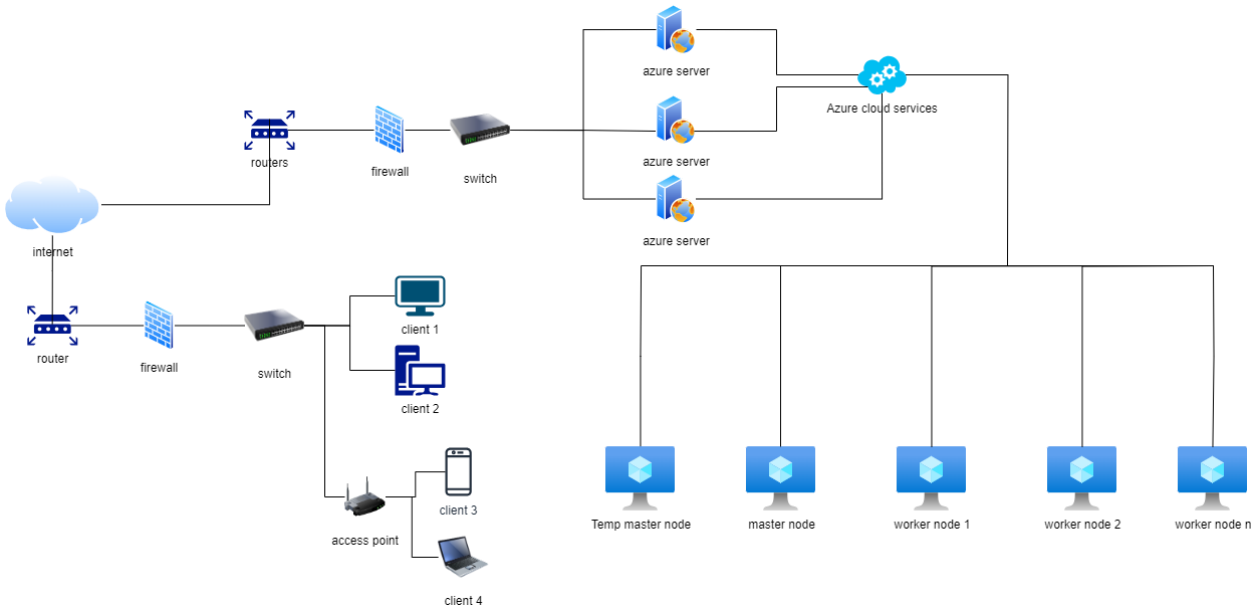

Figure 8 network diagram

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 nodew and the other for the worker nodes:

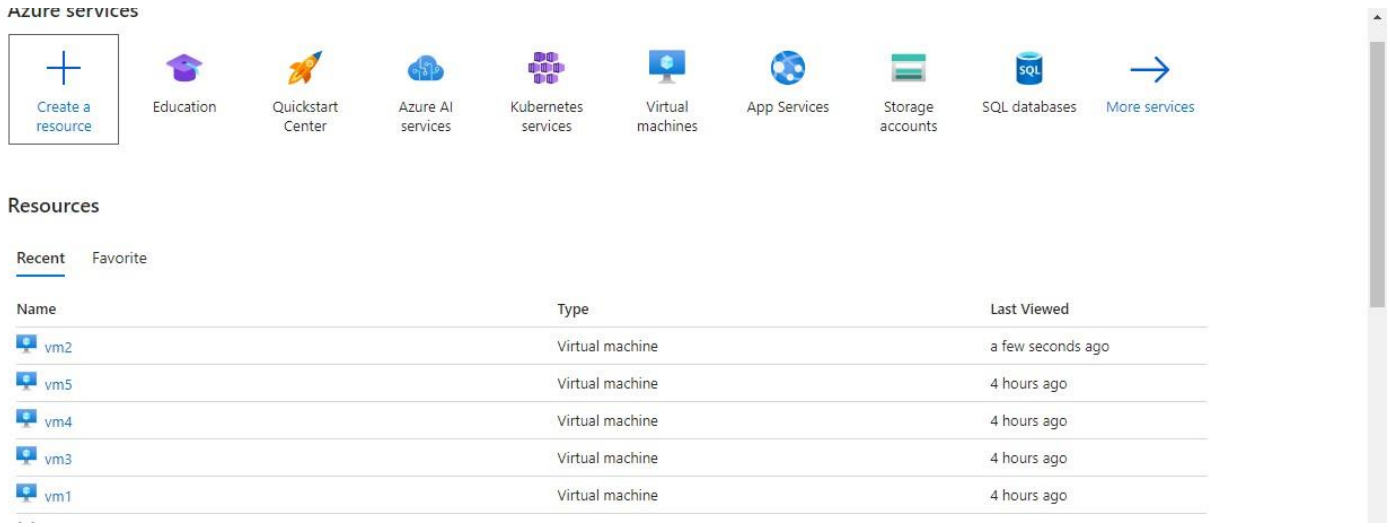

Figure 9 azure resources

Then we downloaded RDP files for both machine to start them

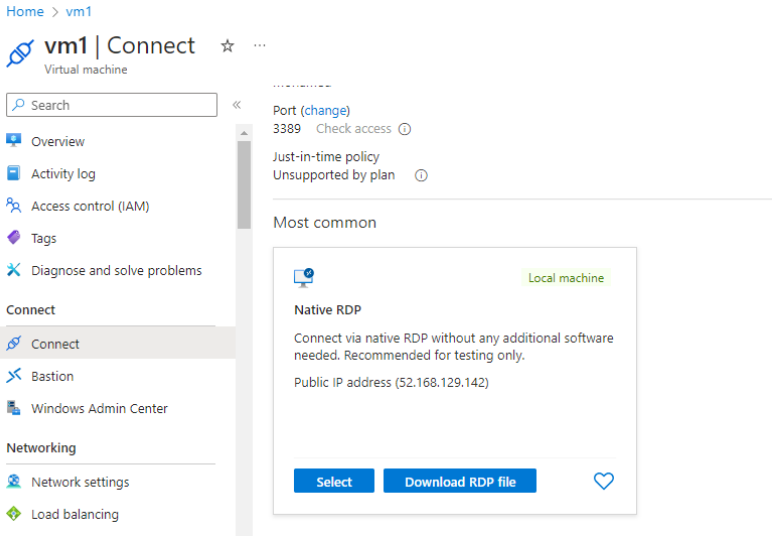

Figure 10 RDP download

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:

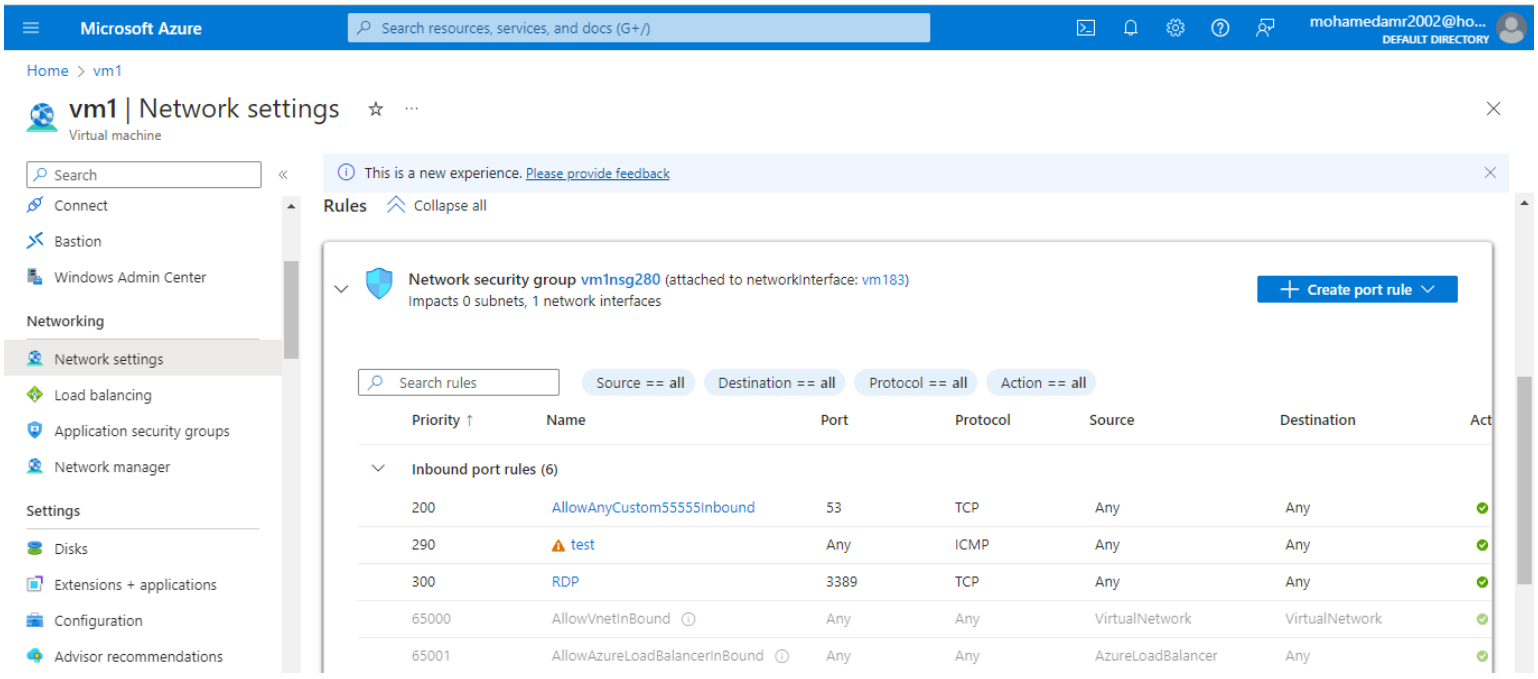

Figure 11 inbound rules

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

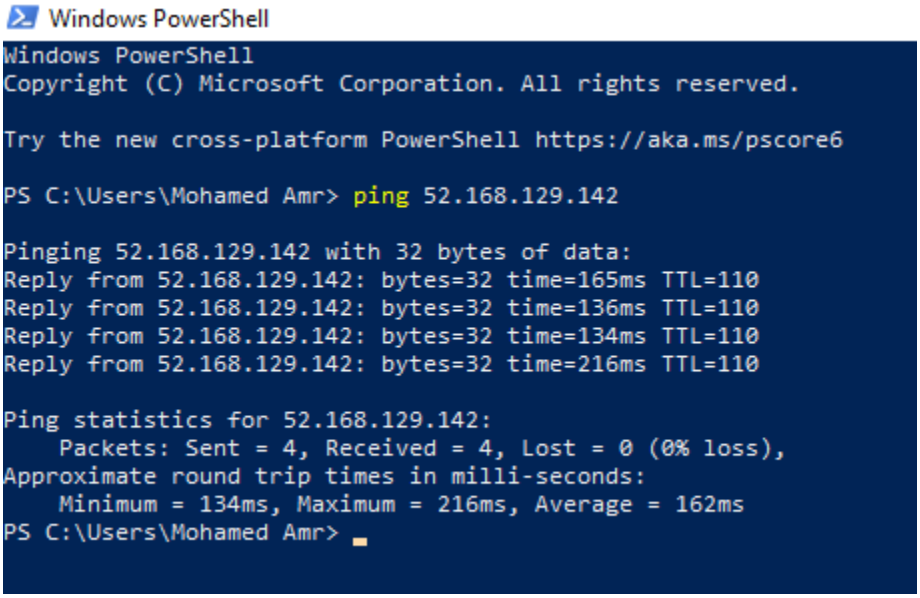

Figure 12 pinging machines

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

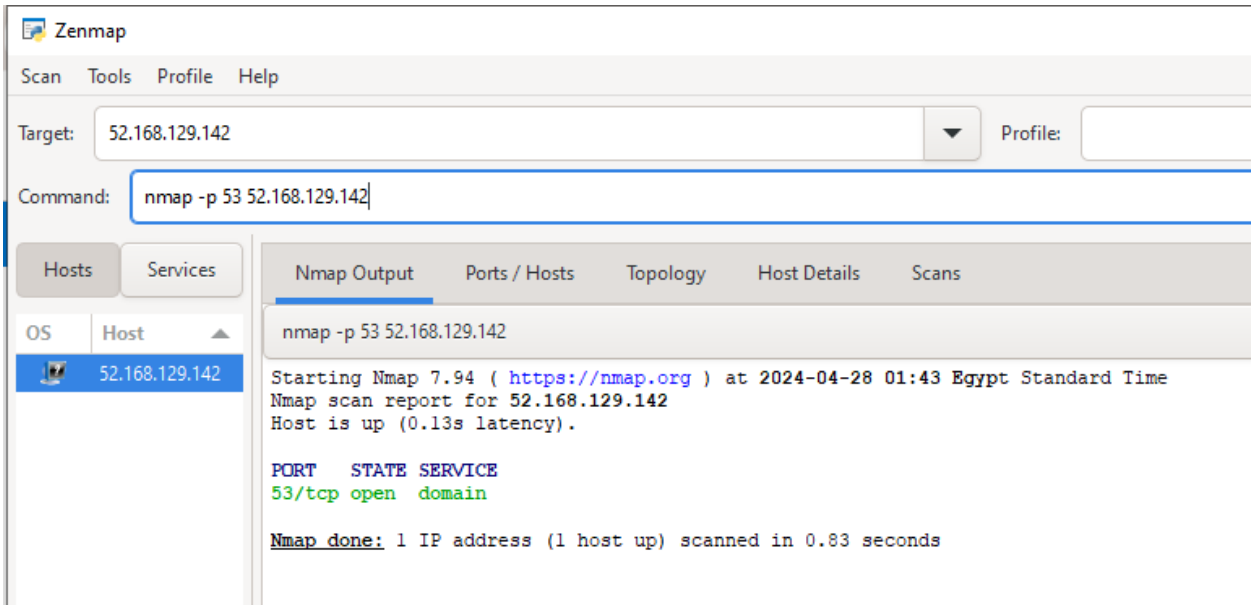

Figure 13 testing the TCP port

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)]#[('lo
calhost',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,
(processed_image_bytes, len(processed_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,por=self.workingmasterslists[0]
        try:
            client_socket.connect((ip, por))
            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.chechWorkingmasters)
        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,por=self.workingmasterslists[0]
            client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client_socket.connect((ip, por))
            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.m1zj0le.mongodb.net/?retryWrites=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. **chechWorkinworkers(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.
- **chechWorkinworkers**: 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 pymongodb online cluster to be able to submit from all the machines but the connection to its 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:

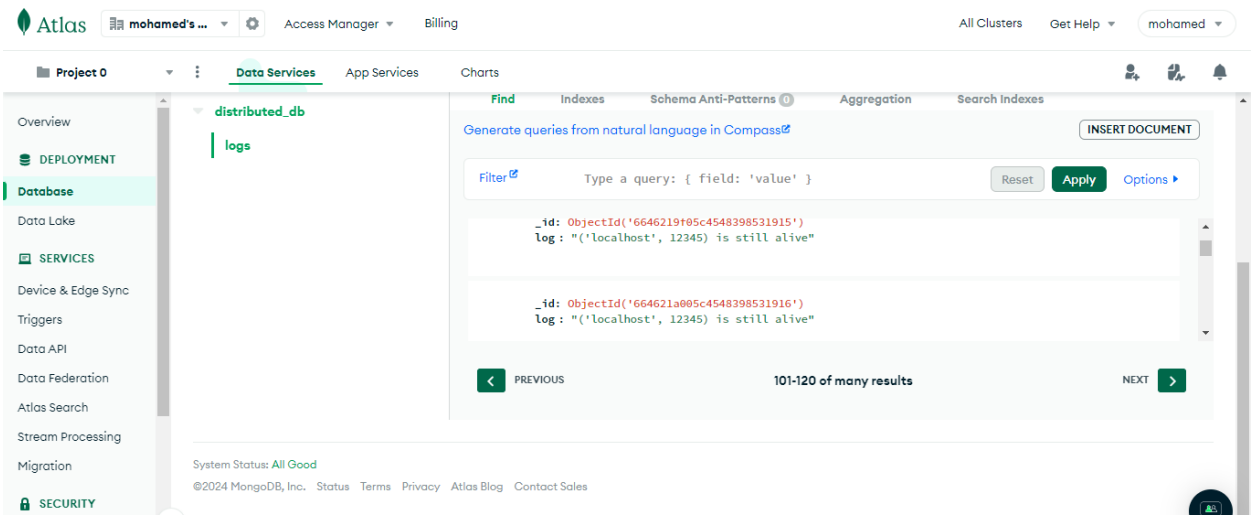

Figure 14 database logs

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

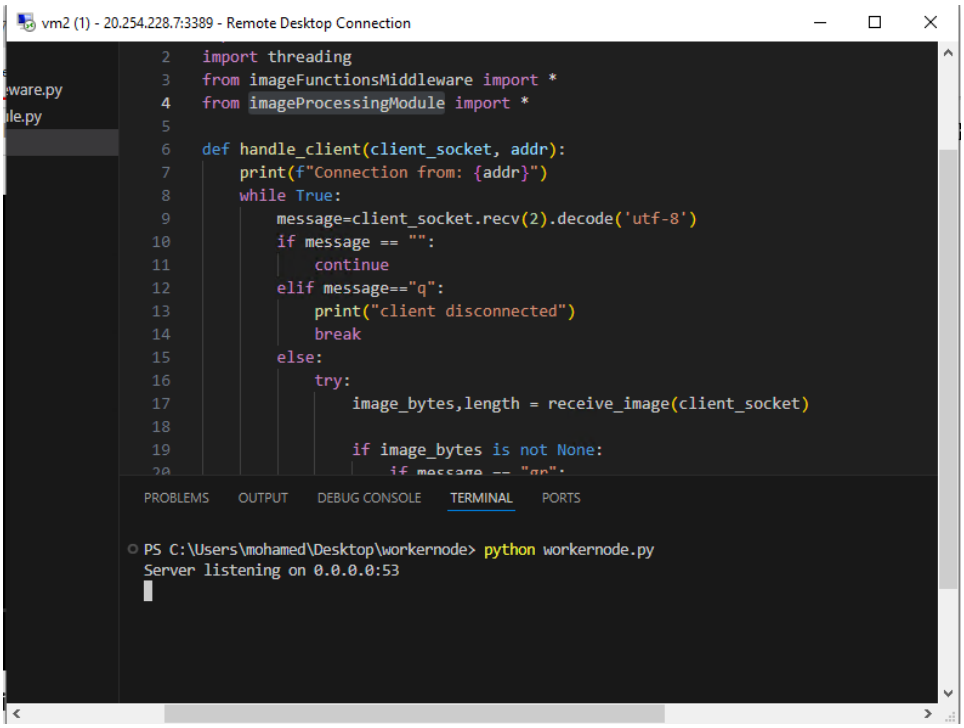

Figure 15 running worker node on vm

And we will run the master nodes on another vms

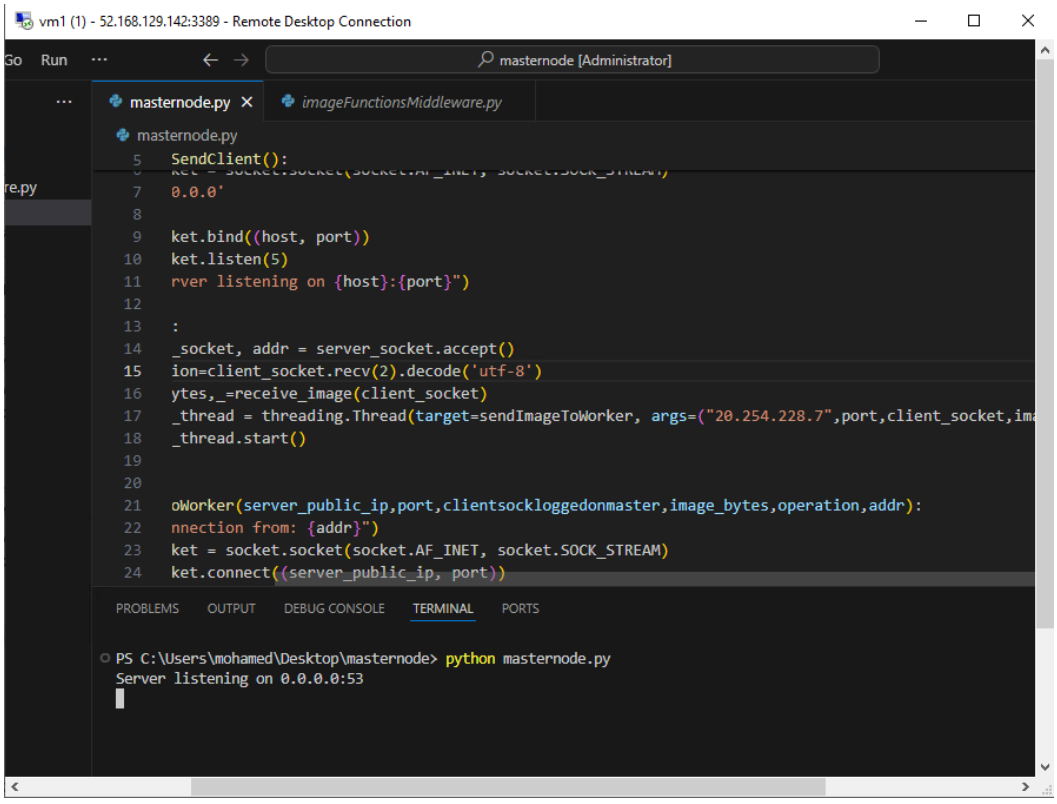

Figure 16 running master node on another vm

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

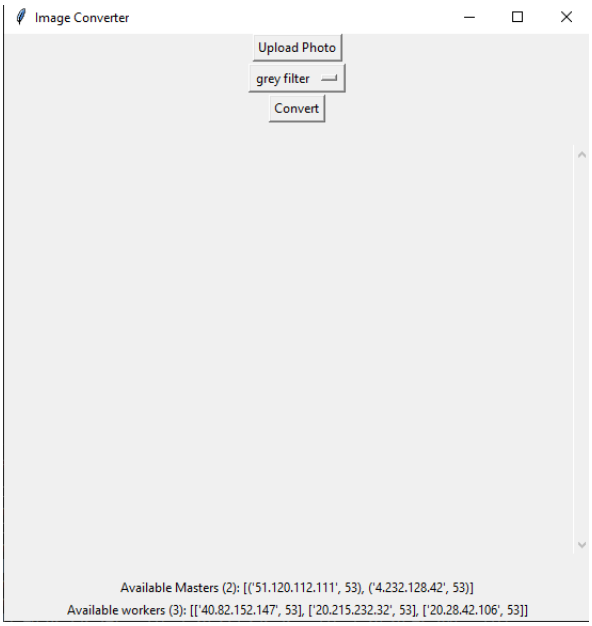

Figure 17 running client GUI on local machine

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

Figure 18 image processed then sent to the client

Here is master node connection from client

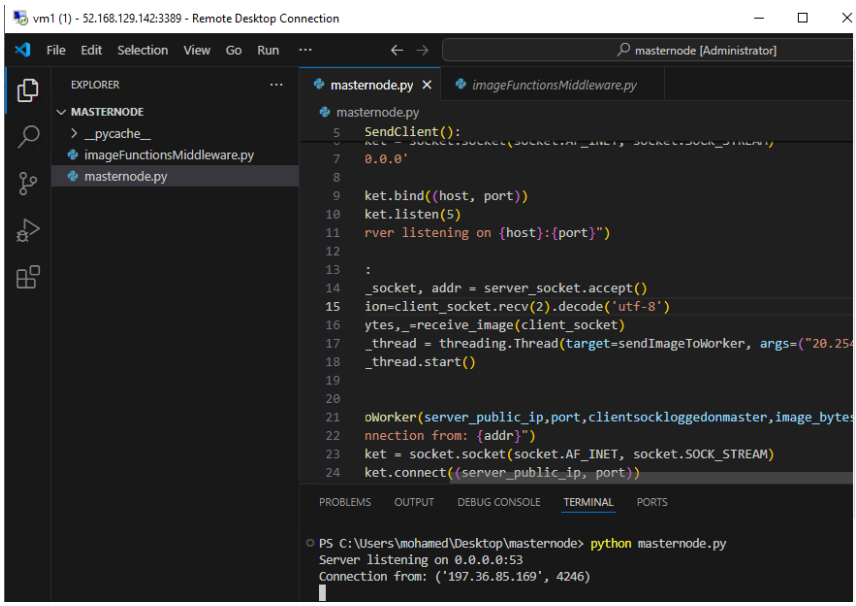

Figure 19 master node connection from client

And this is worker node connection form master node

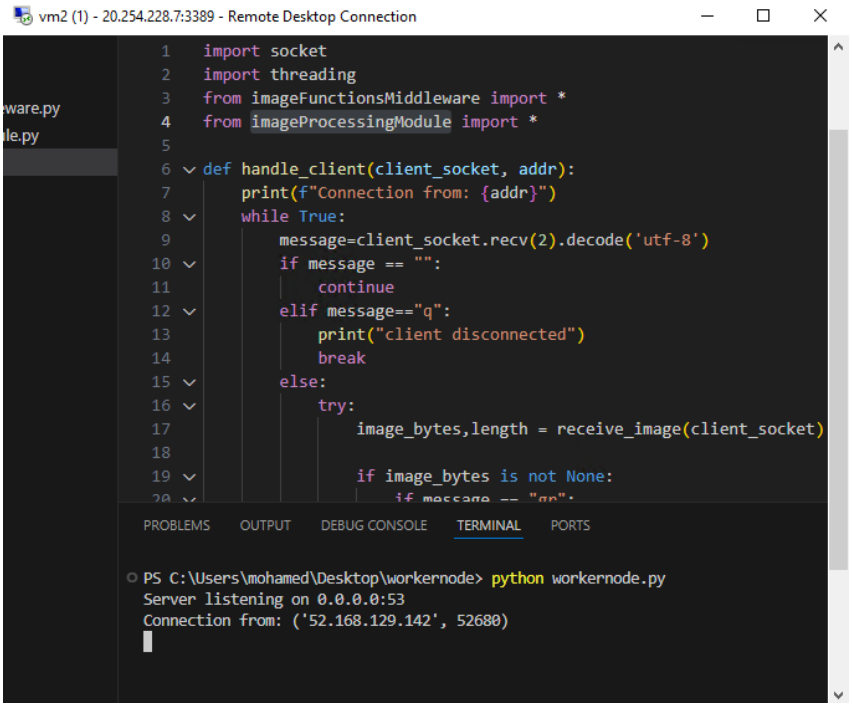

Figure 20 worker node connection from master node

Trying many photos from the same client

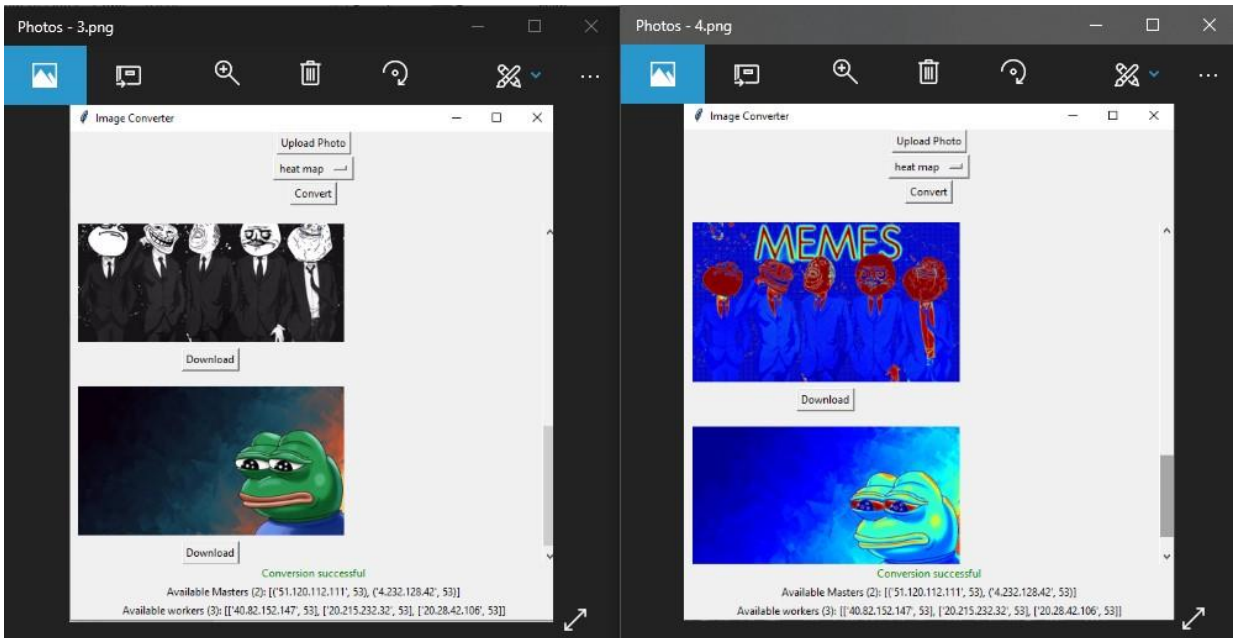

Figure 21 converting many photos from the same client

And this is trying many clients on the same time

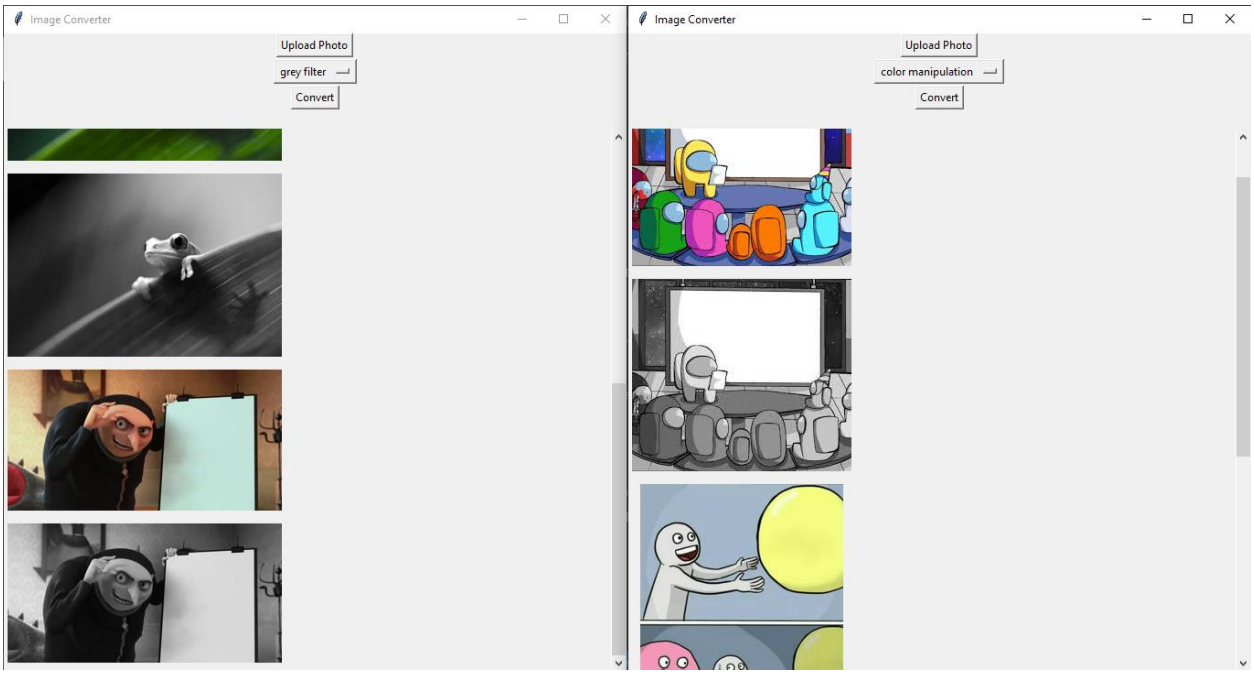

Figure 22 converting many photos from different clients

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/